Diploma Thesis

A Scalable Coarsening Phase for a Multi-Level Graph Partitioning Algorithm

Manuel Holtgrewe

August 26, 2009

Supervisor: Prof. Dr. rer. nat. Peter Sanders

Für meine Eltern.

Acknowledgement

I would like to thank my supervisor Prof. Dr. rer. nat. Peter Sanders for his support, inspiring discussions and support.

Furthermore, I want to thank everyone who was involved in my work on this thesis and my friends. Without their invaluable support, suggestions and critical questions, it would have been impossible for me to complete. My special thanks go to Christian Schulz for his help with the integration of his partition refinement algorithm into my uncoarsening code.

Ich erkläre hiermit, dass ich die vorliegende Arbeit selbständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel verwendet habe.

Karlsruhe, August 26, 2009

Manuel Holtgrewe

Abstract

Graph Partitioning is an important load balancing problem in parallel processing. The simplest case of graph partitioning is as follows: Given a graph $G = (V, E)$ and an integer k, the vertex set is to be partitioned such that each partition block has the same size and minimize the edges adjacent to vertices in different blocks. The problem can be extended to graphs with weighted vertices and edges.

Since the graph partitioning problem is $\mathcal{NP}$-hard, real-world problem instances are solved using approximation algorithms. Beginning from the mid 1990s, the most successful practicable codes use a multi-level approach.

We present a scalable coarsening phase for a distributed memory, parallel multi-level partitioner and an experimental evaluation thereof. The main contribution is using high-quality matching algorithm with objective functions (which we call *edge ratings*) different from the edge weight. Also, the algorithm for finding matchings of vertices that are not local to one process is more advanced than other partitioning codes we are aware of.

We identify five promising edge ratings. Two of them yield better initial partitions of the coarsest graph in terms of edge cut than the one of KMETIS. We are using approximate matching algorithms that are more expensive in terms of running time. However, we achieve a lower running time of our coarsening algorithm than PARMETIS from 256 or 512 processes for large graphs, depending on the graph.

8

Contents

Contents

1. Introduction

Modern life would be unthinkable without computers. We are constantly reminded of this fact by interacting with and through devices such as mobile phones and personal computers

Regardless of how powerful these small computers are today, a lot of challenges ask for larger machines. Some examples:

- Weather forecast would not be as precise as it is today without huge numerical models using a lot of data.

- The design process of machines would be much more expensive without simulation of the machines in the computer. Think of car crash tests, for example.

- Indexing the internet and its 20 billion webpages ([12]) calls for more processing power than one desktop computer can offer.

Huge problems such as the three ones listed above are solved using massively parallel computers. Today, the most powerful parallel machines consist of a huge number of small, cheap machines called *nodes* ([65]). Each node has its own independent hard drive, its own memory and buses to connect its components. The nodes are then connected using a network. Such a machine is called a *distributed memory* computer.

Each node has to store the data it needs for its computation in its main memory. It must use the network for communication whenever it needs to access a piece of data which is located in another node's memory. Communicating through the network is at least one order of magnitude slower and has a higher latency than accessing main memory directly. Thus, the distribution of the data on the parallel computer is important. One important aim is to minimize the amount of communication and make the nodes as independent of each other as possible.

If one node solved the whole problem on its own then the amount of communication would be minimal: None at all. Obviously, this is not efficient. For this reason, one considers *load balancing* restrictions: Each node should do the same amount of work to lower the total time spent solving the problem.

A lot of large-scale problems solved on distributed memory machines have a large graph as their main data structure. The graph of all links between web pages is one example, another one is social networks. For other large classes of problem solvers, the communication requirements naturally map to a graph of communication partners. Examples for this is the finite element method and linear equation system solvers.

The topic of this thesis is *A Scalable Parallel Coarsening Phase for Multi-Level Graph Partitioning*. Graph partitioners are used by large-scale problem solvers to distribute the

graph on a shared memory machine. The objective is to minimize communication while satisfying load balancing requirements. The aim of this thesis is to engineer (design, implement and evaluate) the first of three phases for the multi-level approach to graph partitioning.

1.1. Our Contribution

We present a scalable coarsening phase for a distributed memory parallel multi-level graph partitioning algorithm.

Our main contribution is

- using more advanced algorithms for approximate maximum weight matching in the context of a parallel multi-level graph-partitioning algorithm and

- using different objective functions for edge weights (called *edge ratings*).

To the best of our knowledge, other parallel graph partitioners use the HEAVY-EDGE-MATCHING algoritm (see Section 5.2) or variations thereof.

Our implementation uses the GLOBAL-PATH-ALGORITHM from [49] for computing an approximate maximum weight matching on local edges. Then, it approximates a maximum weight matching on the edges between processes using a variant of the algorithm proposed in [48]. The main difference to the matching algorithms used in other parallel graph partitioners we are aware of is the usage of ghost vertices to reduce the necessary amount of communication.

We evaluate our implementation experimentally on a collection of graphs arising in Finite Element Method applications, road networks and sparse matrices. In many cases, two of our proposed edge ratings yield better results in terms of edge cut of the partition of the coarsest graph than using the edge weight.

Additionally, although the local matching algorithm is more expensive in terms of running time, our coarsening scales better than the one of PARMETIS on graphs from various classes: We observe a higher running time for PARMETIS than for our algorithm from 256 and 512 processes for large graphs, depending on the graph.

1.2. Overview Over This Thesis

Chapter 2 introduces some fundamental facts, notations, nomenclautures, and definitions used throughout this thesis. Chapter 3 cites related work, mainly regarding algorithms for partitioning graphs or finding matchings in graphs. Chapter 4 describes our coarsening and uncoarsening algorithms. Chapter 5 describes the matchings in our implementation and gives a detailed formulation of the parallel algorithm sketched in [48]. Chapter 6 shows our experimental results. Finally, Chapter 7 summarizes this thesis and gives an outlook on future work.

2. Fundamentals

As usual, $\mathbb{N}$ is the set of natural numbers, starting with 0 and $\mathbb{R}$ is the set of real numbers. If S is a set, $|S|$ is its cardinality.

Given a set S, a *partition* of S of order k is a sequence of disjoint sets $\langle S_1, \ldots, S_k \rangle$ such that $S = \bigcup_i S_i$. The sets S_i are called *blocks*.

The symbol "log" denotes the logarithm with base 2.

We use square brackets to denote the characteristic function of a predicate P: $[P]$ is equal to 1 if P is true and 0 otherwise.

2.1. Graphs

In this thesis, we mostly consider undirected graphs. An *(undirected) graph $G = (V, E)$* consists of its set of vertices V and its set of edges E. We denote the number of vertices as $n = |V|$ and the number of edges as $m = |E|$. Without loss of generality, we assume that the the vertices are subsequent numbers starting from 0, thus $V = \{0, \ldots, n-1\}$. E is a subset of $\{\{u, v\} : u, v \in V, |e| = 2\}$.

A *weighted graph* is a graph with a weight function w that assigns a non-negative weight to each vertex and edge. We also define $\omega : 2^{V \cup E} \to \mathbb{R}^{\geq 0}$ that assigns sets of vertices and edges the sum of the weight of their elements. Each unweighted graph also is a weighted graph with weight function $w \equiv 1$.

In *directed graphs*, the set of edges is a subset of $V \times V$. We are not concerned with graphs having self-loop edges (u, u).

When representing undirected graphs, directed graphs must have edges (u, v) and (v, u) for each edge $\{u, v\}$ in the undirected graph. In the case of representing weighted undirected graphs, we also require that $w(u, v) = w(v, u)$ holds for all edges in the directed graph.

Usually, we use the letters u, v, x, y and z for vertices and the letters e, f and g for edges.

According to [14], a family $F = \langle G_0, G_1, \ldots \rangle$ of graphs is *sparse* if the number of edges is linear in the number of vertices, i.e. $m \in \Theta(n)$. A family of graphs is *dense* if the number of edge is quadratic in the number of vertices, i.e. $m \in \Theta(n^2)$. We relax this notion a bit and consider graph families with $m \in O(n \cdot \log n)$ as sparse, too. We also introduce the notion of a sparse graph (and not a family) if it has "few" edges.

The distance between two vertices u, v in a graph G is defined as the number of vertices on the shortest path between u and v in G ([14]).

The neighbours of a vertex v are denoted by $N(v)$.

2.2. Parallel Algorithms

The nomenclauture for classifying parallel algorithms is not consistent throughout literature. We will call algorithms for the various variants of the PRAM *parallel*. Furthermore, for processing graphs, mostly two models of computation are considered.

First, one can provide one processor for each vertex, communication between processors is explicit. It has to be taken into consideration in the analysis in terms of number of messages and communication volume. We will call algorithms for this model *distributed* algorithms.

Second, we can provide a fixed number p of processors that is independent of the problem size. Communication between processors is still explicit and has to be analyzed in terms of number of messages and communication volume. We call these algorithms *parallel*. For our needs, the second model is the more interesting one since the problem size will be much larger than the number of processors we use.

In parallel computing, the notion of *scalability* is of some importance (also see [24, page 114]). Ideally, a problem of size n and running time $T(n)$, can be solved in time $T(n)/p$ using p processors. Of course, according to Amdahl's law, the possible speedup of a program/algorithm is limited by the sequential proportion of the program/algorithm. However, if an algorithm/program is able to achieve "sufficiently good" speedups up to a certain p', we say that it shows good *strong scalability* up to this number.

Besides trying to get shorter running times for a fixed problem size, parallel processing can be used to process larger amounts of data in the same time. Taking this point of view, we say that a program shows good *weak scalability* (up to a certain count of processors p'), if a "sufficiently larger" amount of data can be processed in the same time for each added processor.

2.3. Matchings

In this section, we will define terms related to matchings and give some examples. These definitions follow [8] and [46].

Definition 2.3.1 ((Maximal) Matching): Given a graph $G = (V, E)$, a matching $M \subseteq E$ of G is a set of edges such that no pair of edges from M has a common adjacent vertex. Formally: $M \subseteq E$ is a matching if and only if for all $e_1 = \{u, v\}, e_2 = \{x, y\} \in M : e_1 \neq e_2 \Rightarrow e_1 \cap e_2 = \emptyset$.

A matching M is *maximal* if it cannot be augmented with an edge from E without violating the requirement above.

The *Maximal Matching Problem* is defined as finding a maximal matching for a given graph. ◇

Definition 2.3.2 (Maximum Weight Matching): For a weighted graph G with weight function w, we define the weight of a matching M as the sum of the weights of its edges: $w(M) := \sum_{e \in M} w(e)$.

(a) A maximum match-
ing M_1 for G.

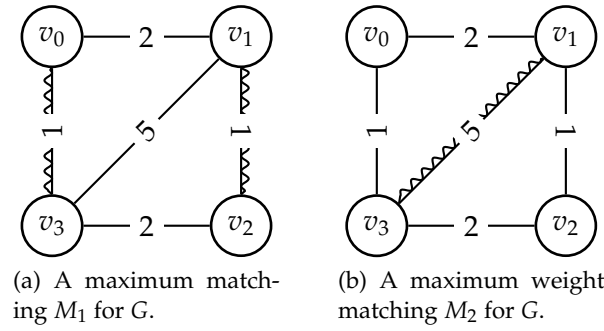(b) A maximum weight
matching M_2 for G.

Figure 2.1: Examples of maximal and maximum weight matchings.

A matching is said to have maximum weight if there is no matching with a greater weight.

The *Maximum Weight Matching Problem* is defined as finding a maximum weight matching for a given weighted graph. ◇

Figure 2.1 (p. 15) shows the difference between a maximal and a maximum weight matching. Both figures show the same weighted graph. The matched edges are marked with "snake lines". Figure 2.1a shows a maximal matching M_1 of graph G whereas Figure 2.1b shows a maximum weight matching M_2. Note that for G, M_1 is no maximum weight matching and M_2 is no maximal matching.

Both, the Maximum Weight Matching and the Maximum Matching Problem are in $\mathcal{P}$ (also see Section 3.3).

In the rest of this thesis, we relax the notion of matching as follows: A matching is not a set of edges but of unordered pairs of vertices. An equivalent point of view is that every graph is a weighted complete graph. Non-existent edges are interpreted as edges with weight zero.

2.4. Graph Partitioning

This section defines the Graph Partitioning Problem as stated in [19].

Definition 2.4.1 (Graph Partitioning, Unweighted): Given a graph $G = (V, E)$ and an integer $k \geq 2$, find a partition $\mathcal{S} = \langle S_1, \ldots, S_k \rangle$ of V of order k such that

- the block sizes are balanced, i.e. $|S_i| \leq (1 + \varepsilon) \cdot |V| / k$ for $i = 1 \ldots k$ and a small ε.

- the *cut size* is minimized.

The cut size of a partition is defined as the number of edges across blocks:

$$\left| \{ \{u, v\} \in E : i \neq j \wedge u \in S_i, v \in S_j \} \right|$$

The *balance* of a partition is defined as $|S_{\max}| / \lceil |V| / k \rceil$, i.e. the ratio of the size of the largest block in the partition and the largest block in a perfectly balanced partition. ◇

We can easily extend this definition to weighted graphs.

Definition 2.4.2 (Graph Partitioning, Weighted): Let G, k and $\mathcal{S}$ be defined as in Definition 2.4.2, however let G be weighted with weight function w. Find $\mathcal{S}$ such that

- the block weights are balanced, i.e. $W(S_i) \leq (1 + \varepsilon) \cdot W(V)/k$ for $i = 1 \ldots k$.

- the *cut size* is minimized.

In this case, the cut size is the sum of the weights of the edges crossing blocks. W is the extension of w on sets of vertices: $W(S \subseteq V) = \sum_{v \in S} w(v)$.

The *balance* of such a partition is defined as $W(S_{\max})/\lceil W(V)/k \rceil$, i.e. the ratio of the weight of the largest heaviest in the partition and the weight of the heaviest block in a perfectly balanced partition. $\diamond$

The graph partitioning problem is known to be $\mathcal{NP}$-hard ([19]).

The edge cut always has to be considered together with the balance of a partitioning because they are closely interdependent. An improvement in one metric is easy when ignoring the other one.

When using graph partitioning for load-balancing, the workload is represented by the vertex weights. Dependencies between two entities (and thus communication) is represented by edges. Besides the balance and the edge cut, the literature also contains various other metrics such as *domain shape/aspect ratio*, see for example [71].

Figure 2.2 (p. 17) shows examples for the partitioning of a graph. Vertices from the same block have the same color. Figure 2.2a shows a partition of order 3 on an unweighted graph. The optimal maximal number of vertices each partition is 4, but the gray partition has 5 vertices. Thus, the balance is 1.25. The edge cut is 11. Figure 2.2b shows a partition of order 2 on a weighted graph. The balance is 1, the edge cut is 9.

2.5. Coarsening Fundamentals

In this section, we give the definitions we need for the coarsening of the graph.

Definition 2.5.1 (Contraction of a graph by an edge): The contraction of a graph $G = (V, E)$ with weight function w by an edge $e = \{u, v\}$ is the new graph $G' = (V', E')$ with weight function w'. The set of new vertices V' is defined as $V \setminus \{u, v\} \cup \{x\}$ where x is a new vertex. $w' \equiv w$, except that w' is not defined at u or v and $w'(x) := w(u) + w(v)$.

The set of new edges E' is defined as follows: The edge e is removed. Each edge that was adjacent to u or v and to another vertex z is removed and instead, an edge $\{u, z\}$ or $\{v, z\}$ is inserted. In the case where such an edge already exists, the weight function gives the sum of the two previous edges as the weight of the new edge. The new edge now represents two original edges. $\diamond$

Definition 2.5.2 (Contraction of a graph by a set of edges): The contraction of a graph $G = (V, E)$ by a set of edges M is defined as the result after iteratively contracting all edges in M. $\diamond$
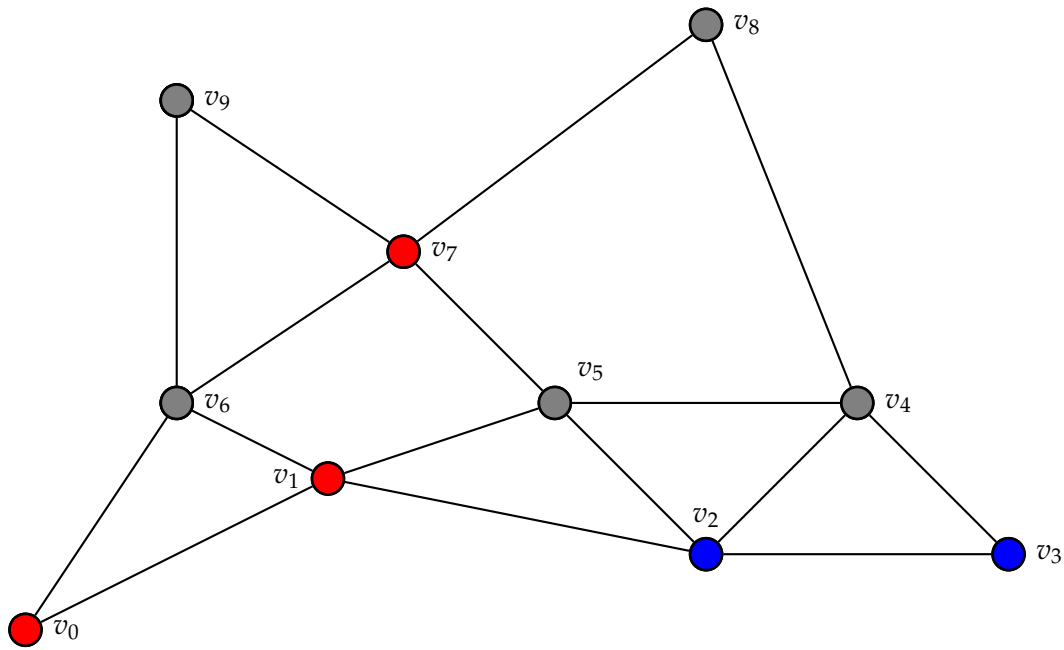
(a) A suboptimal partition of order 3 for a an weighted graph. The vertices in the same block have the same color. The balance is 1.2, the edge cut is 11.
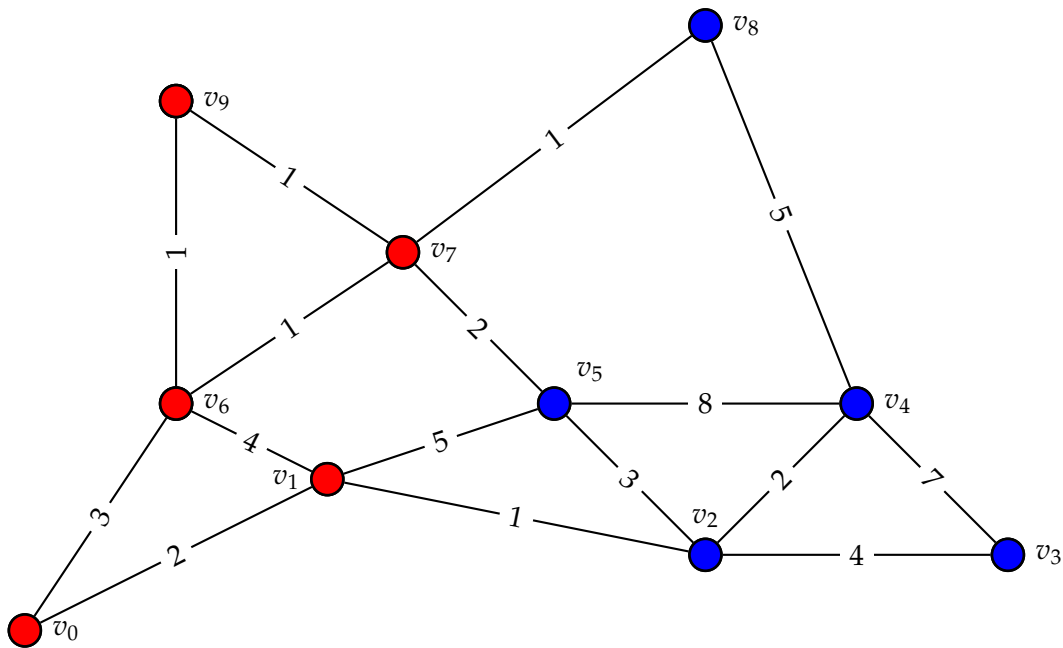
(b) A partition of order 2 for a a weighted graph. The vertices in the same block have the same color. The balance is 1, the edge cut is 9.

Figure 2.2: Examples of graph partitions.

We can generalize Definitions 2.5.1 and 2.5.2 as follows:

Definition 2.5.3 (Contraction of a graph by a function): $G = (V, E)$ is a graph and f is a mapping $f : V \rightarrow V'$. The result of the *contraction of G using f* is $G' = (V', E')$. We define

$$E' = \{\{f(u), f(v)\} : u, v \in V, f(u) \neq f(v)\}.$$ ◇

Besides storing weights for edges and vertices, we also tally the sum of all internal edges for each vertex. In the original graph, the *internal edges* measure is 0. When contracting an edge $\{u, v\}$ of weight a, vertex x will have the sum of a and the internal edge values of u and v as its new internal edges value.

Furthermore, we consider the inverse f^{-1} of f. Since f does not have to be an injection, we define $f^{-1}(y)$ as $\{x : x \in V, f(x) = y\}$.

2.6. Miscellaneous Definitions

In the experimental evaluation, we will compare different quality metrics which are to be minimized. The improvement of a metric is defined as follows (also see [62].

Definition 2.6.1 (Improvement/Degradation of a Metric): Given two samples $X_1, \ldots, X_N$ and $Y_1, \ldots, Y_N$ and their means $\overline{X}$ and $\overline{Y}$, the *improvement* of X over Y (equivalently, the improvement of $\overline{X}$ over $\overline{Y}$) is defined as

$$\frac{\overline{Y} - \overline{X}}{\overline{Y}}.$$

A negative improvement of x is a *degradation* of $-x$. ◇

3. Related Work

This chapter presents related work on graph partitioning as well as sequential and parallel approximation of maximum weight matchings.

3.1. Graph Data Structures

As stated in Section 2, we assume that the vertices of a graph with n vertices are numbered 0 to $n-1$.

In many applications, static graphs are represented in computers using the *adjacency array* datastructure (e.g. [50]). It is also sometimes referred to as *compressed sparse row* (CSR), *forward star* ([22]) and *row-wise compacted adjacency matrix* ([25]) representation.

The adjacency array representation is as follows: We need an array *adj* with length $n+1$ of integers. Additionally, we have an array *heads* with lenght m of integers. *heads*[j] stores the identifier of the head of the jth edge. *adj*[u] gives the offset of the first outgoing edge of vertex u in *heads*. The ith neighbour of vertex u is stored in *heads*[*adj*[u]]. The out-degree of u is *adj*[$u+1$] $-$ *adj*[u]. If edge weights are required, they can be stored in an array *edge-weights* that is used the way *heads* is. Generally, we sort the outgoing edges for each vertex by the identifier of the target vertex so binary search is possible.

Figure 3.1 shows an example for an undirected graph and its adjacency array representation.

The adjacency array representation is static, i.e. vertices and edges cannot be added easily. However, updating edge weights is possible in constant time. The running time of the operations on adjacency arrays is the same as for adjacency lists. If we sort the outgoing edges for each vertex, queries for and updates of edge weights can be done in $\mathcal{O}(\log \deg(u))$ where u is the tail of the vertex. The main advantage of this representation is its cache efficiency. We only have to store one integer more than the absolutely minimal information of the graph. For arbitrary graphs there is hardly any more space efficient representation of graphs, short of compression.

3.2. Graph Partitioning

The graph partitioning problem has been the topic of a lot of research. We give a list of the general approaches, mainly following Fjällström's survey paper [19], leaving out spectral bisection methods. Section 3.2.1 is about exact solvers, the rest of this section is about heuristics. Section 3.2.2 describes local improvement methods. The subsequent Sections 3.2.3 and 3.2.5 describe global methods to create a partition of a graph "from scratch."

(a) An example for an undirected graph

(b) Adjacency array representation of this graph.
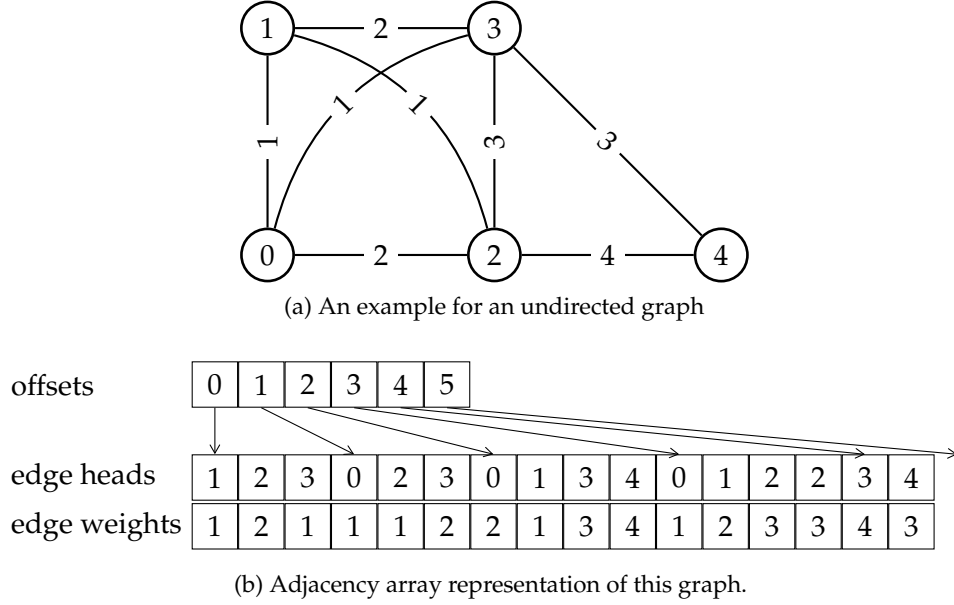
Figure 3.1: An undirected graph and its adjacency array representation.

3.2.1. Exact Solvers

Various super-polynomial algorithms are known to solve the graph partitioning problem. [35] describes an approach using semidefinite programming and also lists references to solvers using branch-and-bound sequentially or in parallel or column generation. According to [35], this is only practicable for general graphs of 60 vertices or less.

3.2.2. Local Improvement Methods

A local improvement methods takes a graph *G* and a partition *P* of *G* as its input. It then tries to improve the balance and/or edge cut of *P* using some sort of local search (e.g. [50]).

To be able to use a local improvement method, an initial partition *P* is required. Thus, local improvement methods must be combined with a global method yielding such a partition.

One of the earlierst proposed methods is due to Kernighan and Lin ([44]). Many other local improvement methods are based on the Kernighan-Lin (KL) heuristic Maybe the most prominent is a linear-time heuristic due to Fiduccia and Mattheyses (FM, see [18]). Subsequently, FM served as the base for multiple local improvement variants. See [19] for an account of them. [62] presents a parallel variant that improves the best known partitioning for many graphs in Walshaw's partitioning archive ([73]). This diploma thesis also has a detailed description of the FM heuristic.

Other approaches for local improvement include simulated annealing, tabu search

and genetic algorithms.

3.2.3. Geometric Partitioning

Geometric partitioning requires a coordinate to be associated with each vertex. Since graph partitioning is often applied to geometric graphs (such as finite element meshes), this is possible for a large family of problems.

The simplest variant is *recursive coordinate bisection* (RCB) [3]. It is very similar to building a so-called *k*-d-tree, proposed by Bentley [2]: Consider the case where the order *k* (unrelated to the *k* in *k*-d-tree) of the partitions is a power of 2. The set of points (vertices with coordinates) is bisected by a plane orthogonal to one of the axes, splitting the set into two halfs of equal size. This bisection is recursively continued until the desired number *k* of partition entries is reached. Usually, the coordinate axis is used for which the coordinates have the largest spread. This involves the computation of the median of the point sets in one coordinate which can be done in linear sequential time (see [8] for a deterministic and [50] for a randomized linear time algorithm).

Another approach ([17, 45]) improves on the simple RCB by selecting the axis to cut orthogonally to be the axis of *minimum angular momentum*. Yet other approaches select the coordinate axis and move the plane by some ε such that the edge cut is minimized ([54]) or are based on bisection using *d*-dimensional spheres ([5]).

3.2.4. Recursive Graph Bisection

The *recursive graph bisection* (RGB) method is a coordinate-free method ([63]). It begins by finding a *pseudo peripheral vertex* in the graph. This means finding a pair of vertices that have the approximately greatest distance from each other in the graph.

Then, breadt-first search is performed from the selected vertex to determine the distance from all other vertices to the selected one. Finally, the vertices are sorted accordingly to this vertex and the sorted set is divided into two equal-sized sets.

Recursive graph bisection can be combined with local improvement methods: After each bisection, the local improvement can be called on the newly created bisection.

3.2.5. Multi-Level Partitioning

The simpler variants of geometric bisection are very fast but do not yield very good partitionings (the more complex variants achieve better partitioning results). Recursive graph bisection yields good results, especially when combined with local improvement. However, RCB is very slow when combined with FM heuristics.

Hendrickson and Leland proposed a *multi-level* approach to graph partitioning in [27] to overcome this limitation of RCB. The idea is to create a sequence of shrinking graphs (in terms of vertex count) where each graph resembles the next larger one. This step is called *coarsening phase*. The smallest graph is then partitioned using RCB with FM in the *initial partitioning phase*. Finally, from a graph with a partitioning, a partitioning of the next larger graph is extrapolated in the *uncoarsening phase*. After each extrapolation,
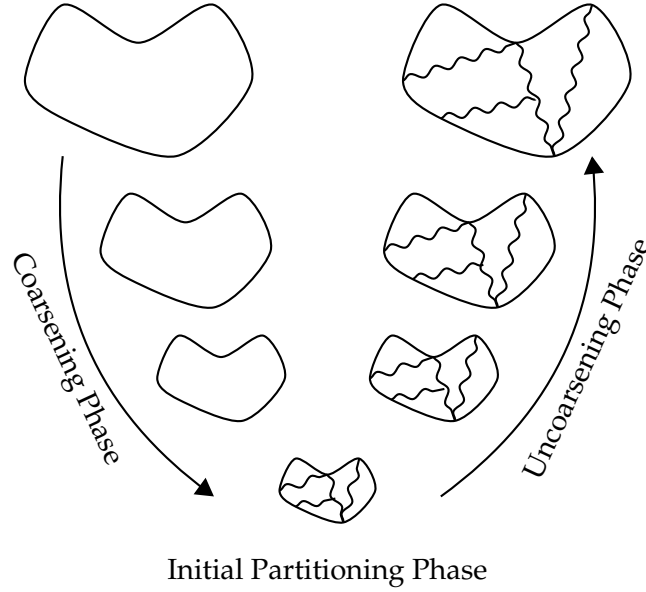
Figure 3.2: The idea behind the multi-level algorithm ([40]).

the FM heuristic is used to improve the partitioning. Figure 3.2 shows the idea behind the coarsening, partitioning and uncoarsening phase. See [19, 37] for further reading on multi-level graph partitioning.

In [37], Karypis and Kumar report that their multi-level algorithm produces good partitions and is faster than a multi-level recursive bisection algorithm by two orders of magnitude.

3.2.6. Available Software Packages

Since graph-partitioning (and the related problem of matrix reordering) is important for almost all parallel scientific computation programs, there are several software packages available.

None of these programs considers a different objective function for the maximum weight matching but the edge *weight*.

All software packages only use the HEAVY-EDGE-MATCHING algorithm for the computation of matchings in the coarsening phase.

In their parallelizations of HEM, PT-SCOTCH and JOSTLE use simple parallelizations of this algorithm: After the local matching computation, each process computes a candidate vertex to match with for all local edges and stores it in a buffer for each neighbouring process. The buffers are exchanged and if the same pair of vertices was matched in two processes, their connecting edge is accepted into their matching. In this case, they are not considered further. The process is then repeated until no vertex has any unmatched neighbours or aborted after at most 5 steps in the case of PT-SCOTCH.

PARMETIS computes a vertex coloring of the graph after matching the local edges

with HEM. Afterwards, for each color, the following is performed: Each process computes a matching candidate for each local vertex of the given color. The matching candidates are sent to the owning processes. In the case of a collision, only one request is answered positively.

No software that we are aware of uses ghost vertices in the way that the MANNE-BISSELING-MATCHING algorithm does. The aim there is to minimize collisions and the number of rounds to run.

Metis. METIS ([37, 36, 41, 42]) contains the family of graph partitioning algorithms that resulted from the research of Karypis et al. The programs are written in ANSI C, portable and very fast.

By our experience, the code base is small and sufficiently clean to instrument and modify. This is also shown by the fact that other research is based on the METIS program, e.g. [15]. However, METIS is known not to create the best possible partition [29]. Its comments could be more extensive, too.

The results of the research of Karypis et al. on parallel graph partitioning are available as PARMETIS ([38, 39, 40, 43]). PARMETIS uses MPI for the parallelization of distributed memory computers. Both METIS and PARMETIS can be compiled as libraries and used from other programs. METIS is one of the most widely known graph partitioning packages, possibly because it is freely downloadable, portable and fast.

We will see in Chapter 6 that the coarsening phase of PARMETIS does not scale as well as our code and yields worse results in terms of initial edge cut.

Scotch. SCOTCH ([58]) by Pellegrini et al. is a software package for static mapping of graphs to network topologies. It also contains a standalone graph partitioning program and provides a library interface that is compatible to the one of METIS. PTSCOTCH ([7]) is the parallel version of SCOTCH, both are available freely for download.

Jostle and NetWorks. JOSTLE is a software package by Walshaw et al. ([72, 71, 69, 70]) with sequential and parallel codes for graph partitioning.

Party. PARTY [59] is a software package by Preis and Diekmann for graph partitioning. A thread-parallel version of PARTY is available. The parallelization for distribution memory is difficult because of the helpful sets used in the partitioning algorithm ([53]) and there exists no such implementation.

DiBaP DIBAP [52] is a graph partitioning package based on diffusion. It yields the best partitioning results for some of the graphs in [73].

DIBAP contains parallel code for shared memory using threads and distributed memory using MPI. However, the existing MPI code does not scale well ([53]).

3.3. Matchings in Graphs

Both the Maximal Matching and the Maximum Weight Matching problems are in $\mathcal{P}$. The proofs can be found in [46] which gives a broad theoretic foundation on matchings. The fastest known algorithm to compute maximum weight matchings has an asymptotic running time of $\mathcal{O}\left(n\left(m + n \log n\right)\right)$ ([20]).

For large sparse graphs, faster approximations are desirable. [49] provides a recent experimental study of approximate matching algorithms, including the GLOBAL-PATHS-ALGORITHM (GPA) algorithm we use. This paper also has an account of related work on exact and approximate algorithms. The GREEDY algorithm for approximating maximum weight sets is well-known folklore.

In [6], Bouhmala presents two alternatives to the HEM matching algorithm used by Karypis in [38]: Gain Vertex Matching (GVM) and Closest Vertex Matching (CVM). They are specific to the coarsening phase of a multi-level graph partitioning algorithm.

Work on algorithms to compute matchings in parallel is less numerous and mostly of theoretical nature. In [33], Israeli and Shiloach present a deterministic algorithm for the computation of maximal matching running in time $\mathcal{O}(\log^3 m) = \mathcal{O}(\log^3 n)$ using $n + m$ processors. In [32], Israeli and Itai present a randomized parallel algorithm that runs in $\mathcal{O}(\log m)$ expected time using m processors. Both algorithms require a CRCW-PRAM.

There are three implementations of parallel matching algorithms we are aware of:

In [39], Karypis and Kumar propose a simple scheme for distributing the graph and then computing matchings only on local parts. In [38], they propose a simple parallelization of the HEM algorithm described in Section 5.2. The parallelization is based on finding colorings of graphs using Luby's parallel algorithms for the maximum independent set problem (see [47]).

In [48], Manne and Bisseling present a "direct" parallel algorithm for approximating maximal weight matchings that does not need to compute a coloring of a graph as a precomputation step. They build upon the work of Hoepman in [30] and show that Hoepmann's algorithm is a variant of Luby's classic parallel algorithm for maximum cardinality independent sets from [47]. Their algorithm is a Bulk Synchronous Parallel (BSP, [67]) style algorithm. The description of their parallel algorithm is not very precise and they do not present extensive practical results.

Recently, Pothen et al. built upon the work by Manne and Bisseling in [26]. Their algorithm shows good strong and weak scaling on three large machines and various real-world graphs. This algorithm works asynchronously at the cost of a higher complexity.

In [72], Walshaw describes the matching algorithm used in the parallel JOSTLE version. Each process selects the heaviest outgoing edge for each unmatched vertex to an unmatched neighbour. After the selection, the processes take the mutually selected edges (after a communication step) and match the adjacent vertices.

4. Distributed Coarsening

In Section 3.2.5, the general idea behind multi-level graph partitioning was explained. In this section, we will present the implementation of our coarsening phase.

Note that we do not claim that the following algorithms are new. They seem to be the "natural" way to perform graph coarsening and contraction efficiently. We implemented them after studying the source of KMETIS ([37]).

4.1. Coarsening Overview

As described in Section 3.2.5, the goal of the coarsening phase is responsible to create coarsened problem instances $\mathcal{I}^i$. Each $\mathcal{I}^i$ is to resemble $\mathcal{I}^{i-1}$.

Each problem instance is a weighted Graph $G = (V, E)$. We coarsen the graph by contracting sets of vertices on level i to vertices on level $i + 1$ using a function $f : V \to V'$ (also see Definition 2.5.3).

For each level of this hierarchy, we also store $f^{-1} : V' \to 2^V$, the mapping back from vertices on level i to sets of vertices on level $i - 1$. Using this mapping, we can uncoarsen the graph later.

At a high level, the coarsening algorithm is very simple. It is shown in Algorithm 1.

Algorithm 1 High-level description of the coarsening algorithm.

DISTRIBUTED-COARSENING(G)

1 *hierarchy* $\leftarrow \langle G \rangle$
2 **while not** TERMINATION-CONDITION(*hierarchy*)
3   **do** $\triangleright$ Invariant G is the coarsest graph in *hierarchy*
4          $M \leftarrow$ DISTRIBUTED-MATCHING(G)
5          $\left(G', f^{-1} \right) \leftarrow$ CONTRACT-DISTRIBUTED-MATCHING(G, M)
6   append $\left(f^{-1}, G' \right)$ to hierarchy.
7          $G \leftarrow G'$
8 **return** *hierarchy*

After each contraction, we add a new level to the hierarchy. Each level consists of a mapping to the previous level and the coarsened graph. Note that the top level does not need such a mapping f.

Algorithm 1 leaves several important functions undefined:

- TERMINATION-CONDITION returns TRUE if the coarsening algorithm should terminate and is described in Section 4.2.

- DISTRIBUTED-MATCHING approximates a maximum weight matching $M \subseteq E$ for $G = (V, E)$. Our parallel matching algorithm is described in Chapter 5.

- CONTRACT-DISTRIBUTED-MATCHING performs a graph by a matching M and returns the coarsened graph G' and the mapping from vertices in G' to vertices in G. The contraction algorithm is described in Section 4.4 and Section 4.5.

4.2. Limits for Coarsening

The aim of the coarsening phase is to reduce the size of the problem that we have to solve initially, in this case graph partitioning.

First, this means that the time spent in coarsening should pay off. Coarsening, solving the problem on the coarsened graph and uncoarsening should not take longer than solving the problem directly on the original graph. The main problem here is when the cardinality of the found matching drops too low and too few vertices are coarsened away in each level. We introduce the term of the *coarsening rate*, defined as $1 - n'/n$ where n' and n are the number of vertices in the coarser and original graph. If the rate drops below a certain value, coarsening stops (TERMINATION-CONDITION returns TRUE). The value in our experiments is 5%, the same as in METIS ([42]).

The coarsening rate can drop if a coarsened graph mainly consists of few stars. After the stars' centers have been matched, no other vertices can be matched.

Second, the coarsest graph should be "partitionable" well. For this, the coarsest graph should have a sufficient number of vertices left so the partition algorithm has a certain amount of freedom of choice. Additionally, the coarsest graph should not contain vertices that are too heavy because they also restrict the amount of freedom of the partition algorithm.

We try to achieve this by giving a lower bound on the number of vertices in the coarsest graph and bounding the maximal weight of a vertex that is created during coarsening. We also adapted the same bounds parallel coarsening for our parallel program.

In the sequential case, the lower bound on the number of vertices is $n_{\min}$, the maximal vertex weight is $w_{\max}$. In the sequential case, we give these bounds for each process, named $n_{\min,\,\text{local}}$ and $w_{\max,\,\text{local}}$. They are defined as:

$$n_{\min} := \max\left\{\frac{n}{40\log k}, 20k\right\}$$

$$w_{\max} := 1.5\frac{n}{n_{\min}}$$

$$n_{\min,\, local} := \max\left\{\frac{n}{40k\log k}, 20\right\}$$

$$w_{\max,\, local} := 1.5\frac{n_{local}}{n_{\min,\, local}}.$$

The values for $n_{\min}$ and $w_{\max}$ are taken from METIS ([42]). $n_{\min,\, local}$ and $w_{\max,\, local}$ are the natural adaptions to the parallel scenario.

Vertices that would have a weight greater than $w_{\max}$ (or $w_{\max,\, local}$ in the parallel case) are not considered as matching candidates. If the global number of vertices drops below $n_{\min}$ in the sequential case or the local number of vertices drops below $n_{\min,\, local}$, coarsening stops (TERMINATION-CONDITION returns TRUE).

4.3. Data Structures

4.3.1. Distributed Graph Data Structure

Our distributed graph data structure is based on the adjacency array graph data structure described in Section 3.1. The distributed graph has n vertices and m edges.

Each vertex has a globally unique identifier. The vertex identifiers are the contiguous sequence $\langle 0, \ldots, n-1\rangle$. Each process is responsible for a contiguous subsequence of these identifiers. The vertices are split at positions s_i $(i = 0, \ldots, k)$, the *vertex splitters*. The following always holds: $s_0 = 0$ and $s_k = n$. Process i is responsible for the contiguous subsequence $\langle s_i, \ldots, s_{i+1} - 1\rangle$, the *local vertices*.

The edges are managed in a similar way: Each process stores the edge heads for all local vertices. This means that process i stores the edges heads with the indices/identifiers $adj[s_i]$ to $adj[s_{i+1}] - 1$.

Each process uses an adjacency array representation for storing the local graph. The edge bucket offset array is index by *local vertex identifiers*. For a global vertex identifier v, the local identifier v_l on this process is given by $v_l = v - s_i$. Similarly, for a global edge identifier e, the local index/identifier e_l on this process is given by $e_l = e - adj[s_i]$.

Additionally, the distributed graph contains the properties (vertex weight, internal edge count for the vertices and edge weight) of the local part of the graph.

The vertex splitters $\langle s_0, \ldots, s_k\rangle$ and *edge splitters* $\langle adj[s_0], \ldots, adj[s_k]\rangle$ are stored in all processes.

4.3.2. Coarsening Hierarchy Data Structure

The data structure for the coarsening hierarchy is simple: A hierarchy is a list of hierarchy levels. Each level consists of a distributed graph part and the mapping f^{-1}.

f^{-1} is a mapping from a contiguous range of integers to static lists of integers. We use an adjacency array data structure to represent this nested list. Since we know the cardinality of the domain n and the cardinality of the set of values n' of f, we can compute f^{-1} in time in $\Theta(n)$.

4.4. The Sequential Contraction Algorithm

The sequential contraction algorithm SEQUENTIAL-CONTRACTION is shown in Algorithm 2. The matching is given as a *mate mapping*, a static array mapping local vertex identifiers to their global partner in the matching. The algorithm returns the contracted graph G' with the mapping f from vertex id in G' to the vertex id in G. f is given as an integer array. Section 2.5 contains a more formal description of the term *contraction*.

The general idea behind the algorithm is similar to the external memory algorithm for constructing minimal spanning trees from [13].

First, the algorithm builds f from *mate-map* and computes n' in lines 1 to 9. Second, it traverses all edges in G and renames the edges according to f in lines 10 to 12. The new edges are collected together with their original edge id. The original edge id is required for retrieving the edge weight in the original graph. Third, it sorts the array of edges in line 13. Fourth, it traverses the edge array in lines 14 to 28. If two edges (u, v) and (x, y) are adjacent in the sorted array with $u = x$ and $v = y$ then G' only contains one edge (u, v) whose weight is the sum of the weights of both original edges. The number of edges in G' yields m'. Finally, it combines the information into the graph G' in line 29.

The running time is in $\mathcal{O}(m \log m)$ if comparison-based sorting is used. With integer sorting, the running time is linear in m.

We can create another variant of this algorithm using comparison based sorting that runs in $\mathcal{O}(m \log d)$ where d is the maximum degree of G: We preallocate an array of n buckets, each with a maximal size of $2d$. When renaming the edges, we put the edge (u, v) directly into the bucket for u. Afterwards, sorting only has to be performed for each bucket.

4.5. The Parallel Contraction Algorithm

The parallel contraction algorithm works similar to the sequential one. However, it is much more complex because vertices can be migrated between processes if edges between processes are matched.

The main challenge is in exchanging the mapping f from vertex ids in G to vertex ids in G'. The easiest solution would be for each process to broadcast its part of the mapping to all other processes. However, this would effectively require $\Theta(n)$ storage on each process and prevent scaling.

Thus, each process should only have to store a small part of f: Each process must know the value of f for all local vertices u of G, also for the vertices migrated to it. Additionally, each process must know the value of f for all vertices v that an edge (u, v) exists of where u is a vertex of G that is local to the process after migration.

Algorithm 2 The sequential contraction algorithm.

SEQUENTIAL-CONTRACTION($G = (V, E), \textit{mate-map}$)

$\quad \triangleright$ Build f and compute n'.
1 $f \leftarrow$ array of size n
2 $i \leftarrow 0$ $\qquad\qquad\qquad\qquad\triangleright$ identifier of next contracted vertex
3 **for** $j \leftarrow 0$ **to** $n - 1$
4 $\quad$ **do if** $\textit{mate-map}[j] \geq j$
5 $\qquad$ **then** $f[j] \leftarrow i$
6 $\qquad\qquad$ $i \leftarrow i + 1$
7 $\qquad$ **else** $\triangleright$ $f[j] < j$
$\qquad\qquad\qquad\triangleright$ Invariant: $\textit{mate-map}[j] < j$
8 $\qquad\qquad\qquad$ $f[j] \leftarrow f[\textit{mate-map}[j]]$
9 $n' \leftarrow i$
$\quad \triangleright$ Build $X = \langle(s, t, \textit{orig-id})\rangle$ and compute m'.
10 $X \leftarrow \langle\rangle$
11 **for** $(u, v) \in E$, edge id is e
12 $\quad$ **do** append $(f(u), f(v), e)$ to X
13 SORT(X)
14 $\textit{offsets} \leftarrow \langle 0 \rangle$
15 $\textit{edge-heads} \leftarrow \langle\rangle$
16 $j \leftarrow 0$
17 $(x, y, d) \leftarrow X[0]$
18 $w'(j) \leftarrow w(d)$
19 **for** $i \leftarrow 1$ **to** $|X| - 1$
20 $\quad$ **do** $(u, v, e) \leftarrow X[i]$
21 $\qquad$ **if** $x = u \wedge y = v$
22 $\qquad\quad$ **then** $w'(j) \leftarrow w'(j) + w(e)$
23 $\qquad\quad$ **else** $j \leftarrow j + 1$
24 $\qquad\qquad\quad$ append j to $\textit{edge-heads}$
25 $\qquad\qquad\quad$ $w'(j) \leftarrow w(e)$
26 $\qquad\qquad\quad$ $(x, y) \leftarrow (u, v)$
27 append j to $\textit{edge-heads}$
28 $m' \leftarrow j$
29 $G' \leftarrow$ new graph from n', m', $\textit{offsets}$, $\textit{edge-heads}$ and w'
30 **return** (G', f)

The interface of the algorithm is similar to the sequential variant. It takes a distributed graph as the input and an array *mate-map* describing the mapping. If s_i is the vertex offset of process i and v is a vertex on process i, *mate-map*$[v - s_i]$ gives the global id of the mate in the mapping. As a shortcut, we denote *mate-map*$[v - s_i]$ as *mate-map*$[v]$, implicitly subtracting s_i.

We will describe the contraction algorithm from a high-level perspective. A detailed account would not be very instructive and it complicates the description. Figure 4.1 (p. 33 ff.) shows an example of the distributed coarsening algorithm.

Compute n' and n'_{local}. First, the algorithm computes n'_{local}. This is done by iterating over the local vertices u in G. For each u, we check whether $v := $ *mate-map*$[u]$ is a local vertex.

If this is the case, the vertex that is larger by its numeric id is contracted into the smaller one. We only count u if $u \leq v$.

If v is not local then the process with the smaller process id gets ownership of the resulting vertex in the coarser graph if $u + v \equiv 0 \bmod 2$. Otherwise, the other process gets the ownership. The aim of this scheme is to distribute the coarser vertices evenly between the processes.

Then, the processes exchange their values of n'_{local} with all other processes and compute n' and the vertex splitters of the coarser graph from this information.

Figure 4.1a shows the initial distribution. The vertex migration is indicated as gray arrows. For each process P_i, the value for n'_{local} is shown.

Migrate vertices to new owners. Second, the vertices that changed their owner are migrated. Vertex can change their owners when they are matched with a vertex on another process and fulfill the condition above. The new owners can be determined by iterating over all local vertices and a binary search on the vertex splitters of G. For each process, the information about the vertices and their outgoing edges that are to be migrated to them is stored in a buffer.

After the buffers are filled, they are exchanged using an ALL-TO-ALL operation. For simplicity, processes send vertices that are not to be migrated to themselves.

Build f and f^{-1} for local vertices in G'. Third, the mappings f and f^{-1} are built for the vertices the current process owns in G'. f is implemented as a hash table, f^{-1} is implemented as a CSR structure. We can preallocate the memory for f^{-1} since we know n'_{local} and can compute the number of vertices from G residing on this process after migration from the received messages. Building the mappings is done by iterating over all received vertices (with outgoing edge information) in the order of their sender's process id where the sender's process id is smaller than the current process' ones.

For each received message, we iterate over the outgoing edges around vertices u. Starting from the vertex offset in G', we assign new vertex ids to them, defining f on u. In f^{-1}, we fill out the first entry for the new id of u. Then, we iterate over all local vertices u and consider $v := $ *mate-map*$[u]$. Depending on whether u was matched with

another vertex at all, $u < v$ or $u > v$ and whether v is a local vertex, we iteratively build f and f^{-1}.

Figure 4.1b shows the graph after distribution. It also shows the vertices v_i' of the contracted graph and the local definitions of f and f^{-1} as far as the processes know them at this point.

Distribute f to all involved processes. As stated above, each process must know the id of the vertex in G' that contains the vertex u for all local vertices u in G. Additionally, each process that has received an edge (u, v) in a list of outgoing edges must have its local part of f defined for v. It already knows the value of f for u.

In the first step, each process sends the pair $(u, f(u))$ back to the original owner of u for all vertices u that were migrated to it (Figure 4.1c). The processes receive these pairs and can define f at the vertices u (Figure 4.1d). At this point, each process knows the value of f for all local vertices they owned before and own after migration. In the second step, they forward the pairs $(u, f(u))$ to the original owner of v for all local edges (u, v) in their local part of G (Figure 4.1e, result shown in Figure 4.1f). Then, these processes forward information they received in the second step to the new owners of v in a third communication step. At this point, each process knows the value of f for u and v for all edges (u, v) in the list of outgoing edges it received.

In our example, the third communication step is not necessary: Forwarding would mean that the involved processes would send messages to themselves which are suppressed since they are unnecessary. We refrained from giving an example where the third communication step is necessary since this would have made the example too complex.

Build the final graph data structure. Finally, the coarser graph G' is built from the information created so far and returned along with f and f^{-1}.

4.6. The Parallel Uncoarsening Algorithm

The parallel uncoarsening algorithm is responsible for creating a sequence of finer graph from the graphs in the coarsening hierarchy. Each graph is "finer" than the one before. Each graph resulting from the extraction gets refined by the refinement algorithm from [62].

Initialization. The uncoarsening algorithm takes the coarsening hierarchy created by the coarsening algorithm an the initial partition of the coarsest graph as its input. From this, it creates a list of the ids of the vertices that are local to this process. From here on, the current partition is given by the distribution of the vertex ids on the processes.

Uncoarsening. For each graph in the hierarchy, beginning with the coarsest, the algorithm first refines the partition using the refinement algorithm from [62]. Then, it maps

the vertices of the current level to their counterpart in the finer level by applying f^{-1}.

Input for the Refinement Algorithm.   The refinement algorithm requires the neighbouring processes of the current one as its input. Two processes are neighbours if there is a vertex owned by the first that is adjacent to a vertex owned by the second. The algorithm also needs and the vertex information (neighbours, weight of outgoind edges, vertex weight, internal edge count) for all vertices that are local to the process at this point.
   Each process first gathers the vertex information for all vertices that are local to it before refinement. Using this information, it then computes the neighbouring processes. Afterwards, the refinement algorithm is called.

Application of f^{-1}.   Afterwards, each process has a list of vertex ids in the current level that it owns after the refinement step. Now, each process assembles f^{-1} for each local vertex u. This information can already exist in the same process if the vertex u already existed in the process when coarsening. Otherwise, it requests the value of $f^{-1}(u)$ from the original owner of u (which can be determined by the vertex splitters).
   Finally, it applies f^{-1} on each local vertex and gets a list of vertices that are owned by the current process after the de-contraction.
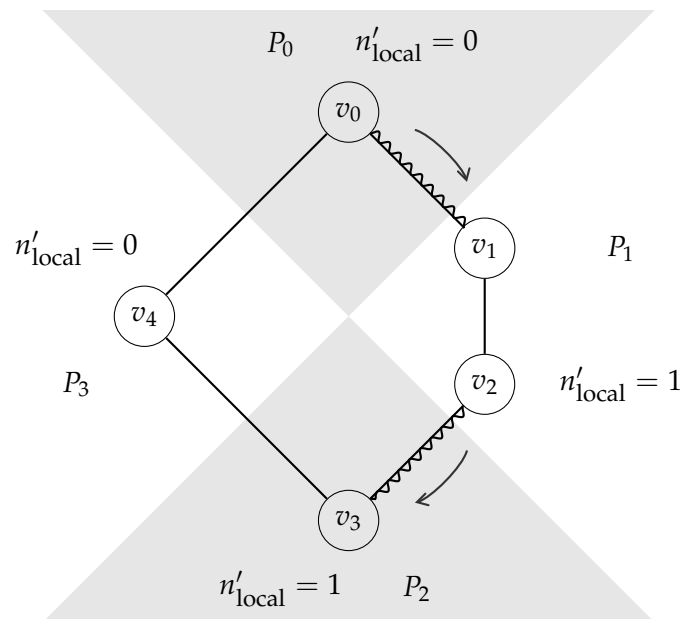
(a) Initial distribution with matchings of graph G to four processes P_i. The gray arrows indicate vertex migration.
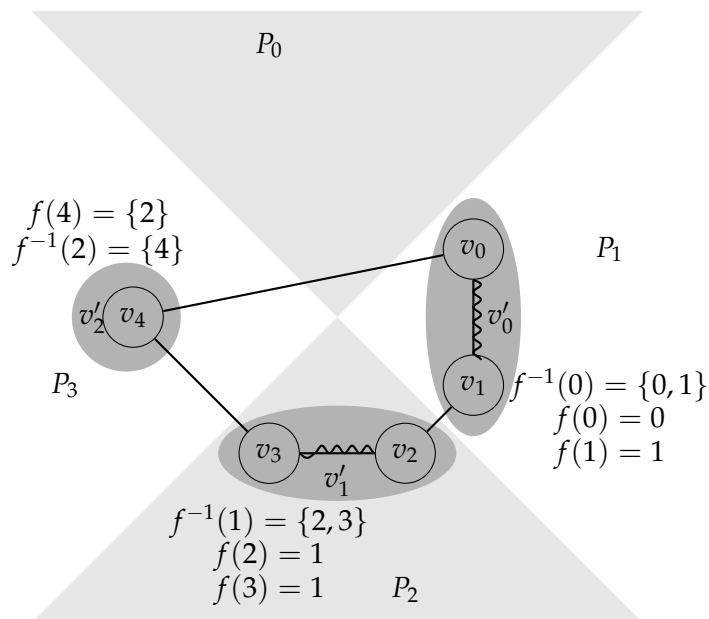
(b) The graph G after migration. The graph G' is shown in dark gray. The diagram also shows the mapping information known to each process.

Figure 4.1: Example for the distributed coarsening algorithm, part one of three.

(c) First round of communicating vertex id mappings.

(d) Result of the first vertex id mapping communication round.
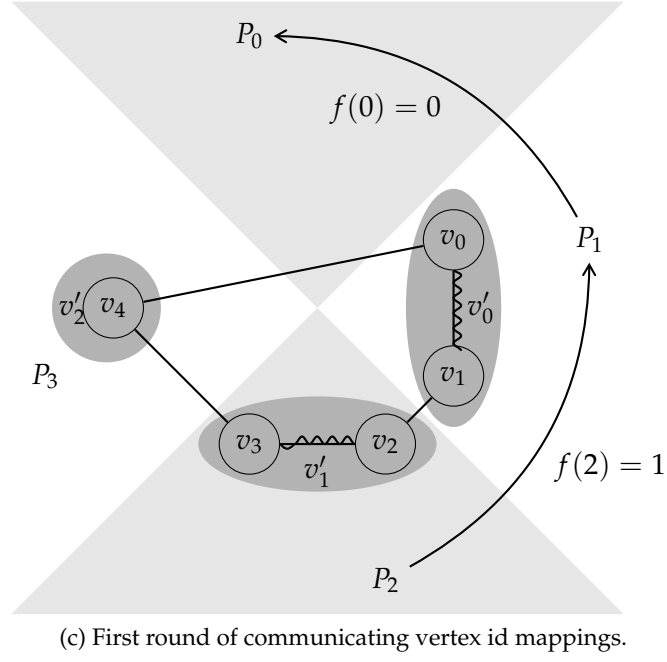
Figure 4.1: Example for the distributed coarsening algorithm, part two of three.

(e) Second round of communicating vertex id mappings.

(f) Result of the second vertex id mapping communication round.

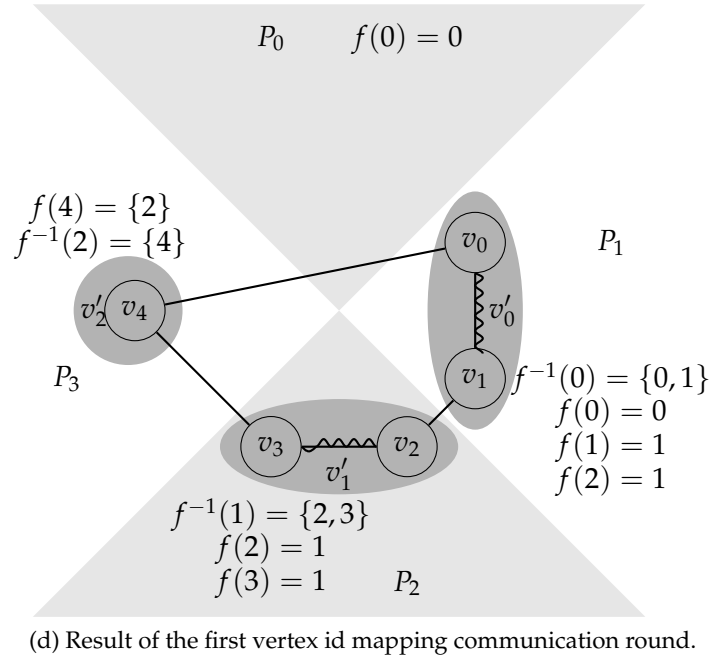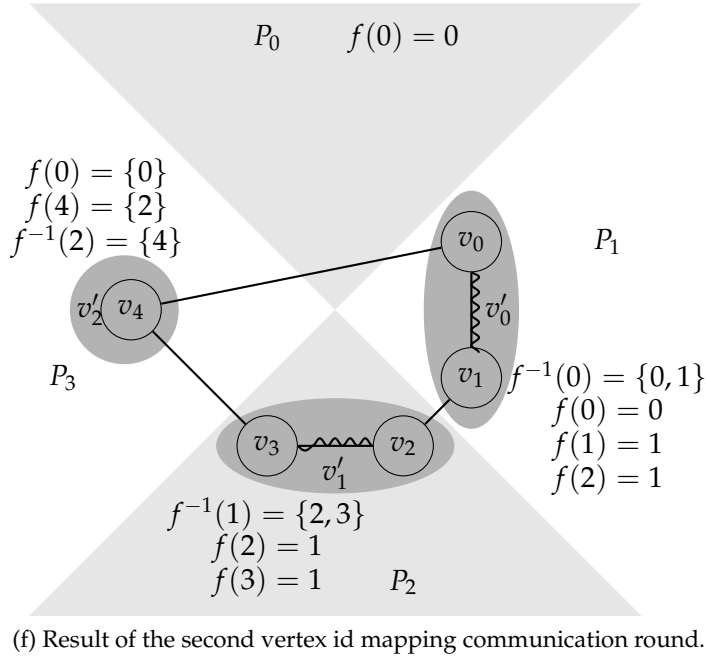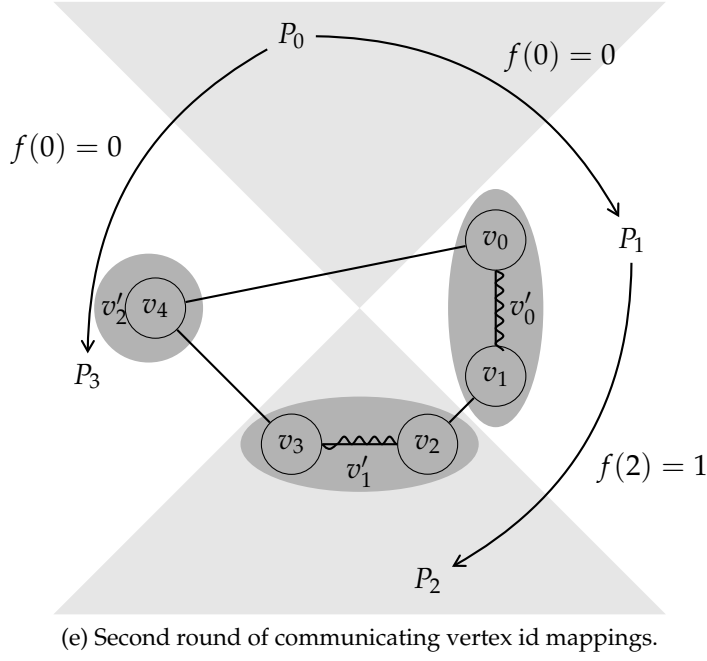Figure 4.1: Example for the distributed coarsening algorithm, part three of three. Note that there is no need for the third communication step in this example. It would consist of the processes sending messages to themselves.

4. Distributed Coarsening

5. Matching Algorithms

In this section, we show three sequential and one parallel approximation algorithm for the Maximum Weight Matching Problem defined in Section 2.3.

As stated in Section 3.3, the fastest known algorithm to compute maximum weight matchings is due to Gabow ([20]) and has a running time of $\mathcal{O}\left(n\left(m + n \log n\right)\right)$. In this thesis, we are mainly considering sparse graphs. For sparse graphs, this algorithm runs in $\mathcal{O}\left(n^2\right)$. This is too slow for large graphs.

Thus, we are interested in fast approximation algorithms. Sections 5.1, 5.2 and 5.3 describe three sequential algorithms and Section 5.4 describes parallel algorithms to approximate maximum weight matchings.

5.1. A Sequential Greedy Algorithm

Applying the greedy method ([8]) to the Maximum Weight Matching problem directly yields the well-known $1/2$ approximation algorithm in Algorithm 3.

Algorithm 3 Greedy algorithm to find an approximate MAXIMUM WEIGHT MATCHING with an approximation factor of $1/2$.

MWM-GREEDY$(G = (V, E), w)$
1   $M \leftarrow \varnothing$ ▷ matched edges
2   $S \leftarrow \varnothing$ ▷ matched vertices
3   **for** all edges $e = \{u, v\}$ in ascending order of their weight
4   **do if** $u \notin S \wedge v \notin S$
5   **then** $S \leftarrow S \cup \{u, v\}$
6                   $M \leftarrow M \cup e$
7   **return** M

It is easy to see that the algorithm can be implemented to have running time $\mathcal{O}(m \log m)$. If linear running time integer sorting can be used for the edge weights, it can even be implemented in $\mathcal{O}(m)$. The proof of the approximation factor $1/2$ is equally easy to see and we will give a sketch of it here:

Algorithm 3 is functionally equivalent to the following: Consider each edge $e = \{u, v\}$ descendingly ordered by weight. Add e to the matching and remove all edges adjacent to the vertices u and v from E. Let M be the resulting matching.

Let M' be a maximum weight matching. Let e be an edge that is in M but not in M'. Now, M' has to contain an edge adjacent to u and an edge adjacent to v:

If M' contained no edge adjacent to u and no edge adjacent to v, we could add e yielding a larger matching. If M' contained either an edge f adjacent to u or an edge f adjacent to v then we could remove f from the matching and add e, yielding a larger matching.

Let the two edges adjacent to u and v be f_1 and f_2. The weight of e is greater than or equal to the weight of both f_1 and f_2. Thus, $w(f_1) + w(f_2) \leq 2 \cdot w(e)$.

By induction, it follows that the sum of the weights of the edges in M is at least half the sum of the weights of the edges in M'.

5.2. A Sequential Local Greedy Algorithm

In [37], Karypis and Kumar propose the algorithm HEAVY-EDGE-MATCHING (HEM) shown in Algorithm 4 for generating approximate maximum weight matchings. The algorithm has a running time of $\mathcal{O}(m)$.

In their graph partitioner software METIS, they use a variant of HEM called SORTED-HEAVY-EDGE-MATCHING (SHEM). Instead of visiting the vertices randomly, SHEM visits them in ascending order of their degrees, breaking ties randomly. This is done to decrease the probability of creating high-degree vertices.

Algorithm 4 The "local" greedy algorithm HEAVY-EDGE-MATCHING.

HEAVY-EDGE-MATCHING($G = (V, E), w$)

```
1   M ← ∅                    ▷ matched edges
2   S ← ∅                    ▷ matched vertices
3   for all vertices v ∈ V in random order
4       do if v ∉ S
5           then for all outgoing edges {v, u}
6               do if u ∉ S
7                   then M ← M ∪ e
8                       S ← S ∪ {v, u}
9   return M
```

Note that no approximation guarantee can be given for HEM or SHEM. Figure 5.1 (p. 39) is a counterexample for any such factor: The value of ε is a very small, non-negative number. In order to get a maximum weight matching, either vertex v_1 or vertex v_2 would have to be considered first. However, we can add an arbitrary number of vertices v_i only connected to v_1 by an edge with weight ε. We can choose arbitrarily large values for n and arbitrarily small values for ε and make the worst-case and expected randomized worst-case approximation ratio arbitrarily low.
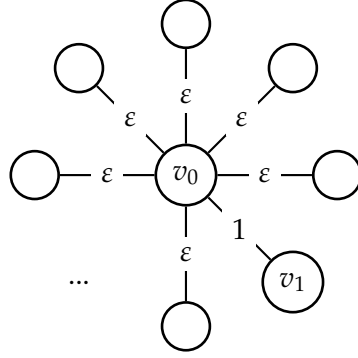
Figure 5.1: A counter-example for any approximation factor of algorithms HEM and SHEM.

In practice, the SHEM and HEM heuristics find good matchings, though.

The main advantage of the HEM and SHEM heuristics is that they are very fast. The implementation in METIS even tries to improve cache locality when traversing the vertices in random order: Instead of truly shuffling the order of vertices, only blocks of length four are shuffled. Another reason for this measure might be to generate fewer random numbers and cause fewer branch misses.

In [56] the graph *wave* ($n = 156\,317, m = 1\,059\,331$) is partitioned, using the HEM and the MWM-GREEDY algorithm. The program spends only half the time with finding matchings when using HEM compared with MWM-GREEDY on a 200 MHz Ultra-SPARC 2.

In [37], Karypis et al. also describe the RANDOM-EDGE-MATCHING algorithm that considers the vertices at random and selects a random edge to an unmatched neighbour.

5.3. The Global Paths Algorithm

The description in this section mainly follows [49].

In [49], Sanders and Maue propose the Global Paths Algorithm (GPA). GPA is a combination of the greedy algorithm described in Section 5.1 and the PGA' algorithm from [16]. It has been shown to find high quality matchings on real-world graph in low execution time. GPA is shown in Algorithm 5.

GPA works by generating an approximate maximum weight set of paths and even length cycles in lines 3-5: At the beginning of the algorithm, each vertex forms a path of length zero. The algorithm then iteratively adds edges in descending order of their weight if the edges are *applicable*. An edge is applicable if and only if it connects two endpoints of different paths or the two endpoints of an odd length path. Then, it computes maximum weight matchings of these paths and even length cycles in lines 6-8 using the dynamic programming routine in Algorithm 6.

The test for an edge being applicable can be done in constant time, as can be adding

---

Algorithm 5 The Global Paths Algorithm from [49]. The helper routine MAX-WEIGHT-MATCHING is shown in Algorithm 6.

$\text{GPA}(G = (V, E), w)$

1   $M \leftarrow \emptyset$
2   $E' \leftarrow \emptyset$
3   **for** each $e \in E$ in descending order of their weight
4   **do if** e is applicable
5   **then** add e to E'
6   **for** each path or cycle P in E'
7   **do** $M' \leftarrow$ MAX-WEIGHT-MATCHING(P)
8         $M \leftarrow M \cup M'$
9   **return** M

Algorithm 6 The helper routine MAX-WEIGHT-MATCHING for Algorithm 5

MAX-WEIGHT-MATCHING$(P = \langle e_1, \ldots, e_k \rangle)$

1   $W[0] \leftarrow 0$
2   $W[1] \leftarrow w(e_1)$
3   $M[0] \leftarrow \emptyset$
4   $M[1] \leftarrow \{e_1\}$
5   **for** $i \leftarrow 2$ **to** k
6   **do if** $w(e_i) + W[i-2] > W[i-1]$
7   **then** $W[i] \leftarrow W(e_i) + W[i-2]$
8             $M[i] \leftarrow M[i-2] \cup \{e_i\}$
9   **else** $W[i] \leftarrow W[i-1]$
10            $M[i] \leftarrow M[i-1]$
11   **return** $M[k]$

---

an edge to a path and the query for a path's length. Growing the paths in lines 3-5 can thus be done in linear time in the number of edges. Calculating maximum weight matchings on paths and cycles can be done in linear time, too. Thus, the total running time is $\mathcal{O}\left(\text{sort}(m) \cdot m\right)$ which is $\mathcal{O}(m \log m)$ in general.

5.4. Combining Sequential and Parallel Matching Algorithms

One dimension of the design space when designing a parallel matching algorithm is the locality of decisions in choosing edges for the matching. Using only local information as the basis for decisions lowers the amount of communication. However, good approximate sequential matching algorithms usually consider larger portions of the graph. Using more data for decisions requires more communication and might limit the achievable speedup.

The idea behind our parallel matching algorithm is to combine sequential and parallel matching algorithms. The graph is predistributed using the simple recursive coordinate bisection approach. For geometric graphs, we can expect a large portion of the edges to be local to one process only.

First, each process runs a sequential matching algorithm on its local edges. Then, the *gap graph* (see Definition 5.4.1) is considered. This graph contains the edges that are adjacent to vertices on two different processes. A parallel matching algorithm is run on the gap graph. The results of the local and the parallel algorithm are combined, yielding the final matching.

This is illustrated in Figure 5.2. This example also shows that in some cases, our algorithm does not yield maximum cardinality and thus no maximum weight matching. This could be fixed by further iterations of a local matching algorithm, followed by an augmentation of the matching in the gap graph. However, if a good initial distribution of vertices is chosen, we estimate that the improvement is neglegible.

Definition 5.4.1 (Gap Graph): Let G be a distributed graph as described in Section 4.3.1. Let M be the local matching, i.e. containing matched local edges only. Then, the gap graph $G^\star = (V^\star, E^\star)$ is given by the edges that connect vertices on different processors filtered by the schema:

Let $e = \{u, v\}$ be an edge in the gap graph with weight w_e. Let w_u and w_v be the weights of the heaviest outgoing local edges of u and v. e is in the gap graph if $w_e \geq w_u + w_v$. ◇

Let both the local and the parallel approximate matching algorithm yield ¹⁄₂ approximations for their inputs. Then the combination of their results also is an ¹⁄₂ approximation. The combination of the two edge sets is the union of the two edge sets without edges from the local algorithm's result that are adjacent to an edge from the paralell algorithm's result. This statement can be proven analogous to the proof that the GREEDY algorithm is an ¹⁄₂ approximation.
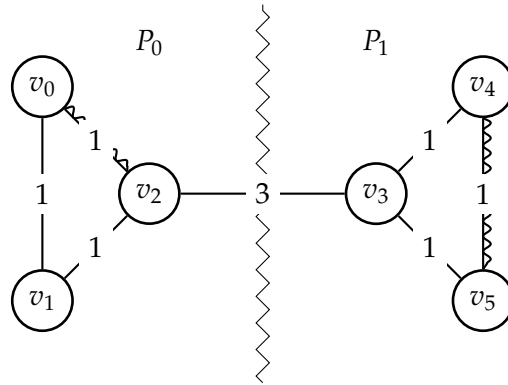
(a) First, compute a local approximate maximum weight matching
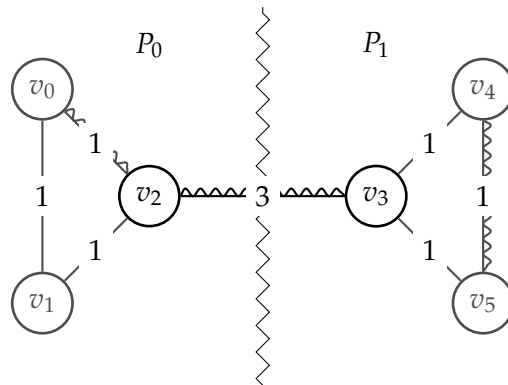
(b) Second, compute a matching in the gap graph. The gap graph is shown in black, the rest of the graph is shown in gray.
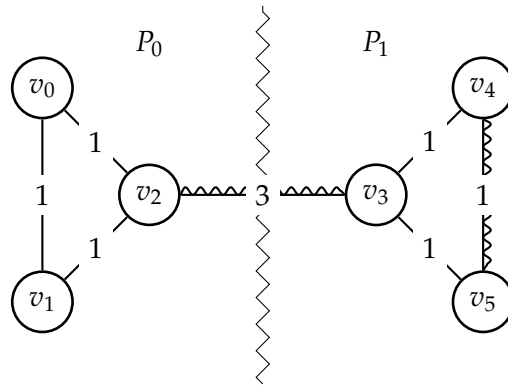
(c) Finally combine the matchings. Matched edges in the gap graph override matched local edges.

Figure 5.2: Illustration of the idea of our hybrid algorithm for parallel matching with predistributed vertices.

5.5. Manne-Bisseling Matching

The following synchronous parallel algorithm for approximating solutions to the Maximum Weight Matching Problem is due to Manne and Bisseling [48]. In their paper, they give a sequential formulation and then sketch a parallel formulation. Here, we will give the basic idea behind the algorithm and then give a detailed description of the algorithm in pseudo code.

The basic idea behind the algorithm is very simple to describe for the PRAM. Each processor handles one vertex. The following is repeated until all vertices are matched or have no unmatched neighbours. Each processor sets the candidate neighbour $c(v)$ for its vertex v to the unmatched neighbour that is connected over the heaviest edge. Ties are broken by neighbour vertex numbers. Then, each processor checks whether the candidate $c(c(v))$ of $c(v)$ is equal to v. In this case, the two vertices are matched and removed from the graph.

Hoepmann proposed this algorithm for the distributed model in [30]. Algorithm 7 shows a possible adaption to the parallel model where each processor handles n/p vertices. It is based on the sketch in [48]. We hope that our more extensive formulation clarifies this sketch.

By itself, the algorithm is similar to the ones used in the existing software packages described in Section 3.2.6. However, in order to minimize unnecessary communication, the following is done:

Each processor keeps the neighbours of its vertices as ghost vertices in its own memory. In the following, the vertices known to a process, i.e. the local vertices and the ghost vertices it knows about, are called *local in the gap graph*.

The overall structure of the algorithm is shown in MANNE-BISSELING-MATCHING of Algorithm 7.

First, the algorithm performs some initialization done in INITIALIZE of Algorithm 7. Then, it runs in rounds until there are no more candidates for the next round to be considered.

Initialization. INITIALIZE of Algorithm 7 shows the initialization. *candidates* stores a set of vertices that are to be considered in the current round and *next-candiate* stores the candidates for the next round. *matched-edges* and *matched-vertices* are sets of the matched edges and vertices. *mate-candidate* gives the mate candidate for all local vertices. *candidate-neighbours* gives the neighbours that are candidates for all local vertices, sorted ascendingly by edge weight.

In lines 7 and 8, the mate candidates are set to NIL for all local vertices.

Then, in lines 9-13, *candidate-neighbours* is initialized. For each vertex v, its mate candidate is set to the neighbour that is adjacent through the heaviest edge.

Matching Rounds. The matching rounds (RUN-MATCHING-ROUND in Algorithm 7) work the following way: First, each process sends a pair $(v, mate\text{-}candidate[v])$ to the

owner of *mate-candidate*[v] for each of its *candidates* set (SEND-MATCHING-REQUEST in Algorithm 8).

Second, it processes all received pairs in HANDLE-MATCHING-REQUESTS of Algorithm 8). If it received a pair (u, v) and (v is owned by this process) and the mate candidate of v is set to u then this process can match v with u. If this process received a pair (u, v) but has not sent (v, u) in this round then it must have sent (v, u) in a previous round. However, the owner of u could not match u with v up to here since it has not received (v, u). Thus, the process sends (u, v) to the owner of u. In the next step, the owner of u can match u and v.

Third, every process notifies the neighbours of all vertices it has matched in the function NOTIFY-MATCHED-NEIGHBOURS of Algorithm 9. These neighbours can then later remove the matched vertices from their *candidate-neighbours* entries' lists.

Finally, each process performs some book-keeping for all unmatched vertices u in UPDATE-BOOK-KEEPING: The vertices matched in this round are removed from the candidate neighbours of u. If no neighbour is left, the candidate is set to NIL and the next unmatched vertex is processed. Otherwise, it is set to the candidate neighbour adjacent through the heaviest outgoing edge. If the candidate neighbour did not change then the next unmatched vertex is processed. Otherwise, if u is a ghost vertex and the mate candidate of u is not a ghost vertex then the mate candidate of u is added to the candidates for the next round. If u is no ghost vertex or u is a ghost vertex and *mate-candidate*[u] is no ghost vertex then u is added to the set of vertices to consider in the next round.

5.6. Edge Rating Variants

This section lists the *edge rating* variants we consider. The edge rating replaces the role of the edge weight in our matching algorithms. The aim of rating the edges possibly differently from the weights is to improve the "partitionability" of resulting coarsened graphs over simply using the edge weights.

In the following, *internal-edges*(u) gives the sum of the weight of the edges that were contracted into a vertex u.

We use the term *cluster* for a vertex u that results from lots of contraction of vertices. Its internal vertices are all vertices that u represents in the finest graphs.

Definition 5.6.1 (Weight): The definition of the *weight* of an ege is the trivial one:

$$\text{weight}(e = \{u, v\}) := w(e).$$

The edge weight is the "classic" rating originally introduced by Hendrickson and Leland in [27] and also used by Karypis and Kumar in [37]. ◇

Definition 5.6.2 (Expansion): The *expansion rating* of an edge is defined as

$$\text{expansion}(e = \{u, v\}) := \frac{w(e)}{w(u) + w(v)}.$$

Algorithm 7 Our parallel maximum weight matching approximation algorithm. The helper functions are shown in Algorithms 8 and 9.

MANNE-BISSELING-MATCHING($G = (V, E), w$)

1   INITIALIZE()
2   *global-count* ← −1
3   **while** *global-count* ≠ 0
4   **do** *candidates* ← *next-candidates*
5   RUN-MATCHING-ROUND()
6   *local-count* ← | *next-candidates* |
7   *global-count* ← ALL-REDUCE(*local-count*, +)
8   **return** *matched-edges*

INITIALIZE()

 1   *candidates* ← ∅
 2   *next-candidates* ← ∅
 3   *matched-edges* ← ∅
 4   *matched-vertices* ← ∅
 5   *mate-candidate* ← empty mapping from vertex to vertex
 6   *candidate-neighbours* ← empty mapping from vertex to vertex sequence
 7   **for** all vertices *v* in the local gap graph
 8   **do** *mate-candidate*[*v*] ← NIL
 9   **for** all vertices *v* in the local gap graph
10   **do** *candidate-neighbours*[*v*] ← ⟨*u* for *u* in $N(v)$ ascendingly sorted by weight⟩
11   **if** | *candidate-neighbours*[*v*]| > 0
12   **then** *mate-candidate*[*v*] ← LAST(*candidate-neighbours*[*v*])
13   *next-candidates* ← *next-candidates* ∪ {*v*}

RUN-MATCHING-ROUND()

1   SEND-MATCHING-REQUESTS()
2   HANDLE-MATCHING-REQUESTS()
3   NOTIFY-MATCHED-NEIGHBOURS()
4   UPDATE-BOOK-KEEPING()

Algorithm 8 The SEND-MATCHING-REQUESTS and HANDLE-MATCHING-REQUESTS helper functions for Algorithm 7.

SEND-MATCHING-REQUESTS()

1 *messages* ← array of p messages
2 *sent-pairs* ← $\varnothing$
3 **for** all v in *candidates*
4   **do** $u \leftarrow$ *mate-candidate*$[v]$
5   *pid* ← rank owner of u
6   **if** *pid* $\neq$ *rank*
7   **then** append (v, u) to *messages*$[pid]$
8   *sent-pairs* ← *sent-pairs* $\cup \{(v, u)\}$
9 ALL-TO-ALL(*messages*)

HANDLE-MATCHING-REQUESTS()

1 *messages2* ← array of p messages
2 **for** all received (u, v) in *messages*
3   **do if** *mate-candidate*$[v] = u$
4   **then** *matched-edges* ← *matched-edges* $\cup \{(v, u)\}$
5   *matched-vertices* ← *matched-verties* $\cup \{u, v\}$
6   **if** $(v, u) \notin$ *sent-pairs*
7   **then** ▷ We have not sent (v, u) to the owner of u.
8   *pid* ← rank of owner of u
9   add (u, v) to *messages2*$[pid]$
10 ALL-TO-ALL(*messages2*)
11 **for** all received (u, v) in *messages2*
12   **do if** *mate-candidate*$[v] = u$
13   **then** *matched-edges* ← *matched-edges* $\cup \{(v, u)\}$
14   *matched-vertices* ← *matched-vertices* $\cup \{u, v\}$

Algorithm 9 The NOTIFY-MATCHED-NEIGHBOURS and UPDATE-BOOK-KEEPING helper functions for Algorithm 7.

NOTIFY-MATCHED-NEIGHBOURS()

1 *messages3* ← array of *p* messages
2 **for** all vertices *v* matched in this round
3   **do for** all neighbours *u* of *v*
4   **do** append *v* to the message for the owner of *u*
5 ALL-TO-ALL(*messages3*)
6 *matched-vertices* ← *matched-vertices* ∪ {*v* for *v* in *messages3*}

UPDATE-BOOK-KEEPING()

1   **for** all vertices *u* in the local gap graph
2   **do if** *u* is matched
3   **then continue**
4   *candidate-neighbours*[*u*] ← *candidate-neighbours*[*u*] \ *matched-vertices*
5   *old* ← *mate-candidate*[*u*]
6   **if** SIZE(*candidate-neighbours*[*u*]) = 0
7   **then** *mate-candidate*[*u*] ← NIL
8   **continue**
9   *mate-candidate*[*u*] ← LAST(*candidate-neighbours*[*u*])
10   **if** *mate-candidate*[*u*] = *old*
11   **then continue**
12   **if** *u* is a ghost vertex
13   **then if** *mate-candidate*[*u*] is a ghost vertex
14   **then continue**
15   *next-candidates* ← *next-candidates* ∪ {*mate-candidate*[*u*]}
16   *next-candidates* ← *next-candidates* ∪ {*u*}

The denominator of the fraction is equal to the number of vertices in the finest graph that the contraction of u and v would represent. Heavy edges represent many edges in the finer graph. The intuition here is that heavy edges should be more likely to be contracted if they join a small number of vertices than if they join a large number of vertices.

The rating was proposed by Sanders in [61]. It is inspired by the edge edge expansion coefficient $i(G)$ which is defined (see [1], for example) as

$$i(G) := \min_{U \subseteq V} \frac{E(U, V \setminus U)}{|U|}.$$

Subgraphs induced by vertex sets U that yield small values for the fraction are intuitively "not strongly connected" to the rest of the graph. Selecting edges e that yield high values for expansion(e) should be a good heuristic for contracting edges that connect vertices within such sets U. ◇

Definition 5.6.3 (Edge Density): The edge density rating of an edge is defined as the number of edges in the cluster per vertices in the cluster:

$$\text{edge-density}(e = \{u, v\}) := \frac{\textit{internal-edges}(u) + \textit{internal-edges}(v) + w(e)}{w(u) + w(v)}.$$

The intuition here is that a high number of edges relative to the number of vertices implicates a good clustering. We are not aware of the existance of this metric in the literature. ◇

Definition 5.6.4 (Edge Density2): The edge density rating of an edge is defined as the number of edges in the cluster per possible number of edges in the cluster

$$\text{edge-density2}(e = \{u, v\}) := \frac{\textit{internal-edges}(u) + \textit{internal-edges}(v) + w(e)}{(w(u) + w(v))^2}.$$

A complete undirected graph with n vertices has $n(n-1)/2$ edges. Since all edges are rated with the same function, the factor $1/2$ has no influence, making n^2 a good approximation with a bias against clusters with few vertices. Again, we are not aware of the existance of this metric in the literature. ◇

Definition 5.6.5 (Inner_outer): The *inner_outer* edge rating is defined as the weight of the contracted edge divided by the number of edges out of the resulting cluster.

$$\text{inner\_outer}(e = \{u, v\}) := \frac{w(e)}{\sum_{f=\{u,x\} \in E} w(f) + \sum_{f=\{v,x\} \in E} w(f) - 2w(e)}.$$

The intuition is that the number within a cluster should be large compared to the number of edges out of the cluster. Again, we are not aware of the existance of this metric in the literature.

Note that different from *inner_outer2* (Definition 5.6.6), we do not consider the internal edges of u and v. Adding them in the denominator would favour large clusters that already have a large number of vertices and edges in them. ◇

For the following, we define interface-edges(u, v) as the sum of the weights of the edges adjacent to u and v without the weight of the edge between u and v.

Definition 5.6.6 (Inner_outer2): The *inner_outer2* edge rating is defined as the number of edges in the resulting cluster per edges out of the cluster:

$$\text{inner\_outer2}(e = \{u, v\}) := \frac{\text{internal-edges}(u) + \text{internal-edges}(v) + w(e)}{\sum_{f=\{u,x\}\in E} w(f) + \sum_{f=\{v,x\}\in E} w(f) - 2w(e)}.$$

Again, the intuition is that the number within a cluster should be large compared with the number of edges out of the cluster. We are not aware of the existance of this metric in the literature either. ◇

Definition 5.6.7 (Coverage): The edge rating variant *coverage* rates the edges according to the number of internal edges the contraction would create in the resulting cluster:

$$\text{coverage}(e = \{u, v\}) := w(e) + \text{inner-edges}(u) + \text{inner-edges}(v).$$

The aim is to maximize the coverage metric ([21]) of the resulting clustering. This is the same as minimizing the edge-cut, i.e. the number of internal edges. ◇

Definition 5.6.8 (Performance): We consider the clustering $\mathcal{C} = (C, \overline{C})$ where C is the cluster resulting from contracting e and $\overline{C}$ is its complement. Then, the *performance* edge rating is defined as

$$\text{performance}(e = \{u, v\}) := \text{performance}(\mathcal{C}).$$

n and m are the number of vertices and edges in the original graph. The performance edge rating considers the cluster resulting from the contraction of e and its complement and computes the performance between them. The aim is to optimize the cluster metric *performance* ([21]) for the resulting graph. ◇

Definition 5.6.9 (Performance 2): n, m, C, C' and $\mathcal{C}$ are defined as in Definition 5.6.8. The *performance2* edge rating is defined similar to *performance* but we consider the internal edges of the resulting cluster instead of $m(C)$. Again, the aim is to optimize the cluster metric *performance* ([21]).

$$\text{performance}(e = \{u, v\}) := \frac{2\overline{m}(\mathcal{C}) - 2(m(C)) + \sum_{M\in\mathcal{C}} |M|(|M| - 1)}{n(n-1)}.$$

◇

6. Experimental Results

In this chapter, we benchmark our partitioner program. As a reminder:

The program first loads the graph from the disk in parallel. Then, it executes recursive coordinate bisection and redistributes the graph accordingly. Afterwards, it coarsens the graph until the termination condition described in Section 4.1 is fulfilled. Then, the coarsest graph is collected an a sequential partitioning program is executed. The coarsest graph is distributed according to the initial partition. Finally, the graph is uncoarsened and at each step, the partition is refined.

The program can be parametrized in various ways and one can imagine many more parameters. Here, we consider the following parameters.

- Choice of the *edge rating* used for the matching in the coarsening phase.

- Choice of the sequential partitioning library for the initial partition.

One aim of this thesis is to select a *edge rating* for the coarsening step that yields "well"-coarsened graphs. We consider the edge cut of the partition of the coarsest graph, the *initial partition*, after the coarsening step as our primary metric for the quality of the coarsened graphs. The coarsening should contract many edges but still not force the initial partitioning algorithm to yield high edge cuts or balances. The running time of the initial partitioning algorithm is no primary goal when evaluating the quality of the coarsening phase.

For this reason, we first evaluate several libraries for the initial partitioning in Section 6.2. The most promising candidate for the initial partitioning library is SCOTCH [58].

Then, we perform a preliminary study regarding the edge rating variants using our experimental sequential coarsening program in Section 6.3. After selecting the most promising variants, we will run our parallel coarsening algorithm using them and show the results, in terms of quality, in Section 6.4. The strong and scalability properties of our program are considered in Sections 6.5 and 6.6.

In this chapter, we will label the results of our program with the name of the edge rating variant we used, i.e. *expansion*, *inner_outer*, *weight*, *inner_outer2*, and *edge_density*.

Most figures and tables can be found in Section 6.8 (p. 65. They were moved there to improve readability.

6.1. Benchmark Graph Set

As with most parallel programs, the performance of our parallel partitioner depends on the initial distribution of underlying datastructure. However, this is the very problem

6. Experimental Results

graph	n	m	#c
3elt	4 720	13 722	
3elt_dual	9 000	13 278	
whitaker3	9 800	28 989	
crack	10 240	30 380	
4elt	15 606	45 878	
whitaker3_dual	19 190	28 581	
crack_dual	20 141	30 043	
4elt_dual	30 269	44 929	
shock.9	36 476	71 290	
pwt	36 519	144 794	57
body	45 087	163 734	351
bracket	62 631	366 559	
tooth	78 136	452 591	
ocean	143 437	409 593	
auto	448 695	3 314 611	
bel	463 514	591 882	615
nld	893 041	1 139 540	690
deu	4 378 446	5 483 587	2 590
eur	18 029 721	22 217 686	14 299
af_shell9	504 855	8 542 010	
audikw_1	943 695	38 354 076	
af_shell10	1 508 065	25 582 130	
cage15	5 154 859	94 044 692	

Table 6.1: Basic properties of the graphs from our benchmark set. We consider three groups, separated by lines in the table above: FEM graphs, street networks and sparse matrices. Within their groups, the graphs are sorted by size. The number of edges is given as *undirected* edges. Coordinates are available for all graphs but *auto* and the matrices. The number of components (#c) is only shown when $\neq 1$.

we want to solve.

For graphs that have coordinates associated with their vertices, a good initial distribution can be computed using recursive coordinate bisection. We mainly limit our evaluation to such graphs.

We also consider graphs resulting from matrices: The matrix is interpreted as the adjacency matrix of the graph. In most cases, the non-zero entries are close to the diagonal. This means that the resulting graph's vertex numbering is "local" in the following sense: If two vertices u and v are connected then $|u - v|$ is "small" in most cases.

Table 6.1 (p. 52) shows basic properties of the graphs we use for our benchmarks. We consider three groups of matrices, they are separated by lines in the table. The first group are the graphs from Walshaw's Graph Partition Archive [73] for which we could find coordinates and *auto*. The second group are graphs of Belgium's, The Netherlands', Germany's and Europe's street network. These graphs were provided by the PTV AG, Karlsruhe and obtained from [23]. The third group are sparse matrices from the Florida Sparse Matrix Collection [10].

We have coordinates for all graphs of the first two groups excluding *auto*.

Note that some of the graphs have more than one component. However, they generally have one large component, few very small ones and most of the components are isolated vertices.

6.2. Choice of the Sequential Partitioning Library

We want to use the edge cut of the coarsest graph as our primary quality metric for the edge rating used in the coarsening. For this, we need a sequential graph partitioning library. Remember that two partitions can only be compared in a sensible way if their balances are very close, ideally 1.

Thus, we have to make sure that we use a partitioning library that yields partitions with a consistent balance: Given different seeds for their random number generator, the variance in balance should be small. Ideally, the variance of the initial edge cuts should also be small, so we can compare their means/medians and get statistically significant results.

In this section, we describe the experiment we performed to make an informed choice of the library for the initial partitioning. The result is that SCOTCH [58] appears to be the most promising candidate for such a library.

6.2.1. The Experiment

We are considering the variants METIS [42], PARTY [55] and SCOTCH [58] for the initial partition of the coarsest graph. For METIS, the recursive bisection based code PMETIS is used. We did the following experiment:

The two graphs *3elt* and *ocean* were partitioned into $k = 2, 8, 64$ parts. *3elt* is the smallest graph in Walshaw's collection, *ocean* is the largest graph with coordinates and the fifth largest overall. $k = 2$ is the smallest values considered by Walshaw's benchmark,

$k = 64$ is the largest, $k = 8$ is in between. Basic properties of the two graphs can be found in Table 6.1 (p. 52).

The graphs were coarsened using the *kmetis* program (variant *metis*) and using our sequential implementation using GPA (*gpa*) and GREEDY (*greedy*) matching. The edge rating is *weight* in all cases. For the initial partition of the coarsest graph, the partitioning was run using the recursive bisection algorithms by METIS, PARTY and SCOTCH. Finally, a refinement step from the METIS program is run.

The partitioning was run for 50 times with different seeds. The pairs of edge cut and balance of the initial partition are plotted in Figure 6.2 (p. 66). For each run of the experiment, a point is shown. The balance of the initial partitioning is shown on the x axis, the initial edge cut is shown on the y axis.

Box-plots of the balance of the initial partitioning can be found in Figure 6.3 (p. 67). Box-plots of the initial edge cut can be found in Figure 6.4 (p. 68).

6.2.2. Evaluation

In the beginning, we will focus on the resulting balances.

Consider the scatter plots in the left column Figure 6.2 (p. 66). These plots show the pairs of balance and edge cut of the initial partitioning for the graph *3elt*.

First, we notice the following for $k = 64$: There are only few values for the initial balance along which the initial edge cut values are positioned. A likely explanation is that the initial partitioning algorithm creates partitionings that are close to local minima of the METIS refinement step executed after it. This refinement step is able to improve the edge cut, keeping the balance constant. Because the refinement is randomized, it yields different values for the edge cut.

For $k = 2$, the point cloud is very dense and the range of the balance values is from 1 to 1.05. We can see that using SCOTCH for the initial partitioning consistenty yields low balance values. METIS yields low balance values, too. However, the points stray farther than for SCOTCH. Whereas SCOTCH almost always keeps the balance below 1.01, METIS has the largest outliers. PARTY yields some small balances, too, but its balance values stray the most. We can verify these observations by looking at the box plots in Figures 6.3a (p. 67) and 6.4a (p. 67). The first one shows the balance values and the second one shows the edge cut values of the initial partitioning using the nine variants. Again, using SCOTCH as the partitioning yields the best balances, followed by METIS and then by PARTY. The values from SCOTCH do not stray very far. All variants yield initial edge cuts in similar ranges (120-160). PARTY yields the best values, followed by METIS and SCOTCH. Remember that in most cases, partitionings with higher balances yield lower edge cuts, so this is to be expected.

For $k = 8$ and $k = 64$, using SCOTCH for the initial partitioning yields the most consistent balance values, too. Again, the initial edge cut is higher than when using METIS and PARTY which is to be expected considering their higher balances. We can verify these observations from the scatter plots by considering the left columns of Figures 6.3 (p. 67) and 6.4 (p. 68).

Furthermore, we report that the median number of levels in the coarsening hierarchy is 8 for $k = 2$ and $k = 8$ with few occurences at 7 and 9, the coarsest level for $k = 64$ is 4 in all cases.

Now, consider the scatter plots in the right column of Figure 6.2 (p. 66). These plots show the pairs of initial balance and edge cut for the graph *ocean*.

For $k = 2$, using METIS for the initial partitioning yields the best results in terms of balance. The second best partitioning variant is SCOTCH, PARTY is the worst. The separation between the three variants is clearly distinguishable, the balances returned by PARTY stray greatly. All yield values for the initial edge cut in the same range.

For $k = 8$, the relative ranking is the same. However, the points are not clearly separated in terms of balance any more. For $k = 64$, the relative ranking is reversed: SCOTCH yields the best balances, followed by PARTY and METIS. While SCOTCH and PARTY yield low variance in the balance, METIS has a relatively high variance. Again, we can verify these observations from the scatter plots by considering the right columns of Figures 6.3 (p. 67) and 6.4 (p. 68).

Now, let us consider the edge cuts. For the graph *ocean*, the *metis* matching variant is consistently worse than the *gpa* and *greedy* variants in terms of initial edge cut.

For the graph *3elt*, things are less clear. For each initial partitioning variant, consider the different matching variants. This way, the results are somewhat comparable since they have a similar balance. There is no clear winner. However, the edge cuts resulting from using *metis* for the matching have a higher median than when using the two other variants for all considered values of k.

Consider the edge cuts yielded by the different initial partitioning algorithm for each maching algorithm. For *ocean*, choosing the best initial partitioining scheme does not let the edge cuts degrade greatly. The difference is greater for *3elt*. However, if our reasoning regarding the alignment of points on vertical lines is correct then this is a problem with the initial refinement rather than the partitioning algorithm.

6.2.3. Summary

As discussed above, using SCOTCH for the initial partitioning yields the best balances for *3elt* and the best result for *ocean* when $k = 64$. It yields the second best results on *ocean* for other values of k. For *ocean*, the edge cut does not degrade greatly. For *3elt*, it does but we assume that this is a problem with the initial refinement step. Also, the number of vertices and edges is roughly 30 times larger in *ocean* than it is in *3elt* which makes it the more important test subject for the initial partitioning algorithm of a scalable coarsening phase.

For all following, we will use SCOTCH for computing the initial partition. By the results in this section, we expect it to yield the most consistent initial partitions in terms of balance. This is important so the initial edge cuts resulting from two different coarsening variants can be compared in a meaningful way.

6.3. Preliminary Study of Edge Ratings

In this section, we describe a preliminary study of the effect of edge ratings in the coarsening. The aim is to select a subset of promising edge ratings from the total set of edge ratings. We also compare the effect of the matching algorithms GPA, GREEDY, and the coarsening of KMETIS.

To the best of our knowledge, the software packages described in Section 3.2.6 use the same matching algorithm that KMETIS uses: SHEM (see Section 5.2) or variations of it. Thus, we only consider KMETIS at this point and expect the coarsened graph to be similar for the other software packages.

We do not consider running time since we use our sequential coarsening code. This code has not been tuned so comparisons in terms of running time are not meaningful.

We ran our sequential coarsening algorithm on the FEM graphs from the benchmark set of Table 6.1 (p. 52). These graphs were the only ones we had acquired at the time we made the benchmarks.

The sequential algorithm was run 50 times with each combination of the matching algorithms *gpa* and *greedy* and the edge rating variants described in Section 5.6. Additionally, we ran our *shem* implementation with edge weights as ratings and the *metis-shem* algorithm on the graphs, each for 50 times. From here on, we will refer to these combinations as ⟨matching⟩.⟨edge rating⟩. After the coarsening, we used SCOTCH to compute an initial partition of the graphs and ran one METIS refinement step on the initial partition of the coarsests graph.

By our analysis in Section 6.2, we expect balance of the resulting initial partitions not to stray too far and thus yield comparable results. Nevertheless, we will consider the balance at the end of this section. From here on, we will refer to the result of this refinement step as the initial partition. The edge cut of the initial partition is the initial edge cut, the balance is the initial balance.

For each coarsening variant and value of k, Table 6.3 (p. 69) shows the geometric mean of the yielded initial edge cuts. Table 6.4 (p. 70) shows the improvement (see Section 2.6.1) over the result of KMETIS in percent. The last column shows the mean of the improvement values in each row. The rows are sorted by the overall average improvement. The best value in each column is printed in bold letters.

Based on the average improvement of the geometric means, we can make the following ranked groupings of our coarsening variants:

1. *inner_outer* and *expansion*

2. *weight*

3. *edge_density* and *inner_outer2*

4. *edge_density2* and *coverage*

5. *performance2* and *performance*.

Section 6.8 contains Tables 6.5 (p. 71), 6.6 (p. 72), 6.7 (p. 73), and 6.8 (p. 74). These tables show the initial balance and number of coarsest vertices resulting from the experiment in this section and the improvements.

We can see that the difference in balance between the variants is small (the average improvement in the geometric means is < 0.6 %). The difference in coarsest vertex count is also small (≤ 25.8 %). Thus, the comparison and rankings are not invalidate by large differences in terms of these metrics.

6.4. Quality of Parallel Coarsening

In this section, we will consider the quality of the parallel coarsening phase.

We ran our parallel coarsening program on the graphs from Table 6.1 for 20 times with different seeds. We also coarsened the graphs using the sequential KMETIS that uses SCOTCH for the initial partition.

We do not show quality results for PARMETIS for two reasons. First, PARMETIS stops its coarsening with a condition different from our coarsening phase and KMETIS (also see Section 4.2): It stops when the global vertex count drops below $25k$. Second, we did not change the PARMETIS code to use SCOTCH for the initial partitions.

The initial edge cuts yielded by PARMETIS were consistently worse than the ones yielded by KMETIS. Also, we observed that KMETIS generally yielded better results than PARMETIS for the final partitions.

All programs were instrumented to print the edge cut and balance of the partitioning result of the coarsest graph. We will refer to the means of these metrics as the *initial edge cut* and *initial balance* from here on. The program also printed the number of vertices in the coarsest (*coarsest vertex count*) graph.

Note that we consider the initial balance and number of coarsest vertices as secondary quality metrics. As explained in Section 6.2, the balance is considered to make sure that the edge cut is not lower because of missing balance. We also consider the number of coarsest vertices for two reasons:

First, all vertices of the original graph that were contracted into one vertex of a coarser graph belong to the same block in the partition of the coarsest graph. Thus, the initial partitioning algorithm cannot assign them freely to the blocks of the partition. Second, yielding a higher number of coarsest vertices means that less time has been spent in coarsening.

KMETIS was executed with access to 16 GiB of memory. Most other variants were assigned 2 GiB of main memory per process. For *cage15*, our coarsening phase needed 8 GiB of main memory per process for $k = 4$ and 4 GiB per process for $k = 8$. Our coarsening phase needed 16 GiB per process for $k = 2$ on *cage15*.

Consider Table 6.9 (p. 75). The table shows the average improvements over KMETIS in edge cut for the edge rating variants and different values of k. The improvements were averaged over the FEM graphs from the benchmark set, the street network graphs, the graphs generated from matrices and all graphs.

We can see that the grouped ranking from Section 6.3 still holds when considering

the overall average improvements. However, *inner_outer2* yields better results than *edge_density*. The ranking by overall average improvement yields

1. *inner_outer*

2. *expansion*

3. *weight*

4. *inner_outer2*

5. *edge_density*.

Section 6.8 contains tables with more detailed values of the initial edge cut, balance and coarsest vertex count. The values are shown for the best edge rating *inner_outer*, the classic edge rating *weight* and KMETIS. Additionally, the improvements over KMETIS are shown for *inner_outer* and *weight*. We chose to show details for *inner_outer* since it is the best edge rating by our ranking and *weight* because it is the "classic" one. KMETIS was chosen as a reference.

Tables 6.10 (p. 76), 6.11 (p. 77), and 6.12 (p. 78) show the initial edge cuts. Tables 6.13 (p. 79) and 6.14 (p. 80) show the improvements in initial edge cut over KMETIS. Tables 6.15 (p. 81), 6.16 (p. 82), and 6.17 (p. 83) show the initial balance values. Tables 6.18 (p. 84) and 6.19 (p. 85) show the improvements in initial balance over KMETIS. Tables 6.20 (p. 86), 6.21 (p. 87), and 6.22 (p. 88) show the coarsest vertex count. Tables 6.23 (p. 89) and 6.24 (p. 90) show the improvements in coarsest vertex count over KMETIS.

We can see that there are no big irregularities that invalidate taking the averages over all graphs in each group. Also, for *inner_outer*, the average initial edge cut is mostly better than for KMETIS. Degradations are smaller than 0.5 % in most cases.

The coarsest vertex count is not much larger than KMETIS than for *inner_outer* in most cases. In some cases, KMETIS yielded twice the number of coarsest vertices or more than *inner_outer*. This raises the question whether it is fair to compare the initial edge cuts.

Completing the graph partitioning process and then comparing the final partition would allow for an easier comparison of edge ratings. Thus, we can only give the improvements in edge cut over KMETIS as a preliminary result.

However, we saw in Section 6.3 that the geometric means over all coarsest vertex counts for the sequential coarsening are similar (Tables 6.7 and 6.8). Thus, we are confident that the ranking of the edge ratings themselves when used in our coarsening phase makes sense.

6.5. Strong Scalability Studies

In this section, we perform studies on the strong scaling properties of our program. For this, large graphs are interesting. We chose the largest graphs from the street network family and the matrix family, *eur* and *cage15*.

Section 6.5.1 shows the results for *eur* Section 6.5.2 shows the results for *cage15*.

6.5.1. Strong Scalability on *eur*

We coarsened the graph *eur* into $k = 2, 4, \ldots, 1024$ parts using k processes. Figures 6.5 (p. 91) and 6.6 (p. 92) show the running time and work of the coarsening using all edge rating variants. For each data point, an error bar is shown to visualize the variance of the data. The bars show a confidence level of 95 %.

Overall Running Time and Work. When considering the initial partitioning time, *inner_outer* yields the lowest time together with *inner_outer2* When not considering the initial partitioning time, *inner_outer*, *expansion*, and *inner_outer2* yield the lowest running time.

In Section 6.4, we saw that *inner_outer* was the second best edge rating in terms of initial edge cut. In the following, we will thus focus on this edge rating and the "classic" variant *weight*.

When not considering the time spent in initial partitioning, we can see that the work is almost constant for $k = 4$ to 256. Beyond that, it raises to 512 and then more steeply to 1024. Thus, the algorithm shows good strong scalability on *eur* up to $k = 256$.

The small increase in work from $k = 2$ over 4 to 8 could be explained by the structure of our machine (also see Section D.1): With two processes, each process can run on its own processor. For $k = 4$ and 8, more processes have to share the processor's cache.

Running Time and Work of the Separate Steps. This raises the question which part of the coarsening phase does not scale beyond $k = 256$ on *eur*. Figures 6.7 (p. 93) and 6.8 (p. 94) show the time spent and work performed in the different steps of the coarsening for *weight* and *inner_outer*.

The recursive coordinate bisection accounts for a neglegible part of the running time and work. Judging from the work, I/O scales well to $k = 512$ and increases for $k = 1024$. The same is true for the matching and contraction step but its increase for $k = 1024$ is larger still. The initial partitioning with SCOTCH accounts for more than half of the time and work for $k = 1024$ and for most of the time for $k = 512$.

The I/O phase of our program mainly consist of reading contiguous binary data from the parallel file system into main memory. Thus, there is little hope that we can lower the work and time spent here in our code. The time spent in I/O should thus be ignored when considering the scalability of the coarsening phase.

It also makes sense to ignore the time spent in initial partitioning when measuring the scalability of the coarsening phase. The reason is that on the one hand, the initial partitioning library could be switched to a faster one like JOSTLE or KMETIS. On the other hand, JOSTLE is a high-quality graph partitioning library that does not use an initial partitioning algorithm per-se but simply coarses the graph down to k vertices. Our coarsening phase could be used in a similar way.

This leaves the question of why the matching and contraction step does not scale for $k = 1024$.

Running Time and Work of the Parts of the Matching and Contraction Step. Figure 6.9 (p. 95) and 6.10 (p. 96) show the running time and work of the different parts of the matching and contraction step.

The local part of the matching scales nicely and has almost constant work for all k. The work of the initialization part of the distributed matching algorithm has constant work for $k \leq 512$ but increases for $k = 1024$. The contraction and migration part and the gap graph matching part scale well up to $k = 256$ but beyond this point, the work increases, dramatically in the case from $k = 512$ to 1024.

The time spent both in contraction and migration part and the gap graph mathing is below 1 s for $k = 1024$. Further instrumentation reveals that a lot of time here is spent in initialization and book-keeping for this value of k. Also, the number of vertices and edges for each process is very slow. This is the reason that these parts do not scale well for these values of k.

However, there is still be room for improvement in our implementation: At the moment, each process allocates buffer management data structures for each other process. If each process would compute its neighbours, the book-keeping time could be reduced and scalability improved.

Comparison of Time and Work with KMETIS **and** PARMETIS**.** Figures 6.11 (p. 97) and 6.12 (p. 98) shows the running time and work of the coarsening phases of our parallel coarsening code with edge ratings *inner_outer* and *expansion*, PARMETIS, and KMETIS. The running time and work is shown excluding I/O and initial partitioning and excluding I/O, coordinate based prepartitioning and initial partitioning.

We do not show the time including I/O since the PARMETIS program reads in the graph centrally in a text based format. Then, it performs spatial sorting of the vertices for the coordinate based prepartitioning and distributes the vertices to their owners. Our partitioner reads the file completely distributedly from a binary file.

When considering I/O, PARMETIS only gets slightly faster than KMETIS for $k = 32$ to 256.

The spatial sorting used by PARMETIS for the coordinate based prepartitioning is faster for *eur* than our coordinate based recursive parallel bisection. We could change the parallel QUICKSELECT algorithm used there for a parallel version of the randomized selection by Blum et. al ([4]) and improve the running time of our code. Thus, it also makes sense to consider the running time and work without the coordinate based prepartitioning.

When I/O is not considered then PARMETIS is faster than KMETIS for all k and our coarsening code is faster for $k \geq 16$. Our coarsening code is faster than PARMETIS for $k \geq 256$. The same is true when the recursive coordinate based bisection is not considered, but the difference in running time is larger.

The work of PARMETIS is constant for $k \leq 128$ when I/O is ignored. For greater k, PARMETIS does not show good scalability on *eur*.
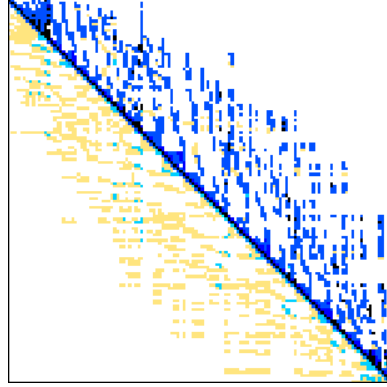
Figure 6.1: Structure of *cage15*.

Comparison of Quality Metrics with KMETIS **and** PARMETIS**.**   Figure 6.13 (p. 99) and 6.14 (p. 100) show three quality metrics of the coarsening phases on *eur*: Number of vertices in the coarsest graph, the edge cut and the balance of the initial partition. The values are shown for KMETIS with SCOTCH for the inital partitioning, PARMETIS, and our coarsening phase using *inner_outer* and *expansion* for the edge rating.

We can see that *inner_outer* and *expansion* yield the best results in all metrics for almost all k. Only the balance is worse than for KMETIS for $k \geq 512$.

6.5.2. Strong Scalability on *cage15*

Besides the graph of the European street network, we also consider the graph resulting from matrix *cage15* from the University of Florida Sparse Matrix Collection ([10]) for strong scaling studies. The number of vertices is 5 154 859, the number of undirected edges is 94 044 692. The matrix arises from the so-called cage model of DNA electrophoresis ([68]). The structure of the matrix is shown in Figure 6.1. Colored points show non-zero entries.

We generated the graph from the matrix by considering the matrix as the adjacency matrix of a graph, making directed edges undirected, removing all edge weights and duplicate edges.

Overall Running Time and Work.   Figures 6.15 (p. 101) and 6.16 (p. 102) show the running time and work of the coarsening using all edge rating variants. When not considering the initial partitioning, the time spent in coarsening, the running times are mainly falling. The reason for local maxima can be attributed to the irregularity of the graph and the missing locality of edges. The two best variants by running time are *inner_outer* and *expansion*. *weight* is the third best edge rating variant.

When considering work, we can see that for *inner_outer* and *expansion*, the work is slightly raising for $k \leq 512$ and roughly doubles from $k = 512$ to 1024.

Running Time and Work of the Separate Steps.   Figures 6.17 (p. 103) and 6.18 (p. 104) show the time spent and work performed in the different steps of the coarsening for *weight* and *inner_outer*. As it was the case for *eur*, the time and work spent in I/O is neglegible. The time spent in the matching and contraction part is decreasing, the work is increasing almost linearly. The initial partitioning dominates the work and running time for $k = 1024$. For $k < 512$, the time spent in the initial partitioning is small.

Running Time and Work of the Parts of the Matching and Contraction Step.   Figure 6.19 (p. 105) and 6.20 (p. 106) show the running time and work of the different parts of the matching and contraction step. The initialitialization of the matching algorithm and the local matching have a low work and short running time. The time spent in contraction and migration step falls, but the work raises with increasing k. The same is true with the gap graph matching.

Comparison of Time and Work with KMETIS **and** PARMETIS**.**   Figures 6.21 (p. 107) and 6.22 (p. 108) show the running time and work of the coarsening phases of our parallel coarsening code with edge rating *inner_outer*, PARMETIS, and KMETIS. When not considering the time spent in I/O, our coarsening program is faster than PARMETIS for $k \geq 512$. PARMETIS is faster than KMETIS for all k, our coarsening code is faster than KMETIS for $k \geq 128$.

Comparison of Quality Metrics with KMETIS **and** PARMETIS**.**   Figures 6.23 (p. 109) and 6.24 (p. 110) show the quality metrics coarsest vertex count, initial edge cut and initial balance of the resulting initial partition.

The number of coarsest vertices of *expansion* is mostly close to the one of KMETIS, but larger than PARMETIS. The initial edge cut of *expansion* is better than the one of PARMETIS and KMETIS. The balance of *expansion* is better than the balance yielded by the two other variants .

The metrics for *inner_outer* have a similar quality, but for $k = 1024$, the number of caorsest vertices is much higher than for *expansion* and the other variants.

6.6. Weak Scalability Tests

In this section, we show weak scaling studies of four coarsening algorithm. We consider the family of Dealaunay triangulations in Section 6.6.1. In Section 6.6.2, we evaluate the weak scalability of our program on random geometric graphs.

6.6.1. Weak Scalability on Delaunay Triangulations.

In this section, we will consider the weak scalability of our coarsening phase on the Delaunay triangulation graph family generated by the generator described in Section B.2. For $k = 2, 4, 8, \ldots, 1024$, we ran the corsening phase on a Delaunay triangulation graph

of a random point set with $2^{16+\log k}$ points. The number of edges is roughly $3n$ (also see Section B.1).

Overall Running Time. Figure 6.25 (p. 111) shows the running time of the coarsening phase on the Delaunay triangulation graphs with and without initial partitioning. When not considering initial partitioning, the running time is seemingly linearly increasing with k up to $k = 512$. From $k = 512$ to 1024, the running time increases stronger than before.

The two best variants are *expansion* and *inner_outer*.

Running Time of the Separate Steps. Figure 6.26 (p. 112) shows the running time of the different steps of the coarsening phase for *weight* and *inner_outer*.

As for the strong scaling studies, the initial partitioning phase running time increases with k.

The other parts take a relatively small portion of the running time and show a similar behaviour to the work in the strong scaling study with *eur*: Their running time increases linearly with a small slope. From $k = 512$ to 1024, the running time of the matching and contraction step increases stronger than before.

Running Time of the Parts of the Matching and Contraction Step. Figure 6.27 (p. 113) shows the running time of the different steps in the matching step for *weight* and *inner_outer*.

The local matching part takes the same time for all k. The initialization of the matching is constant for $k \leq 512$ and increases slightly for $k = 1024$. The gap graph matching part increases slowly with k and jumps from $k = 512$ to 1024. The contraction and migration part dominates the other parts in terms of running time, it raises slowly with k and jumpst from $k = 512$ to 1024.

The problem with scalability beyond 512 processes does not seem to be a problem of the communication system for the weak scalability either. Further instrumentation of the program yields that lots of time is spent in the local computation steps. As for the strong scalability, this could be improved by only allocating buffers and book-keeping data structures for the neighbouring processes and not all of them.

Comparison of Running Time with KMETIS **and** PARMETIS**.** Figure 6.28 (p. 114) shows the running time of our coarsening phase using the edge ratings *inner_outer* and *expansion*, the one of KMETIS, and the one of PARMETIS.

When the time spent in I/O is not considered, our coarsening phase is faster than the one of PARMETIS for $k \geq 256$. When ignoring the time spent in the coordinate based prepartitioning, the break-even of our coarsening phase with PARMETIS is at $k = 128$.

The break-even with KMETIS is at $k = 32$ when considering coordinate based bisection and at $k = 16$ when it is not considered.

n	m	m/n
2^{15}	160 240	4.89
2^{16}	342 127	5.22
2^{17}	728 753	5.56
2^{18}	1 547 283	5.90
2^{19}	3 269 766	6.24
2^{20}	6 891 620	6.57
2^{21}	14 487 995	6.91
2^{22}	30 359 198	7.24
2^{23}	63 501 393	7.57
2^{24}	132 557 200	7.90

Table 6.2: Number of vertices and edges of the random geometric graphs.

Comparison of Quality Metrics with KMETIS **and** PARMETIS**.**   Figures 6.29 (p. 115) and 6.30 (p. 116) show the coarsest vertex count, the initial edge cut and the initial balance of the partitionings resulting from our coarsening phase using *inner_outer* and *expansion*, the one of KMETIS, and the one of PARMETIS.

The coarsest vertex count of *inner_outer* and *expansion* is the same as KMETIS for $k \leq$ 256 and slightly above for $k = 512$. PARMETIS yields lower coarsest vertex counts. The reason is that its breaking condition differs from the one of KMETIS and our coarsening phase.

All variants yield similar edge cuts, *inner_outer* and *expansion* yield the best ones. The resulting balances values are all very small.

6.6.2. Weak Scalability on Random Geometric Graphs.

In this section, we will consider the weak scalability of our coarsening phase on the random geometrc graph family generated by the generator described in Section B.1. For $k = 2, 4, 8, \ldots, 1024$, we ran the corsening phase on a random geometric graph of a random point set with $2^{14+\log k}$ vertices. The number of vertices and edges of the random geometric graphs is shown in Table 6.2.

Overall Running Time.   Figure 6.31 (p. 117) shows the running time of our algorithm with different edge rating variants on the random geometric graphs. When not considering the initial partitioning's running time, the running time raises linearly up to $k = 256$. The slope raises from $k = 256$ to 512 and even more from $k = 512$ to 1024.

Running Time of the Separate Steps.   Figure 6.32 (p. 118) shows the running time of the different steps of the coarsening phase for *weight* and *inner_outer*.

Besides the initial partitioning, the matching and contraction dominates the running time. Recursive coordinate bisection and I/O take neglegible time.

Running Time of the Parts of the Matching and Contraction Step.   Figure 6.33 (p. 119) shows the running time of the matching and contraction step broken down by its parts.

The initialization and local matching do not take much time. The contraction and migration dominates the other parts, followed by the matching in the gap graph.

Comparison of Running Time with KMETIS **and** PARMETIS.   Figure 6.34 (p. 120) shows the running time of the coarsening phase using the edge rating *inner_outer* of our coarsening code, the one of KMETIS, and the one of PARMETIS.

When the time spent in I/O is not considered, our coarsening phase is faster than the one of PARMETIS for $k \geq 128$. When ignoring the time spent in the coordinate based prepartitioning, the break-even of our coarsening phase with PARMETIS is roughly at $k = 64$.

The break-even with KMETIS is at $k = 32$ when considering coordinate based bisection and at $k = 16$ when it is not considered.

Comparison of Quality Metrics with KMETIS **and** PARMETIS.   Figure 6.35 (p. 121) and 6.36 (p. 122) show quality metrics of the resulting partitionings.

Again, *inner_outer* and *expansion* yield coarsest vertex counts that are similar ot KMETIS and PARMETIS yields smaller ones. The edge cut of KMETIS and PARMETIS are very close, *expansion* yields smaller initial edge cuts. The balance value of KMETIS, *inner_outer* and *expansion* are very close, the balance of PARMETIS is worse.

6.7. Summary

In our experimental evaluation, we have seen that

- using the edge ratings *inner_outer* and *expansion* yields better results than the "classic" variant *weight*,

- we can rank the edge ratings by initial edge cut, *inner_outer* is the best one by this metric, followed by *expansion*, *weight* and the other variants and

- our sequential matching algorithm takes a longer time than the one used by PARMETIS but for large numbers of processes, our code is more scalable than PARMETIS both in terms of weak and strong scalability.

6.8. Full Page Figures

This section contains the full-page figures of this chapter.

(a) $k = 2$, graph *3elt*

(b) $k = 2$, graph *ocean*

(c) $k = 8$, graph *3elt*

(d) $k = 8$, graph *ocean*

(e) $k = 64$, graph *3elt*

(f) $k = 64$, graph *ocean*
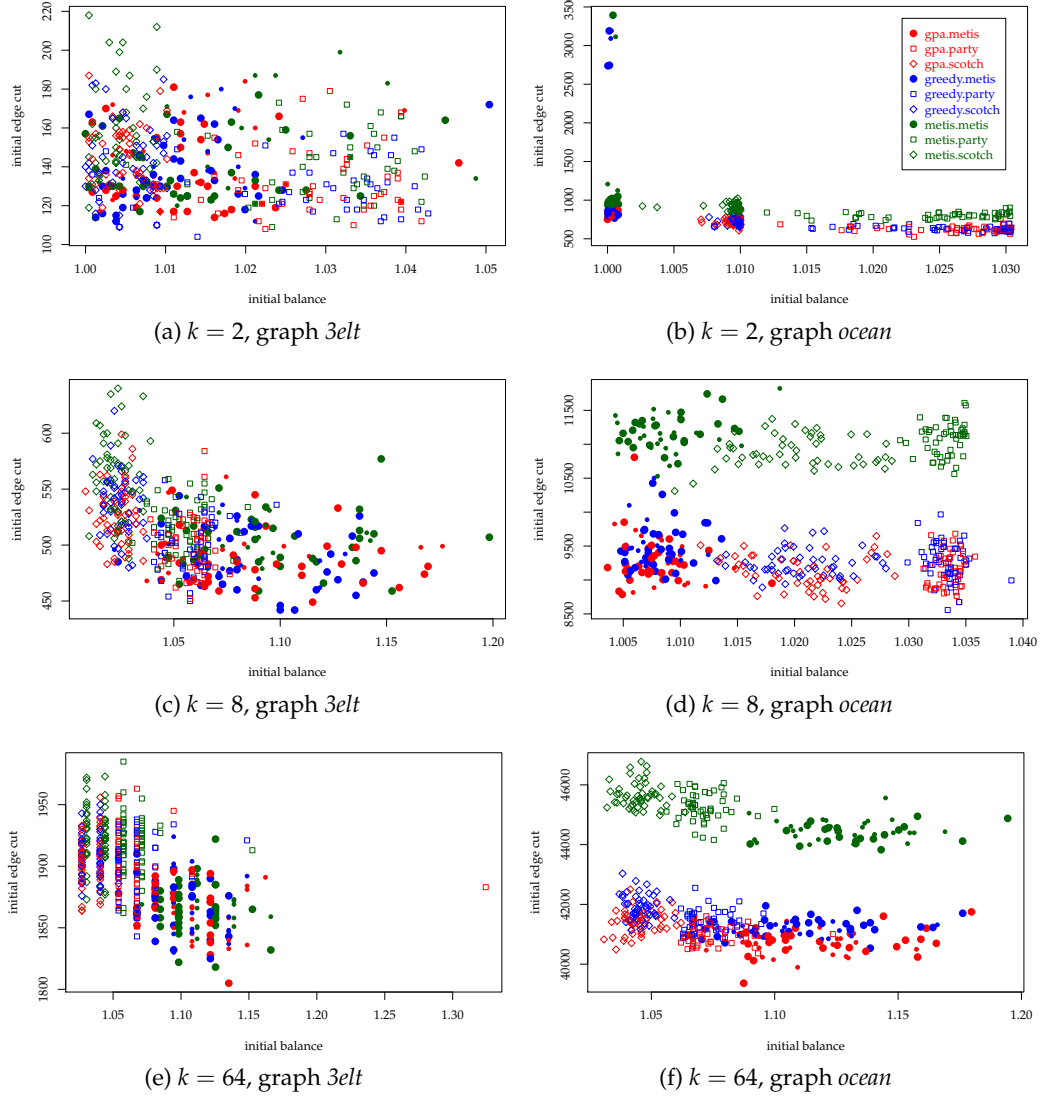
Figure 6.2: Initial partitioning quality of the nine partitioning variants on graphs *3elt* and *ocean*. The key is only shown in the top right plot. Partitioning was performed into 2, 4 and 64 parts. The partitioning variants consist of the matching variant and initial partitioning variant.
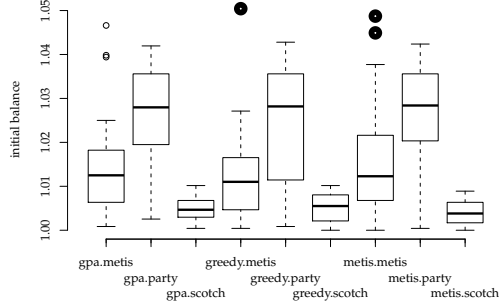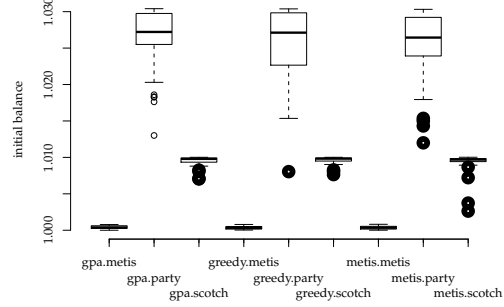
(a) $k = 2$, graph *3elt*
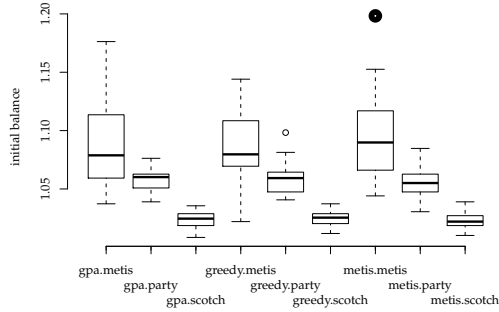
(b) $k = 2$, graph *ocean*
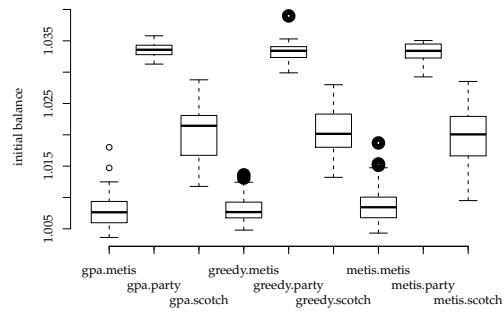
(c) $k = 8$, graph *3elt*
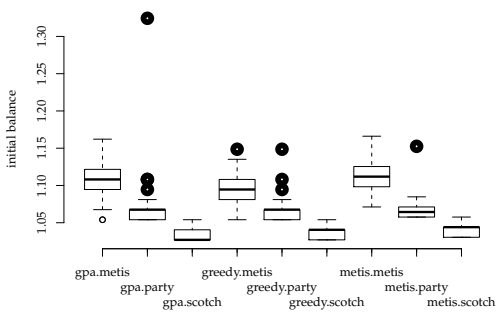
(d) $k = 8$, graph *ocean*
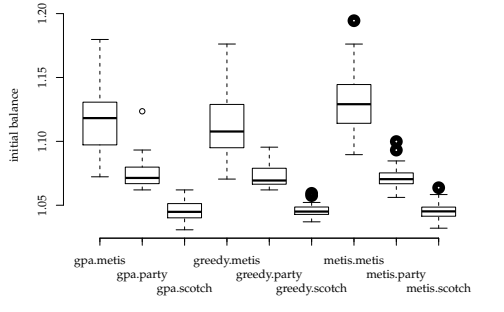
(e) $k = 64$, graph *3elt*

(f) $k = 64$, graph *ocean*

Figure 6.3: Box plots of the balance of the initial partition for the nine variants on graphs *3elt* and *ocean*. Partitioning was performed into 2, 4 and 64 parts. The partitioning variants consist of the matching variant and initial partitioning variant.

(a) $k = 2$, graph *3elt*

(b) $k = 2$, graph *ocean*

(c) $k = 8$, graph *3elt*

(d) $k = 8$, graph *ocean*

(e) $k = 64$, graph *3elt*

(f) $k = 64$, graph *ocean*
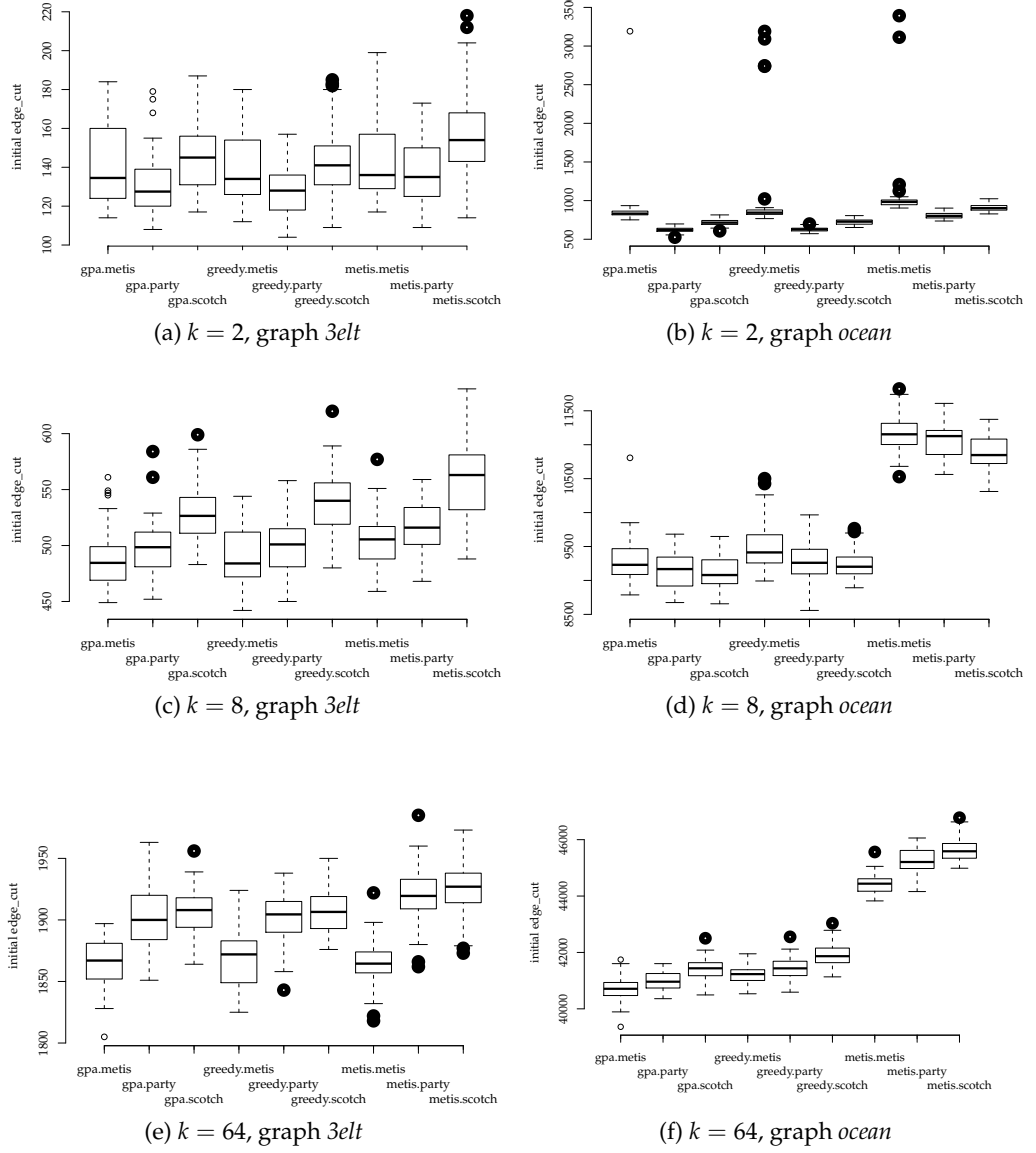
Figure 6.4: Box plots of the edge cut of the initial partition for the nine variants on graphs *3elt* and *ocean*. Partitioning was performed into 2, 4 and 64 parts. The partitioning variants consist of the matching variant and initial partitioning variant.

| variant	2	4	8	16	32	64	128	256	512	1024
gpa.coverage	393	1153	2330	3708	5518	7624	10300	13626	17974	23741
greedy.coverage	408	1250	2426	3845	5643	7698	10370	13663	17970	23782
gpa.edge_density	389	1113	2106	3476	5175	7339	10088	13489	17845	23750
greedy.edge_density	381	1073	2047	3394	5069	7232	9980	13444	17839	23760
gpa.edge_density2	393	1152	2236	3631	5280	7363	10050	13422	17916	23768
greedy.edge_density2	392	1159	2261	3664	5310	7380	10013	13387	17904	23781
gpa.expansion	362	1001	1906	3137	4764	6905	9672	13163	17743	23701
greedy.expansion	360	995	1904	3135	4760	6904	9669	13175	17774	23726
gpa.inner_outer	358	999	1900	3127	4751	6890	9660	13139	17735	23730
greedy.inner_outer	360	996	1899	3135	4753	6910	9672	13155	17774	23727
gpa.inner_outer2	387	1095	2090	3458	5171	7319	10063	13445	17899	23805
greedy.inner_outer2	379	1079	2067	3428	5116	7283	10037	13451	17906	23793
gpa.performance	437	1367	2903	4510	6588	8747	11442	14581	19269	24482
greedy.performance	435	1392	2922	4500	6515	8591	11254	14417	19144	24394
gpa.performance2	423	1306	2740	4286	6290	8405	11097	14307	18876	24246
greedy.performance2	423	1334	2772	4292	6243	8279	10911	14135	18740	24158
gpa.weight	367	1022	1931	3184	4811	6950	9715	13225	17785	23712
greedy.weight	365	1023	1938	3195	4827	6995	9748	13269	17831	23756
metis-shem.weight	372	1046	2007	3310	5043	7326	10323	14252	19653	26887
shem.weight	396	1114	2141	3489	5215	7427	10160	13609	18016	23892

Table 6.3: For each sequential coarsening variant and k, the table shows the geometric mean of the edge cuts for all FEM graphs. The initial edge cuts were measured for the sequential coarsening code with 50 runs each.

variant	2	4	8	16	32	64	128	256	512	1024	average
gpa.inner_outer	**3.8**	4.5	5.3	**5.5**	**5.8**	**6.0**	6.4	**7.8**	**9.8**	11.7	**6.7**
greedy.inner_outer	3.2	**4.8**	**5.4**	5.3	5.8	5.7	6.3	7.7	9.6	**11.8**	6.5
greedy.expansion	3.2	**4.8**	5.1	5.3	5.6	5.8	6.3	7.6	9.6	**11.8**	6.5
gpa.expansion	2.7	4.3	5.0	5.2	5.5	5.7	6.3	7.6	9.7	**11.8**	6.4
gpa.weight	1.3	2.3	3.8	3.8	4.6	5.1	5.9	7.2	9.5	**11.8**	5.5
greedy.weight	1.9	2.2	3.4	3.5	4.3	4.5	5.6	6.9	9.3	11.6	5.3
greedy.edge_density	-2.4	-2.6	-2.0	-2.5	-0.5	1.3	3.3	5.7	9.2	11.6	2.1
greedy.inner_outer2	-1.9	-3.2	-3.0	-3.6	-1.4	0.6	2.8	5.6	8.9	11.5	1.6
gpa.inner_outer2	-4.0	-4.7	-4.1	-4.5	-2.5	0.1	2.5	5.7	8.9	11.5	0.9
gpa.edge_density	-4.6	-6.4	-4.9	-5.0	-2.6	-0.2	2.3	5.4	9.2	11.7	0.5
metis-shem.weight	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0
shem.weight	-6.5	-6.5	-6.7	-5.4	-3.4	-1.4	1.6	4.5	8.3	11.1	-0.4
gpa.edge_density2	-5.6	-10.1	-11.4	-9.7	-4.7	-0.5	2.6	5.8	8.8	11.6	-1.3
greedy.edge_density2	-5.4	-10.8	-12.7	-10.7	-5.3	-0.7	3.0	6.1	8.9	11.6	-1.6
gpa.coverage	-5.6	-10.2	-16.1	-12.0	-9.4	-4.1	0.2	4.4	8.5	11.7	-3.3
greedy.coverage	-9.7	-19.5	-20.9	-16.2	-11.9	-5.1	-0.5	4.1	8.6	11.5	-5.9
greedy.performance2	-13.7	-27.5	-38.1	-29.7	-23.8	-13.0	-5.7	0.8	4.6	10.1	-13.6
gpa.performance2	-13.7	-24.9	-36.5	-29.5	-24.7	-14.7	-7.5	-0.4	4.0	9.8	-13.8
greedy.performance	-16.9	-33.1	-45.6	-36.0	-29.2	-17.3	-9.0	-1.2	2.6	9.3	-17.6
gpa.performance	-17.5	-30.7	-44.6	-36.3	-30.6	-19.4	-10.8	-2.3	2.0	8.9	-18.1

Table 6.4: Shows the improvement in initial edge cut from Table 6.3 of each sequential variant over KMETIS in percent. The last column shows the mean of all values in the row. The rows are sorted by average improvement. In each column, the best value is printed in bold letters.

variant	2	4	8	16	32	64	128	256	512	1024
gpa.coverage	1.006	1.013	1.022	1.031	1.036	1.042	1.047	1.050	1.060	1.116
greedy.coverage	1.006	1.012	1.022	1.030	1.036	1.042	1.045	1.052	1.060	1.115
gpa.edge_density	1.006	1.012	1.021	1.028	1.035	1.041	1.045	1.050	1.061	1.116
greedy.edge_density	1.006	1.012	1.021	1.028	1.036	1.042	1.046	1.050	1.061	1.116
gpa.edge_density2	1.006	1.013	1.021	1.028	1.034	1.041	1.046	1.052	1.060	1.114
greedy.edge_density2	1.006	1.012	1.020	1.028	1.034	1.041	1.045	1.050	1.061	1.115
gpa.expansion	1.006	1.012	1.021	1.028	1.035	1.041	1.044	1.050	1.062	1.115
greedy.expansion	1.006	1.012	1.021	1.029	1.035	1.041	1.045	1.051	1.060	1.116
gpa.inner_outer	1.006	1.012	1.020	1.028	1.034	1.040	1.044	1.050	1.060	1.114
greedy.inner_outer	1.006	1.012	1.020	1.028	1.035	1.041	1.045	1.050	1.061	1.115
gpa.inner_outer2	1.006	1.012	1.021	1.028	1.034	1.042	1.046	1.050	1.061	1.116
greedy.inner_outer2	1.006	1.013	1.021	1.029	1.035	1.042	1.045	1.051	1.062	1.115
gpa.performance	1.006	1.013	1.021	1.030	1.035	1.044	1.048	1.051	1.062	1.117
greedy.performance	1.006	1.012	1.022	1.030	1.036	1.043	1.047	1.052	1.062	1.116
gpa.performance2	1.006	1.013	1.022	1.029	1.035	1.043	1.047	1.051	1.062	1.116
greedy.performance2	1.006	1.013	1.022	1.031	1.037	1.043	1.047	1.052	1.060	1.117
gpa.weight	1.006	1.012	1.021	1.028	1.036	1.041	1.045	1.051	1.060	1.115
greedy.weight	1.006	1.013	1.021	1.028	1.035	1.042	1.045	1.051	1.061	1.116
metis-shem.weight	1.006	1.012	1.021	1.028	1.035	1.043	1.050	1.059	1.079	1.136
shem.weight	1.006	1.012	1.020	1.030	1.037	1.044	1.048	1.050	1.063	1.115

Table 6.5: For each sequential coarsening variant and k, the table shows the geometric mean of the initial balances for all FEM graphs. The initial balances were measured for the sequential coarsening code with 50 runs each.

variant	2	4	8	16	32	64	128	256	512	1024	average
gpa.coverage	0	-0.099	-0.098	-0.292	-0.097	0.096	0.286	0.850	1.761	1.761	0.417
greedy.coverage	0	0.000	-0.098	-0.195	-0.097	0.096	0.476	0.661	1.761	1.849	0.445
gpa.edge_density	0	0.000	0.000	0.000	0.000	0.192	0.476	0.850	1.668	1.761	0.495
greedy.edge_density	0	0.000	0.000	0.000	-0.097	0.096	0.381	0.850	1.668	1.761	0.466
gpa.edge_density2	0	-0.099	0.000	0.000	0.097	0.192	0.381	0.661	1.761	1.937	0.493
greedy.edge_density2	0	0.000	0.098	0.000	0.097	0.192	0.476	0.850	1.668	1.849	0.523
gpa.expansion	0	0.000	0.000	0.000	0.000	0.192	0.571	0.850	1.576	1.849	0.504
greedy.expansion	0	0.000	0.000	-0.097	0.000	0.192	0.476	0.755	1.761	1.761	0.485
gpa.inner_outer	0	0.000	0.098	0.000	0.097	0.288	0.571	0.850	1.761	1.937	0.560
greedy.inner_outer	0	0.000	0.098	0.000	0.000	0.192	0.476	0.850	1.668	1.849	0.513
gpa.inner_outer2	0	0.000	0.000	0.000	0.097	0.096	0.381	0.850	1.668	1.761	0.485
greedy.inner_outer2	0	-0.099	0.000	-0.097	0.000	0.096	0.476	0.755	1.576	1.849	0.456
gpa.performance	0	-0.099	0.000	-0.195	0.000	-0.096	0.190	0.755	1.576	1.673	0.380
greedy.performance	0	0.000	-0.098	-0.195	-0.097	0.000	0.286	0.661	1.576	1.761	0.389
gpa.performance2	0	-0.099	-0.098	-0.097	0.000	0.000	0.286	0.755	1.576	1.761	0.408
greedy.performance2	0	-0.099	-0.098	-0.292	-0.193	0.000	0.286	0.661	1.761	1.673	0.370
gpa.weight	0	0.000	0.000	0.000	-0.097	0.192	0.476	0.755	1.761	1.849	0.494
greedy.weight	0	-0.099	0.000	0.000	0.000	0.096	0.476	0.755	1.668	1.761	0.466
metis-shem.weight	0	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000
shem.weight	0	0.000	0.098	-0.195	-0.193	-0.096	0.190	0.850	1.483	1.849	0.399

Table 6.6: Shows the improvement in initial balance from Table 6.5 of each sequential variant over KMETIS in percent. The last column shows the mean of all values in the row.

variant	2	4	8	16	32	64	128	256	512	1024
gpa.coverage	799	439	365	476	780	1327	2801	5546	10926	17554
greedy.coverage	877	622	440	612	834	1356	2894	5667	10940	17380
gpa.edge_density	714	383	323	441	690	1278	2583	5413	10689	17302
greedy.edge_density	710	387	326	446	696	1241	2559	5346	10631	17293
gpa.edge_density2	677	355	312	395	684	1289	2581	5426	10789	17190
greedy.edge_density2	708	362	310	395	680	1292	2643	5466	10768	17144
gpa.expansion	699	369	327	409	712	1308	2620	5471	10748	17323
greedy.expansion	690	359	318	394	695	1296	2646	5421	10745	17251
gpa.inner_outer	705	355	307	392	680	1285	2601	5464	10846	16938
greedy.inner_outer	670	354	303	383	671	1276	2634	5509	10775	16934
gpa.inner_outer2	675	358	309	389	659	1247	2605	5431	10835	16924
greedy.inner_outer2	688	349	300	378	651	1236	2565	5438	10808	16890
gpa.performance	794	411	331	444	679	1253	2714	5584	11092	17247
greedy.performance	817	519	412	491	689	1257	2755	5560	11218	17238
gpa.performance2	793	411	332	443	677	1261	2690	5565	11067	17250
greedy.performance2	816	524	412	490	689	1261	2740	5545	11174	17236
gpa.weight	748	422	358	506	741	1310	2617	5422	10771	17320
greedy.weight	729	394	334	459	710	1282	2596	5382	10678	17192
metis-shem.weight	1327	688	590	693	1098	1866	3308	5907	9933	13682
shem.weight	672	350	321	394	691	1290	2608	5366	10767	16892

Table 6.7: For each sequential coarsening variant and k, the table shows the geometric mean of the coarsest vertex count for all FEM graphs. The initial balances were measured for the sequential coarsening code with 50 runs each.

variant	2	4	8	16	32	64	128	256	512	1024	average
gpa.coverage	39.8	36.2	38.1	31.3	29.0	28.9	15.3	6.1	-10.0	-28.3	18.6
greedy.coverage	33.9	9.6	25.4	11.7	24.0	27.3	12.5	4.1	-10.1	-27.0	11.1
gpa.edge_density	46.2	44.3	45.3	36.4	37.2	31.5	21.9	8.4	-7.6	-26.5	23.7
greedy.edge_density	46.5	43.8	44.7	35.6	36.6	33.5	22.6	9.5	-7.0	-26.4	23.9
gpa.edge_density2	49.0	48.4	47.1	43.0	37.7	30.9	22.0	8.1	-8.6	-25.6	25.2
greedy.edge_density2	46.6	47.4	47.5	43.0	38.1	30.8	20.1	7.5	-8.4	-25.3	24.7
gpa.expansion	47.3	46.4	44.6	41.0	35.2	29.9	20.8	7.4	-8.2	-26.6	23.8
greedy.expansion	48.0	47.8	46.1	43.1	36.7	30.5	20.0	8.2	-8.2	-26.1	24.6
gpa.inner_outer	46.9	48.4	48.0	43.4	38.1	31.1	21.4	7.5	-9.2	-23.8	25.2
greedy.inner_outer	49.5	48.5	48.6	44.7	38.9	31.6	20.4	6.7	-8.5	-23.8	25.7
gpa.inner_outer2	49.1	48.0	47.6	43.9	40.0	33.2	21.3	8.1	-9.1	-23.7	25.8
greedy.inner_outer2	48.2	49.3	49.2	45.5	40.7	33.8	22.5	7.9	-8.8	-23.4	26.5
gpa.performance	40.2	40.3	43.9	35.9	38.2	32.9	18.0	5.5	-11.7	-26.1	21.7
greedy.performance	38.4	24.6	30.2	29.1	37.2	32.6	16.7	5.9	-12.9	-26.0	17.6
gpa.performance2	40.2	40.3	43.7	36.1	38.3	32.4	18.7	5.8	-11.4	-26.1	21.8
greedy.performance2	38.5	23.8	30.2	29.3	37.2	32.4	17.2	6.1	-12.5	-26.0	17.6
gpa.weight	43.6	38.7	39.3	27.0	32.5	29.8	20.9	8.2	-8.4	-26.6	20.5
greedy.weight	45.1	42.7	43.4	33.8	35.3	31.3	21.5	8.9	-7.5	-25.7	22.9
metis-shem.weight	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0
shem.weight	49.4	49.1	45.6	43.1	37.1	30.9	21.2	9.2	-8.4	-23.5	25.4

Table 6.8: Shows the improvement in coarsest vertex count from Table 6.7 of each sequential variant over KMETIS in percent. The last column shows the mean of all values in the row.

	2	4	8	16	32	64	128	256	512	average
Walshaw	**11.14**	**10.62**	**7.49**	**6.58**	**5.06**	**5.15**	**5.04**	**4.69**	3.93	**6.63**
Roads	8.14	8.45	**15.66**	14.53	**18.81**	17.13	**21.35**	**22.22**	**23.16**	16.61
Matrices	**5.10**	**8.18**	**5.10**	**9.35**	**6.96**	**7.20**	**9.50**	**11.74**	**12.79**	**8.44**
Overall	**9.57**	**9.82**	**8.50**	**8.45**	**7.78**	**7.59**	**8.65**	**8.96**	**8.82**	**8.68**

(a) Improvement of *inner_outer* over KMETIS.

	2	4	8	16	32	64	128	256	512	average
Walshaw	6.58	6.51	1.69	1.83	2.80	4.49	4.87	4.24	**4.12**	4.13
Roads	**10.28**	**11.99**	15.48	**14.66**	17.53	**17.52**	20.22	21.95	22.22	**16.87**
Matrices	3.23	5.74	3.67	6.20	5.62	9.29	8.42	10.91	10.92	7.11
Overall	6.64	7.33	4.43	4.82	5.85	**7.59**	8.16	8.48	8.45	6.86

(b) Improvement of *expansion* over KMETIS.

	2	4	8	16	32	64	128	256	512	average
Walshaw	2.01	-9.08	-5.83	-4.73	-1.30	1.80	3.79	3.59	3.95	-0.64
Roads	9.86	7.51	-3.74	-25.74	-46.86	-43.51	-33.47	-12.89	3.68	-16.13
Matrices	1.94	1.38	1.68	4.60	-0.46	-0.95	0.12	5.17	7.52	2.33
Overall	3.36	-4.38	-4.16	-6.77	-9.08	-6.56	-3.33	1.00	4.52	-2.82

(c) Improvement of *weight* over KMETIS.

	2	4	8	16	32	64	128	256	512	average
Walshaw	-1.87	-6.89	-10.01	-11.18	-6.45	-2.06	1.34	2.31	3.42	-3.49
Roads	-9.24	-20.04	-18.61	-20.22	-20.29	-22.60	-13.45	-7.04	-4.58	-15.12
Matrices	1.23	1.71	-3.86	1.16	-0.52	-0.27	-1.97	1.94	7.54	0.77
Overall	-2.61	-7.68	-10.44	-10.60	-7.83	-5.32	-1.81	0.62	2.74	- 4.77

(d) Improvement of *inner_outer2* over KMETIS.

	2	4	8	16	32	64	128	256	512	avg.
Walshaw	-0.73	-9.78	-12.13	-12.04	-7.59	-0.84	1.40	2.38	3.35	-4.00
Roads	-22.35	-28.46	-52.49	-74.6	-105.6	-102.6	-100.0	-64.70	-37.63	-65.39
Matrices	-1.34	-2.20	-5.06	1.27	-2.23	-0.10	0.07	2.29	5.53	0.20
Overall	-4.60	-11.71	-17.92	-20.61	-23.70	-18.40	-16.47	-9.30	-3.39	-14.01

(e) Improvement of *edge_density* over KMETIS.

Table 6.9: Improvement in initial edge cut of the parallel *inner_outer*, *inner_outer2*, and *edge_density* over KMETIS. The improvements are in percent and given for the graphs from Walshaw's benchmark set, the road networks, matrices from the Florida Sparse Matrix Collection and for all graphs. For each group of matrices and value of *k* and the average, the best value is set in bold letters.

graph	2	4	8	16	32	64	128	256	512
3elt	140	340	550	867	1289	1904	2730	3775	5408
3elt_dual	67	175	296	470	715	1059	1528	2070	2889
whitaker3	155	524	1021	1592	2327	3209	4404	6107	8244
crack	247	582	1050	1640	2399	3397	4613	6343	8609
4elt	197	585	994	1570	2335	3612	5352	7554	10519
whitaker3_dual	81	253	535	866	1267	1828	2440	3437	4565
crack_dual	108	254	473	745	1078	1582	2175	3028	4108
4elt_dual	97	259	518	854	1276	1947	2920	4216	5926
shock.9	166	425	1008	1683	2554	3679	5092	7246	10021
pwt	457	1014	2411	4731	8355	11641	15401	21072	28161
body	485	1195	2045	3374	5075	8144	12643	18609	26773
bracket	1021	4995	13638	23846	32845	42432	58002	77853	100890
tooth	6678	12306	21124	33182	45268	59172	75512	91713	116692
ocean	706	3523	9164	16073	30336	41847	53374	62238	79074
auto	16795	52680	95756	159345	272916	336358	420144	509449	668436
bel	155	372	642	1127	1875	3082	4676	6944	9925
nld	127	268	505	949	1504	2851	4396	6879	10586
deu	305	808	1456	2556	3976	6272	9522	14983	22709
eur	524	1180	1829	3240	5182	8541	13281	20822	32160
af_shell9	12429	30066	61644	121810	212151	350554	495579	690603	906982
af_shell10	35272	76811	147633	243158	394009	627598	914952	1318858	1797782
audikw_1	152032	531676	1349614	2238425	3492831	4923116	6936894	8959215	10740374
cage15	3245868	6058398	9126265	12166202	14966821	17403582	21101320	24536592	27420288

Table 6.10: Initial edge cut of the parallel *inner_outer*.

graph	2	4	8	16	32	64	128	256	512
3elt	162	340	551	880	1304	1908	2729	3775	5408
3elt_dual	66	184	293	477	722	1049	1520	2070	2889
whitaker3	174	636	992	1618	2355	3251	4383	6111	8244
crack	261	821	1330	1771	2406	3450	4859	6403	8609
4elt	212	591	1035	1618	2423	3602	5350	7564	10517
whitaker3_dual	80	311	530	860	1260	1846	2439	3431	4561
crack_dual	110	264	485	751	1079	1593	2222	3037	4114
4elt_dual	98	280	529	856	1320	1945	2900	4170	5906
shock.9	163	524	1057	1706	2572	3758	5124	7298	10006
pwt	462	1113	2600	5200	8386	11902	15771	21080	28138
body	502	1634	2928	4494	5730	8674	12475	18629	26732
bracket	1929	10199	20452	29908	40587	50616	62470	77365	100418
tooth	7884	19061	31484	40895	53813	67405	79409	97555	121115
ocean	724	3710	9496	20689	33515	42342	54008	68031	78935
auto	18321	53538	122634	236088	333908	354589	415467	522529	650540
bel	153	380	806	2067	2980	3948	6245	7541	10263
nld	118	273	644	1380	2998	4010	5605	9036	11561
deu	308	822	1723	3278	8264	13369	17646	22675	29414
eur	528	1165	2206	4296	8060	18216	32931	44169	59147
af_shell9	12211	30440	62564	123514	228654	355465	533431	704442	914833
af_shell10	35145	78153	148286	244911	417443	707815	1068978	1431792	1944055
audikw_1	149601	541404	1394903	2431593	3634503	5091186	7046476	9154836	10890569
cage15	3872984	7771352	10033918	13627303	17395777	21022506	24678734	29716699	31910172

Table 6.11: Initial edge cut of the parallel *weight*.

graph	2	4	8	16	32	64	128	256	512
3elt	161	349	566	857	1331	1931	2724	4005	5702
3elt_dual	72	178	307	478	731	1085	1517	2074	3045
whitaker3	174	613	1051	1625	2353	3311	4506	6118	8674
crack	270	679	1165	1752	2507	3489	4911	6605	9172
4elt	217	603	1054	1639	2460	3723	5427	7648	10545
whitaker3_dual	89	293	561	883	1286	1844	2543	3436	4574
crack_dual	133	308	587	901	1306	1839	2549	3510	4645
4elt_dual	107	287	546	883	1341	2032	2987	4138	5503
shock.9	184	522	1128	1824	2686	3871	5501	7557	10239
pwt	464	1093	2531	5004	8601	12015	16240	21785	28768
body	534	1406	2419	4016	5862	8884	13169	18902	27900
bracket	1286	5828	14652	25219	35388	46998	62654	81834	104571
tooth	7032	13271	22616	35393	48762	62248	78385	98231	121540
ocean	912	4178	10839	20582	34002	45656	57876	72543	88935
auto	19135	54809	94482	160075	247859	359024	469033	582302	714568
bel	171	413	767	1355	2279	3746	5558	7937	11272
nld	133	309	612	1095	1826	3298	5394	8072	12030
deu	349	915	1735	3006	4791	7502	11508	18568	28485
eur	562	1170	2095	3724	6718	10787	20050	36060	62348
af_shell9	12870	32226	64015	124698	211676	347243	509667	710094	973484
af_shell10	34640	80545	156258	251733	402148	605167	937721	1373148	1882430
audikw_1	156546	538794	1281520	2315530	3485141	5120455	7256910	9613267	11728006
cage15	3859702	7580256	10927762	16981188	20293811	24716559	29463526	36864941	39982279

Table 6.12: Initial edge cut of KMETIS.

graph	2	4	8	16	32	64	128	256	512
3elt	12.80	2.57	2.86	-1.17	3.10	1.38	-0.21	5.75	5.16
3elt_dual	6.77	1.91	3.50	1.60	2.29	2.40	-0.72	0.21	5.13
whitaker3	11.41	14.46	2.87	2.03	1.11	3.08	2.27	0.18	4.95
crack	8.25	14.37	9.85	6.36	4.31	2.64	6.07	3.97	6.13
4elt	9.03	3.03	5.67	4.20	5.11	2.99	1.38	1.22	0.25
whitaker3_dual	9.38	13.73	4.56	1.96	1.47	0.86	4.02	-0.03	0.19
crack_dual	19.05	17.60	19.50	17.28	17.43	13.99	14.70	13.73	11.57
4elt_dual	9.19	9.71	5.13	3.30	4.88	4.19	2.26	-1.87	-7.68
shock.9	10.07	18.65	10.61	7.69	4.90	4.96	7.44	4.11	2.12
pwt	1.54	7.18	4.76	5.45	2.86	3.11	5.17	3.27	2.11
body	9.19	15.01	15.47	15.98	13.43	8.34	3.99	1.55	4.04
bracket	20.58	14.30	6.91	5.44	7.19	9.72	7.43	4.87	3.52
tooth	5.05	7.27	6.60	6.25	7.16	4.94	3.67	6.64	3.99
ocean	22.58	15.67	15.45	21.91	10.78	8.34	7.78	14.20	11.09
auto	12.23	3.88	-1.35	0.46	-10.11	6.31	10.42	12.51	6.46
bel	9.30	9.86	16.36	16.83	17.69	17.73	15.87	12.52	11.95
nld	4.00	13.13	17.56	13.36	17.65	13.55	18.51	14.78	12.00
deu	12.51	11.68	16.07	14.95	17.01	16.40	17.26	19.31	20.28
eur	6.77	-0.87	12.67	12.99	22.87	20.82	33.76	42.26	48.42
af_shell9	3.43	6.70	3.70	2.32	-0.22	-0.95	2.76	2.74	6.83
af_shell10	-1.82	4.64	5.52	3.41	2.02	-3.71	2.43	3.95	4.50
audikw_1	2.88	1.32	-5.31	3.33	-0.22	3.85	4.41	6.80	8.42
cage15	15.90	20.08	16.49	28.35	26.25	29.59	28.38	33.44	31.42
avg. imp.	9.57	9.82	8.50	8.45	7.78	7.59	8.65	8.96	8.82

Table 6.13: Improvement of initial edge cut of the parallel *inner_outer* compared to KMETIS in percent.

graph	2	4	8	16	32	64	128	256	512
3elt	-0.65	2.47	2.73	-2.74	2.02	1.20	-0.17	5.75	5.16
3elt_dual	8.30	-3.15	4.45	0.21	1.26	3.27	-0.20	0.19	5.13
whitaker3	0.43	-3.79	5.57	0.43	-0.09	1.79	2.73	0.11	4.95
crack	3.34	-20.90	-14.19	-1.09	4.06	1.13	1.05	3.06	6.13
4elt	2.05	2.01	1.77	1.33	1.53	3.27	1.42	1.11	0.26
whitaker3_dual	10.62	-6.17	5.39	2.66	2.05	-0.09	4.08	0.16	0.29
crack_dual	17.70	14.44	17.45	16.64	17.32	13.36	12.84	13.48	11.43
4elt_dual	7.69	2.49	3.09	3.06	1.63	4.30	2.91	-0.78	-7.32
shock.9	11.49	-0.40	6.27	6.43	4.25	2.92	6.86	3.42	2.27
pwt	0.61	-1.84	-2.73	-3.91	2.49	0.95	2.89	3.23	2.19
body	5.86	-16.22	-21.04	-11.91	2.27	2.37	5.27	1.45	4.19
bracket	-50.01	-74.98	-39.59	-18.59	-14.69	-7.70	0.29	5.46	3.97
tooth	-12.10	-43.63	-39.21	-15.54	-10.36	-8.28	-1.31	0.69	0.35
ocean	20.58	11.20	12.39	-0.52	1.43	7.26	6.68	6.22	11.24
auto	4.25	2.32	-29.80	-47.49	-34.72	1.24	11.42	10.26	8.96
bel	10.61	7.98	-5.10	-52.56	-30.80	-5.39	-12.37	4.99	8.95
nld	11.16	11.49	-5.22	-26.00	-64.15	-21.59	-3.92	-11.95	3.90
deu	11.76	10.15	0.69	-9.06	-72.51	-78.20	-53.33	-22.11	-3.26
eur	5.91	0.40	-5.31	-15.36	-19.98	-68.86	-64.24	-22.49	5.13
af_shell9	5.12	5.54	2.27	0.95	-8.02	-2.37	-4.66	0.80	6.02
af_shell10	-1.46	2.97	5.10	2.71	-3.80	-16.96	-14.00	-4.27	-3.27
audikw_1	4.44	-0.48	-8.85	-5.01	-4.29	0.57	2.90	4.77	7.14
cage15	-0.34	-2.52	8.18	19.75	14.28	14.95	16.24	19.39	20.19
avg. imp.	3.36	-4.38	-4.16	-6.77	-9.08	-6.56	-3.33	1.00	4.52

Table 6.14: Improvement of initial edge cut of the parallel *weight* compared to KMETIS in percent.

graph	2	4	8	16	32	64	128	256	512
3elt	1.004	1.016	1.026	1.031	1.028	1.034	1.059	1.053	1.100
3elt_dual	1.005	1.012	1.024	1.031	1.027	1.032	1.029	1.031	1.061
whitaker3	1.004	1.012	1.022	1.028	1.032	1.030	1.036	1.038	1.050
crack	1.005	1.011	1.023	1.029	1.033	1.041	1.039	1.056	1.100
4elt	1.005	1.010	1.021	1.029	1.031	1.038	1.046	1.055	1.065
whitaker3_dual	1.004	1.010	1.022	1.029	1.035	1.037	1.035	1.042	1.049
crack_dual	1.003	1.009	1.022	1.029	1.032	1.036	1.032	1.033	1.046
4elt_dual	1.005	1.008	1.017	1.029	1.030	1.038	1.042	1.041	1.038
shock.9	1.004	1.010	1.017	1.030	1.035	1.040	1.040	1.044	1.036
pwt	1.005	1.011	1.017	1.031	1.033	1.040	1.037	1.046	1.044
body	1.006	1.010	1.018	1.034	1.037	1.057	1.075	1.120	1.089
bracket	1.005	1.013	1.021	1.026	1.036	1.034	1.043	1.051	1.058
tooth	1.007	1.014	1.019	1.023	1.032	1.042	1.053	1.053	1.061
ocean	1.010	1.016	1.021	1.026	1.033	1.045	1.048	1.051	1.055
auto	1.009	1.016	1.024	1.029	1.027	1.045	1.047	1.059	1.067
bel	1.005	1.011	1.016	1.023	1.028	1.036	1.049	1.058	1.053
nld	1.006	1.013	1.021	1.026	1.029	1.037	1.051	1.056	1.066
deu	1.006	1.014	1.020	1.027	1.031	1.038	1.045	1.047	1.059
eur	1.008	1.014	1.022	1.029	1.033	1.040	1.046	1.052	1.061
af_shell9	1.005	1.010	1.018	1.022	1.029	1.050	1.056	1.060	1.054
af_shell10	1.006	1.011	1.018	1.023	1.031	1.035	1.044	1.067	1.072
audikw_1	1.008	1.014	1.023	1.029	1.037	1.040	1.051	1.051	1.064
cage15	1.008	1.017	1.024	1.034	1.042	1.051	1.056	1.066	1.075

Table 6.15: Initial balance of the parallel *inner_outer*.

graph	2	4	8	16	32	64	128	256	512
3elt	1.004	1.017	1.025	1.027	1.027	1.036	1.054	1.053	1.100
3elt_dual	1.004	1.014	1.025	1.028	1.029	1.030	1.032	1.031	1.061
whitaker3	1.004	1.014	1.024	1.029	1.034	1.029	1.038	1.038	1.050
crack	1.005	1.253	1.361	1.037	1.034	1.041	1.048	1.056	1.100
4elt	1.005	1.009	1.022	1.034	1.035	1.037	1.044	1.050	1.063
whitaker3_dual	1.004	1.009	1.021	1.028	1.032	1.040	1.032	1.038	1.049
crack_dual	1.005	1.009	1.024	1.025	1.032	1.035	1.035	1.035	1.049
4elt_dual	1.005	1.010	1.018	1.035	1.037	1.033	1.041	1.043	1.039
shock.9	1.005	1.009	1.016	1.030	1.037	1.043	1.041	1.040	1.039
pwt	1.005	1.011	1.019	1.038	1.033	1.042	1.042	1.047	1.042
body	1.007	1.012	1.153	1.632	1.090	1.080	1.087	1.072	1.099
bracket	1.007	1.020	1.129	1.185	1.463	1.073	1.054	1.051	1.059
tooth	1.008	1.006	1.187	1.102	1.181	1.176	1.066	1.058	1.068
ocean	1.009	1.015	1.023	1.026	1.058	1.050	1.050	1.058	1.057
auto	1.009	1.019	1.003	1.020	1.062	1.010	1.053	1.059	1.070
bel	1.005	1.013	1.020	1.027	1.174	1.356	1.918	1.096	1.066
nld	1.007	1.013	1.020	1.028	1.076	1.374	1.499	1.843	1.136
deu	1.007	1.014	1.021	1.031	1.029	1.087	1.204	1.321	1.383
eur	1.008	1.014	1.019	1.030	1.039	1.057	1.060	1.197	1.358
af_shell9	1.007	1.011	1.016	1.023	1.028	1.049	1.106	1.059	1.055
af_shell10	1.005	1.010	1.016	1.023	1.030	1.021	1.261	1.509	1.121
audikw_1	1.009	1.011	1.020	1.028	1.037	1.045	1.056	1.048	1.059
cage15	1.009	1.018	1.025	1.033	1.042	1.047	1.056	1.066	1.075

Table 6.16: Initial balance of the parallel *weight*.

graph	2	4	8	16	32	64	128	256	512
3elt	1.005	1.019	1.022	1.030	1.033	1.035	1.058	1.104	1.193
3elt_dual	1.005	1.012	1.024	1.029	1.034	1.037	1.039	1.052	1.098
whitaker3	1.005	1.013	1.023	1.028	1.035	1.041	1.041	1.061	1.102
crack	1.006	1.010	1.022	1.030	1.036	1.041	1.049	1.066	1.103
4elt	1.005	1.011	1.025	1.031	1.036	1.045	1.046	1.056	1.084
whitaker3_dual	1.003	1.010	1.027	1.026	1.032	1.038	1.038	1.045	1.061
crack_dual	1.005	1.010	1.021	1.029	1.034	1.035	1.036	1.039	1.063
4elt_dual	1.005	1.010	1.017	1.031	1.036	1.042	1.046	1.044	1.050
shock.9	1.005	1.008	1.017	1.027	1.035	1.042	1.047	1.049	1.047
pwt	1.005	1.011	1.018	1.030	1.032	1.042	1.051	1.054	1.055
body	1.007	1.012	1.019	1.036	1.048	1.082	1.083	1.104	1.137
bracket	1.007	1.014	1.020	1.026	1.038	1.043	1.050	1.059	1.066
tooth	1.008	1.013	1.019	1.025	1.040	1.045	1.052	1.060	1.068
ocean	1.009	1.015	1.019	1.026	1.037	1.046	1.052	1.062	1.069
auto	1.008	1.017	1.022	1.028	1.036	1.043	1.053	1.058	1.073
bel	1.006	1.011	1.019	1.022	1.029	1.037	1.052	1.056	1.060
nld	1.007	1.013	1.021	1.026	1.032	1.038	1.048	1.062	1.065
deu	1.007	1.013	1.020	1.026	1.035	1.041	1.044	1.050	1.061
eur	1.006	1.015	1.021	1.029	1.034	1.041	1.045	1.045	1.010
af_shell9	1.006	1.012	1.017	1.020	1.027	1.036	1.049	1.057	1.062
af_shell10	1.005	1.010	1.017	1.022	1.030	1.033	1.042	1.054	1.062
audikw_1	1.006	1.013	1.021	1.029	1.034	1.038	1.049	1.063	1.070
cage15	1.009	1.020	1.026	1.039	1.045	1.054	1.065	1.069	1.078

Table 6.17: Initial balance of KMETIS.

graph	2	4	8	16	32	64	128	256	512
3elt	0.08	0.22	-0.39	-0.12	0.44	0.14	-0.17	4.63	7.81
3elt_dual	0.02	0.08	0.01	-0.20	0.69	0.51	0.97	2.08	3.36
whitaker3	0.12	0.13	0.14	0.05	0.24	1.10	0.44	2.08	4.75
crack	0.12	-0.11	-0.08	0.11	0.29	-0.03	0.95	0.94	0.23
4elt	-0.05	0.11	0.38	0.15	0.47	0.73	-0.01	0.14	1.82
whitaker3_dual	-0.07	-0.02	0.51	-0.30	-0.28	0.10	0.31	0.24	1.12
crack_dual	0.13	0.13	-0.05	-0.04	0.26	-0.09	0.35	0.59	1.54
4elt_dual	0.05	0.17	0.03	0.18	0.63	0.39	0.40	0.28	1.15
shock.9	0.04	-0.19	0.02	-0.26	0.07	0.19	0.70	0.49	1.05
pwt	-0.06	0.02	0.11	-0.07	-0.13	0.18	1.27	0.81	1.00
body	0.09	0.21	0.03	0.17	1.05	2.35	0.70	-1.45	4.22
bracket	0.16	0.10	-0.11	-0.03	0.21	0.88	0.75	0.76	0.74
tooth	0.09	-0.10	0.01	0.18	0.73	0.26	-0.04	0.69	0.68
ocean	-0.06	-0.09	-0.26	-0.03	0.44	0.17	0.36	1.02	1.30
auto	-0.10	0.11	-0.20	-0.12	0.84	-0.23	0.52	-0.05	0.52
bel	0.05	0.07	0.34	-0.06	0.03	0.13	0.20	-0.14	0.67
nld	0.07	-0.05	-0.05	-0.05	0.28	0.10	-0.28	0.63	-0.09
deu	0.11	-0.09	0.01	-0.08	0.33	0.30	-0.07	0.26	0.14
eur	-0.15	0.11	-0.08	-0.01	0.04	0.08	-0.05	-0.68	-5.10
af_shell9	0.10	0.13	-0.11	-0.22	-0.14	-1.34	-0.68	-0.29	0.72
af_shell10	-0.14	-0.09	-0.04	-0.14	-0.06	-0.17	-0.18	-1.27	-0.93
audikw_1	-0.22	-0.05	-0.18	0.06	-0.31	-0.19	-0.14	1.15	0.57
cage15	0.08	0.29	0.20	0.40	0.30	0.38	0.88	0.30	0.26
avg. imp.	0.02	0.05	0.01	-0.02	0.28	0.26	0.31	0.58	1.20

Table 6.18: Improvement of initial balance of the parallel *inner_outer* compared to KMETIS in percent.

graph	2	4	8	16	32	64	128	256	512
3elt	0.07	0.15	-0.22	0.21	0.57	-0.05	0.34	4.63	7.81
3elt_dual	0.13	-0.11	-0.06	0.05	0.47	0.68	0.70	2.08	3.36
whitaker3	0.13	-0.12	-0.06	-0.10	0.06	1.13	0.26	2.08	4.75
crack	0.06	-24.04	-33.18	-0.64	0.26	-0.03	0.12	0.94	0.23
4elt	-0.04	0.15	0.29	-0.28	0.11	0.77	0.18	0.61	1.97
whitaker3_dual	-0.06	0.09	0.57	-0.16	0.00	-0.17	0.60	0.63	1.12
crack_dual	-0.01	0.13	-0.26	0.45	0.22	-0.01	0.04	0.35	1.30
4elt_dual	0.06	0.04	-0.11	-0.36	-0.10	0.78	0.44	0.08	1.07
shock.9	-0.06	-0.08	0.11	-0.20	-0.13	-0.04	0.63	0.89	0.79
pwt	-0.01	-0.02	-0.06	-0.74	-0.11	0.03	0.79	0.71	1.26
body	0.01	0.01	-13.22	-57.59	-4.09	0.22	-0.39	2.93	3.28
bracket	-0.01	-0.59	-10.76	-15.50	-40.95	-2.84	-0.36	0.74	0.70
tooth	-0.02	0.65	-16.43	-7.55	-13.60	-12.55	-1.32	0.26	0.04
ocean	-0.05	-0.04	-0.38	0.00	-2.02	-0.35	0.25	0.33	1.10
auto	-0.09	-0.21	1.86	0.71	-2.56	3.18	-0.06	-0.11	0.29
bel	0.10	-0.11	-0.07	-0.43	-14.14	-30.66	-82.37	-3.80	-0.54
nld	-0.03	0.02	0.06	-0.23	-4.35	-32.38	-43.03	-73.45	-6.75
deu	-0.01	-0.15	-0.14	-0.50	0.58	-4.49	-15.26	-25.86	-30.43
eur	-0.18	0.14	0.15	-0.07	-0.50	-1.56	-1.38	-14.54	-34.50
af_shell9	-0.08	0.05	0.09	-0.32	-0.07	-1.19	-5.45	-0.21	0.70
af_shell10	-0.06	0.05	0.13	-0.11	0.01	1.20	-20.99	-43.23	-5.55
audikw_1	-0.31	0.25	0.10	0.18	-0.35	-0.73	-0.58	1.45	1.02
cage15	-0.04	0.19	0.09	0.55	0.34	0.74	0.87	0.36	0.35
avg. imp.	-0.02	-1.02	-3.11	-3.59	-3.49	-3.41	-7.22	-6.18	-2.03

Table 6.19: Improvement of initial balance of the parallel *weight* compared to KMETIS in percent.

graph	2	4	8	16	32	64	128	256	512
3elt	106	73	138	278	712	1335	2481	4720	4720
3elt_dual	202	110	142	270	682	1363	2544	4792	9000
whitaker3	214	117	147	281	555	1386	2746	5149	9800
crack	234	123	160	298	553	1105	2902	5395	10240
4elt	349	189	142	324	673	1257	2343	4396	8216
whitaker3_dual	421	224	145	284	549	1137	2871	5360	10132
crack_dual	443	238	146	285	559	1095	2741	5701	10559
4elt_dual	671	360	227	289	697	1304	2447	4575	8551
shock.9	785	413	270	285	543	1158	2860	5401	10192
pwt	819	458	300	307	565	1234	2954	5473	10230
body	1046	481	371	376	668	1248	2305	4439	9215
bracket	1376	767	508	366	630	1554	2821	5238	9584
tooth	1716	952	631	482	671	1201	2336	6350	11689
ocean	3039	1642	1095	927	602	1084	2231	6126	11474
auto	10088	8516	6465	5940	3354	3545	5739	13346	13302
bel	10353	5305	3831	2760	2462	2039	2798	5150	13315
nld	20182	10391	7587	5449	4800	3526	3475	5754	10963
deu	98348	50577	35703	26058	21919	16358	15314	14755	14936
eur	409309	210191	147451	107893	89339	68049	62325	58219	55804
af_shell9	12093	6271	4305	2679	2886	2119	3816	6496	16526
af_shell10	36216	18608	11729	7980	6252	5568	8485	8017	15820
audikw_1	21822	8485	6471	7280	5329	5934	6164	9017	20010
cage15	107677	53834	33145	21467	25213	27600	19200	20601	31977

Table 6.20: Coarsest vertex count of the parallel *inner_outer*.

graph	2	4	8	16	32	64	128	256	512
3elt	99	84	152	271	701	1353	2485	4720	4720
3elt_dual	195	97	158	280	660	1417	2563	4809	9000
whitaker3	220	111	168	301	558	1308	2774	5142	9800
crack	242	201	275	457	801	1251	2299	5702	10240
4elt	325	185	185	299	691	1347	2429	4470	8227
whitaker3_dual	459	199	173	304	572	1079	2953	5383	10146
crack_dual	491	210	168	315	587	1123	2461	5739	10563
4elt_dual	643	325	253	308	592	1441	2593	4718	8617
shock.9	881	380	305	348	595	1094	2970	5566	10321
pwt	777	466	332	376	724	1185	2746	5659	10366
body	1016	626	477	697	946	1550	2937	4843	9283
bracket	1611	1228	1856	2109	2407	2623	3980	7649	11831
tooth	2274	1587	2535	2716	3437	2947	4428	6618	11702
ocean	2941	1651	1154	976	994	1734	2774	4867	12514
auto	9690	12865	8146	6958	8400	10632	14570	20817	31037
bel	11470	6232	5013	5076	6031	7949	7477	10993	17699
nld	22841	12335	9644	8509	9550	10625	14740	13662	19527
deu	108882	59065	44206	39582	39374	44124	54063	71341	84055
eur	458008	248803	179500	152493	138771	136803	148044	175079	221180
af_shell9	12127	5603	3947	3454	3198	3266	4232	7837	17781
af_shell10	36135	16749	11977	9712	7357	7235	10111	8739	15812
audikw_1	22887	9517	7840	5771	6423	6768	7148	9951	21688
cage15	116368	45308	47617	37292	30437	26176	27218	23377	30165

Table 6.21: Coarsest vertex count of the parallel *weight*.

graph	2	4	8	16	32	64	128	256	512
3elt	103	77	154	310	620	1319	2473	2473	2473
3elt_dual	201	104	152	297	594	1191	2591	4799	4799
whitaker3	214	110	154	307	624	1288	2689	5091	5091
crack	243	114	150	288	597	1151	2169	4082	6767
4elt	340	177	151	302	581	1122	2403	4384	8145
whitaker3_dual	424	218	142	310	623	1260	2545	5490	10255
crack_dual	435	223	161	306	612	1274	2647	5512	10374
4elt_dual	668	345	222	303	599	1191	2377	6063	14176
shock.9	783	404	285	310	612	1213	2393	4792	10094
pwt	870	402	278	295	598	1214	2392	4773	9991
body	1090	506	341	286	599	1171	2517	5333	10651
bracket	1278	663	512	347	602	1257	2440	4697	9030
tooth	1584	821	644	427	631	1281	2575	5151	10457
ocean	2968	1531	1151	794	696	1247	2478	4827	9374
auto	8867	4481	3750	2313	2113	1880	2312	4463	8630
bel	10710	5529	3606	2636	2185	1754	2367	4670	9735
nld	20934	10778	7046	5165	4256	3415	2965	4784	10027
deu	102095	52657	34204	25039	20638	16547	14396	12902	11924
eur	483086	279405	209298	173770	160600	154191	149516	145876	422215
af_shell9	12019	5571	4047	2891	2321	1983	2347	4564	9723
af_shell10	35945	16636	12079	8622	6921	5922	4921	4747	9409
audikw_1	26302	12752	7839	6161	3981	3819	3634	3981	7839
cage15	134506	67360	42091	34142	21084	21084	20688	17355	10557

Table 6.22: Coarsest vertex count of KMETIS.

graph	2	4	8	16	32	64	128	256	512
3elt	-2.28	5.39	10.01	10.45	-14.77	-1.17	-0.32	-90.88	-90.88
3elt_dual	-0.52	-5.33	6.69	9.28	-14.99	-14.44	1.79	0.14	-87.55
whitaker3	0.23	-6.51	4.23	8.46	11.11	-7.62	-2.11	-1.14	-92.50
crack	3.92	-8.29	-6.48	-3.35	7.25	3.99	-33.79	-32.16	-51.33
4elt	-2.83	-6.65	6.44	-7.31	-15.98	-12.01	2.49	-0.27	-0.86
whitaker3_dual	0.70	-2.82	-2.12	8.34	11.93	9.78	-12.79	2.36	1.20
crack_dual	-1.98	-6.93	9.39	6.88	8.74	14.02	-3.55	-3.43	-1.78
4elt_dual	-0.48	-4.32	-2.23	4.43	-16.25	-9.48	-2.96	24.54	39.68
shock.9	-0.24	-2.28	5.30	8.06	11.25	4.56	-19.53	-12.70	-0.96
pwt	5.85	-13.91	-7.88	-4.08	5.50	-1.60	-23.45	-14.68	-2.39
body	4.11	4.83	-8.95	-31.13	-11.66	-6.53	8.43	16.77	13.48
bracket	-7.67	-15.66	0.80	-5.61	-4.62	-23.65	-15.65	-11.50	-6.13
tooth	-8.38	-15.97	2.02	-12.74	-6.37	6.23	9.29	-23.29	-11.78
ocean	-2.41	-7.25	4.83	-16.68	13.48	13.08	9.96	-26.92	-22.40
auto	-13.77	-90.06	-72.38	-156.85	-58.70	-88.59	-148.25	-199.05	-54.14
bel	3.34	4.04	-6.23	-4.68	-12.70	-16.25	-18.20	-10.27	-36.77
nld	3.59	3.59	-7.69	-5.49	-12.79	-3.24	-17.20	-20.29	-9.33
deu	3.67	3.95	-4.38	-4.07	-6.21	1.14	-6.38	-14.37	-25.26
eur	15.27	24.77	29.55	37.91	44.37	55.87	58.32	60.09	86.78
af_shell9	-0.62	-12.56	-6.37	7.34	-24.33	-6.87	-62.61	-42.34	-69.97
af_shell10	-0.75	-11.85	2.90	7.45	9.66	5.98	-72.43	-68.90	-68.14
audikw_1	17.03	33.46	17.45	-18.16	-33.86	-55.36	-69.63	-126.50	-155.27
cage15	19.95	20.08	21.25	37.12	-19.58	-30.91	7.19	-18.71	-202.91
avg. imp.	1.55	-4.79	-0.17	-5.41	-5.63	-7.09	-17.89	-26.67	-36.92

Table 6.23: Improvement of coarsest vertex count (smaller vertex count is better) of the parallel *inner_outer* compared to KMETIS in percent.

graph	2	4	8	16	32	64	128	256	512
3elt	4.12	-8.63	1.46	12.77	-13.09	-2.54	-0.52	-90.88	-90.88
3elt_dual	3.04	7.25	-3.38	5.82	-11.20	-18.95	1.09	-0.22	-87.55
whitaker3	-2.59	-1.46	-9.64	1.94	10.57	-1.53	-3.16	-1.00	-92.50
crack	0.41	-76.32	-83.08	-58.75	-34.20	-8.72	-5.99	-39.68	-51.33
4elt	4.23	-4.09	-22.29	1.03	-19.00	-20.01	-1.09	-1.97	-1.01
whitaker3_dual	-8.28	8.79	-22.07	1.81	8.27	14.38	-16.03	1.95	1.06
crack_dual	-12.90	5.63	-3.84	-2.97	4.24	11.87	7.03	-4.12	-1.82
4elt_dual	3.73	5.94	-14.14	-1.65	1.24	-21.00	-9.09	22.17	39.21
shock.9	-12.55	5.96	-7.07	-12.12	2.78	9.76	-24.11	-16.15	-2.25
pwt	10.69	-15.91	-19.44	-27.44	-21.04	2.42	-14.76	-18.57	-3.75
body	6.81	-23.87	-39.88	-143.31	-57.96	-32.31	-16.68	9.19	12.84
bracket	-26.03	-85.24	-262.79	-508.50	-299.80	-108.68	-63.16	-62.85	-31.02
tooth	-43.61	-93.22	-293.40	-535.28	-444.98	-130.05	-71.96	-28.49	-11.91
ocean	0.89	-7.81	-0.23	-22.88	-42.85	-39.05	-11.91	-0.83	-33.50
auto	-9.28	-187.12	-117.20	-200.87	-297.46	-465.62	-530.26	-366.45	-259.64
bel	-7.10	-12.73	-39.00	-92.52	-176.03	-353.28	-215.82	-135.39	-81.80
nld	-9.11	-14.45	-36.89	-64.73	-124.40	-211.13	-397.08	-185.58	-94.74
deu	-6.65	-12.17	-29.24	-58.08	-90.79	-166.67	-275.56	-452.97	-604.90
eur	5.19	10.95	14.24	12.24	13.59	11.28	0.98	-20.02	47.61
af_shell9	-0.90	-0.57	2.48	-19.47	-37.77	-64.71	-80.34	-71.72	-82.87
af_shell10	-0.53	-0.68	0.84	-12.64	-6.30	-22.17	-105.48	-84.11	-68.05
audikw_1	12.98	25.37	-0.01	6.33	-61.34	-77.20	-96.71	-149.97	-176.67
cage15	13.48	32.74	-13.13	-9.23	-44.36	-24.15	-31.57	-34.70	-185.75
avg. imp.	-3.22	-19.20	-43.38	-75.15	-75.73	-74.70	-85.31	-75.32	-80.92

Table 6.24: Improvement of coarsest vertex count of the parallel *weight* compared to KMETIS in percent.
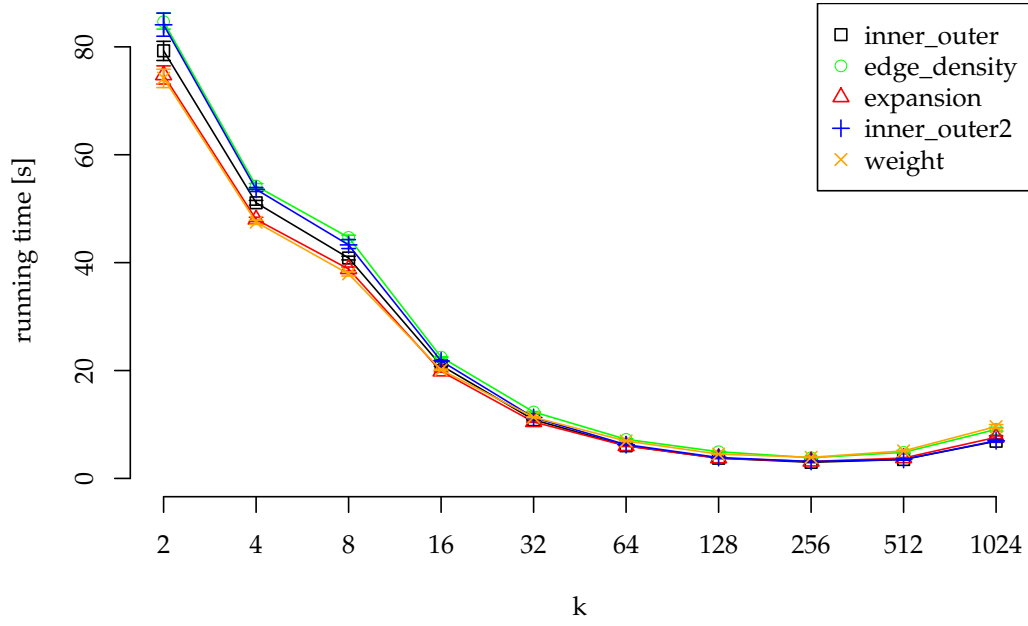
(a) Running time of coarsening with initial partitioning.
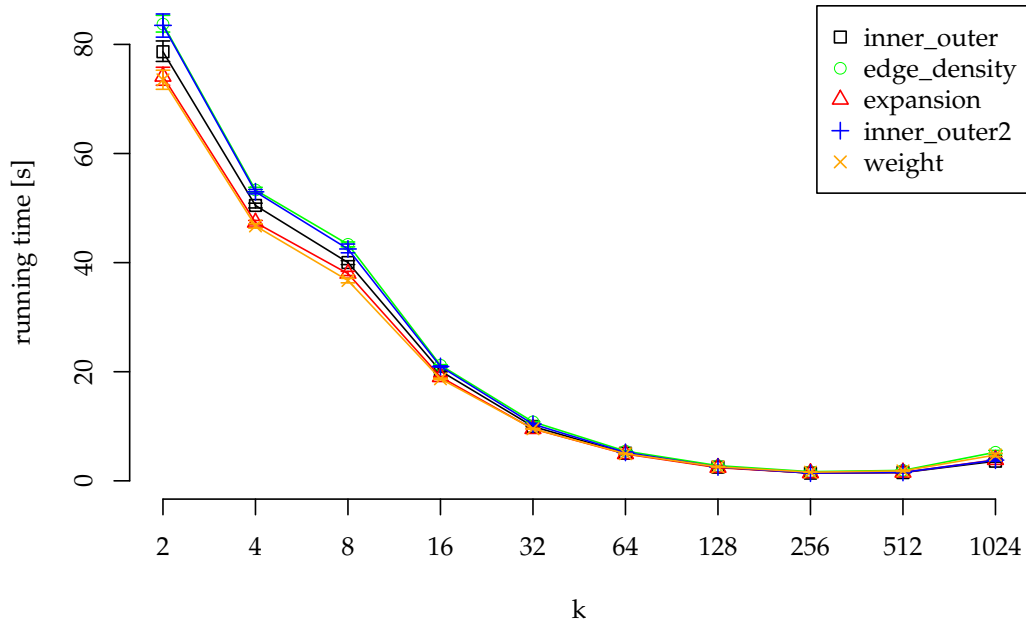
(b) Running time of coarsening without initial partitioning.

Figure 6.5: Running time of coarsening using various rating variants on *eur*.
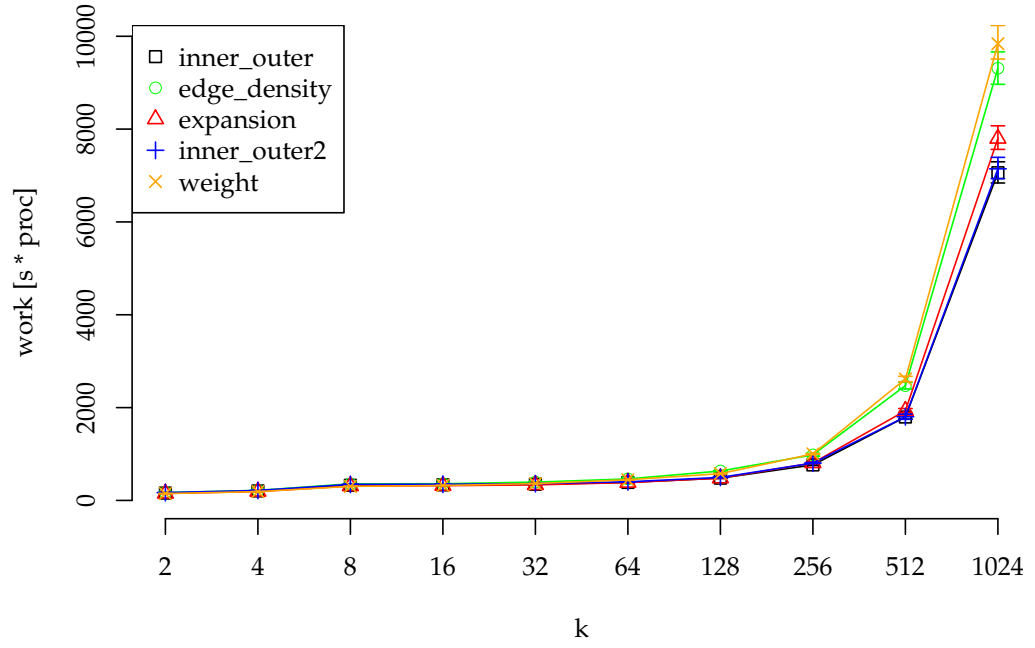
(a) Work of coarsening with initial partitioning.
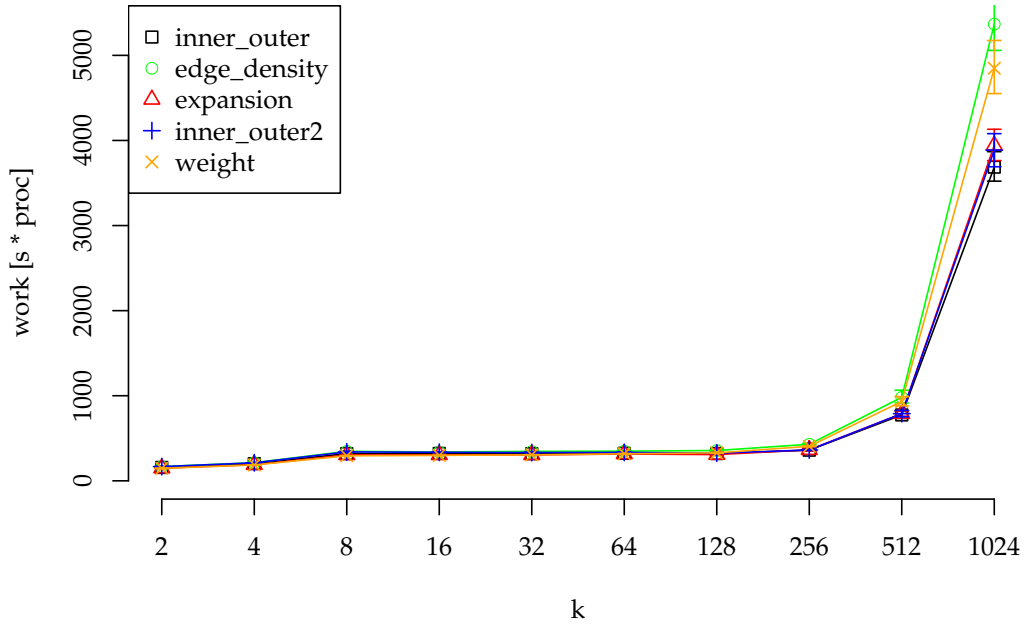
(b) Work of coarsening without initial partitioning.

Figure 6.6: Work of coarsening using various rating variants on *eur*.
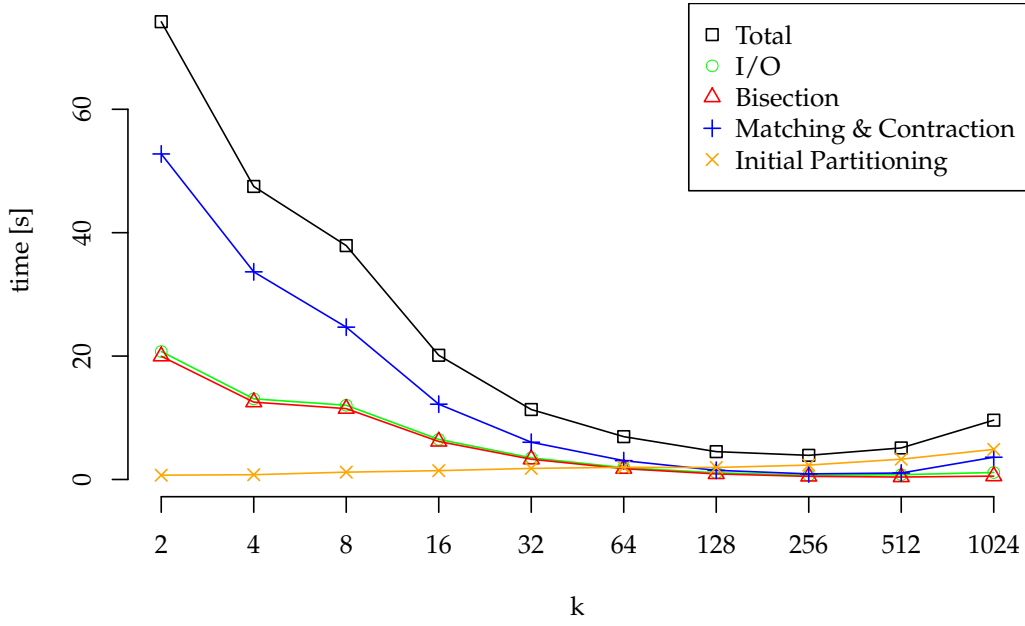
(a) Running time of the coarsening phase parts for *weight*.
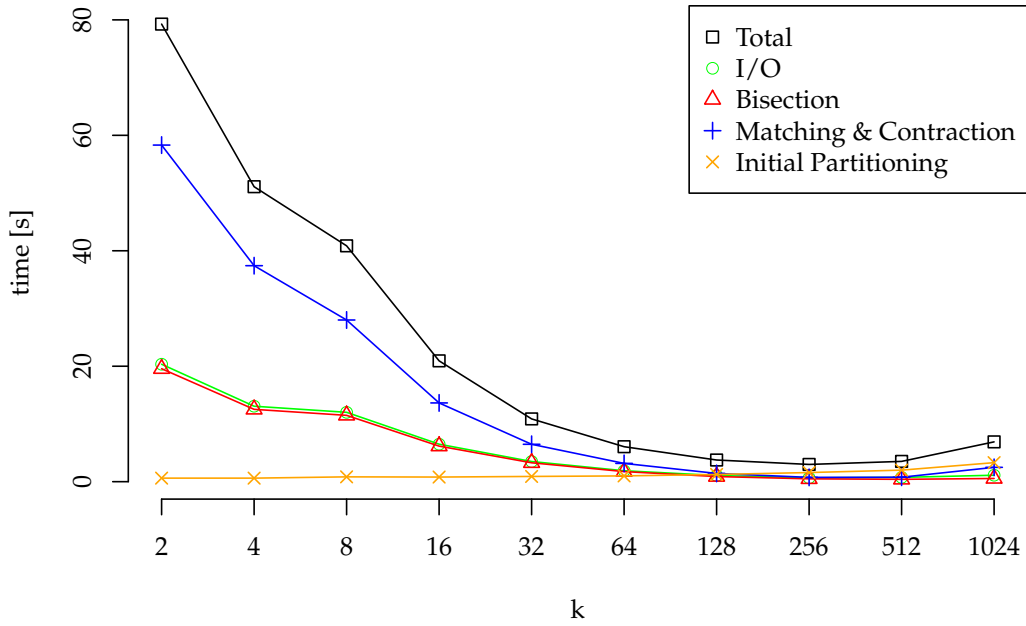
(b) Running time of the coarsening phase parts for *inner_outer*.

Figure 6.7: Running time of the parts of the coarsening phase *eur*.
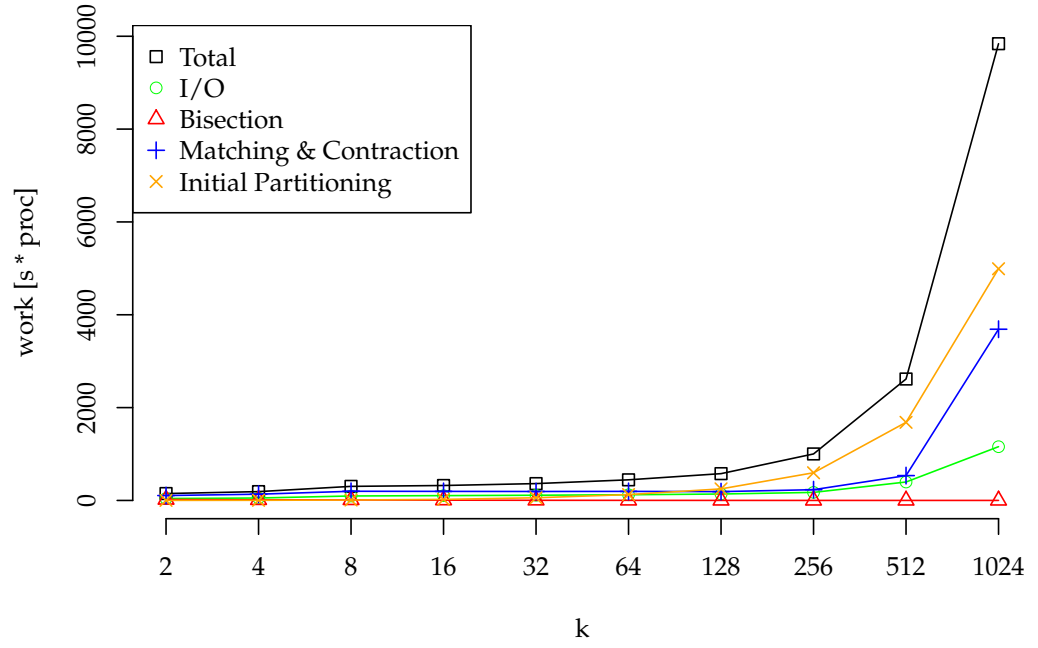
(a) Work of the coarsening phase parts for *weight*.
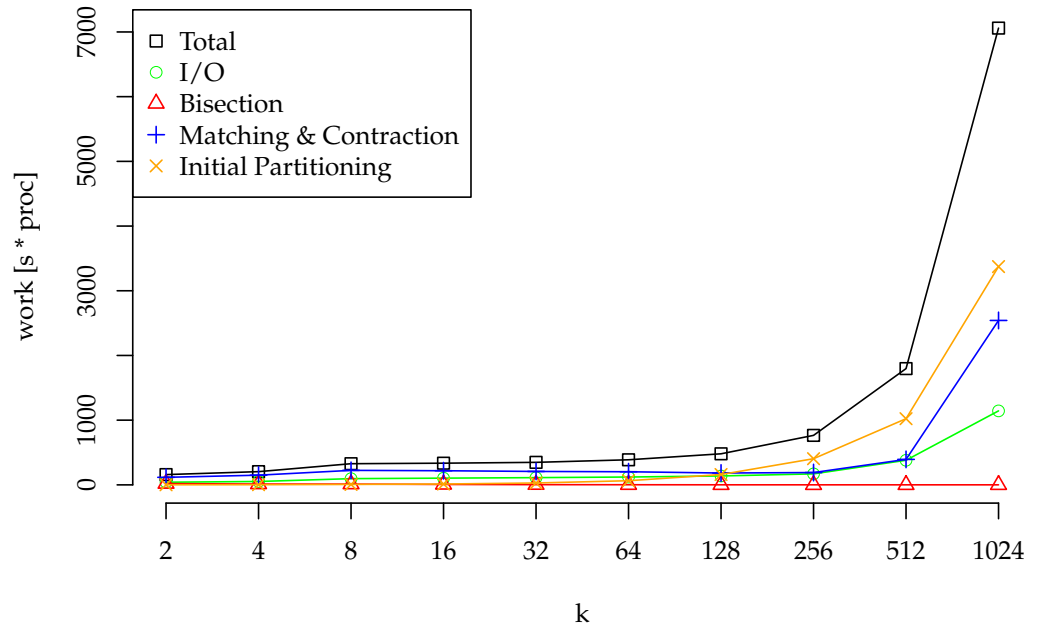
(b) Work of the coarsening phase parts for *inner_outer*.

Figure 6.8: Work of the parts of the coarsening phase *eur*.
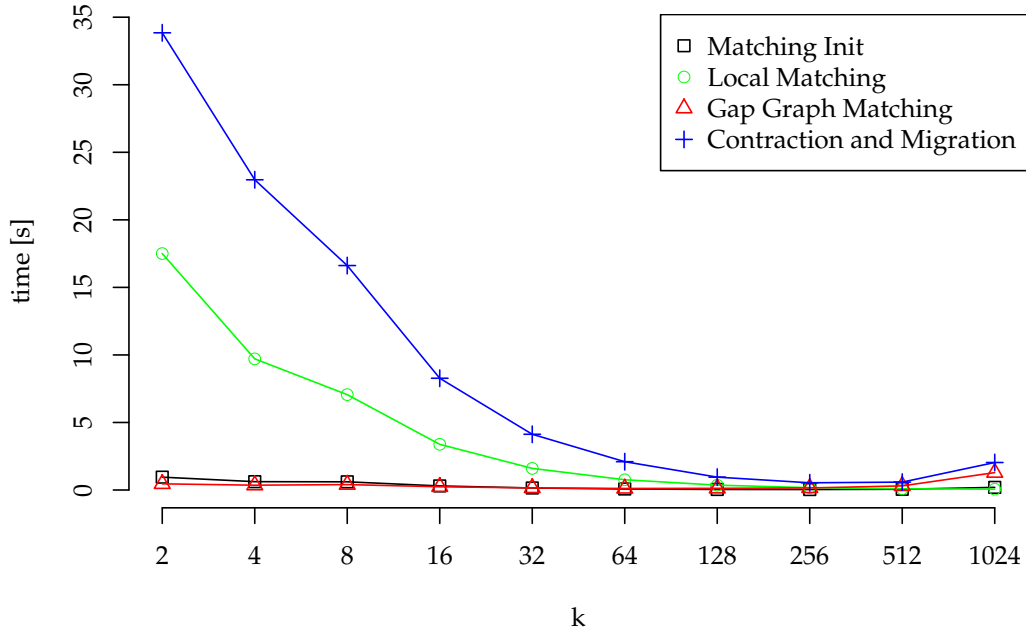
(a) Running time for *weight*.
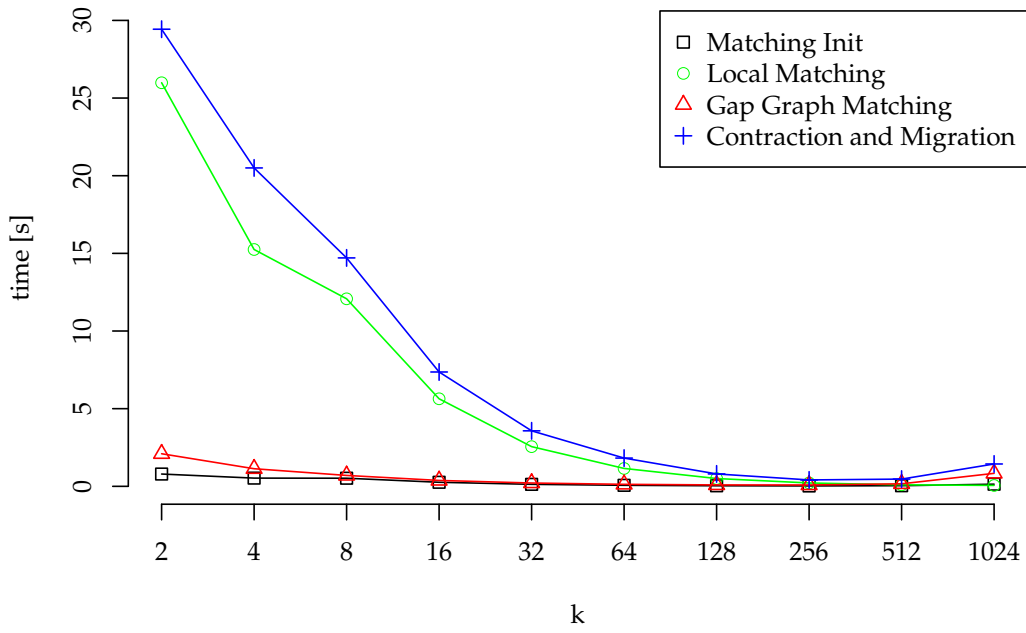
(b) Running time for *inner_outer*.

Figure 6.9: Running time of the matching algorithm's parts and contraction and redistribution on *eur*.
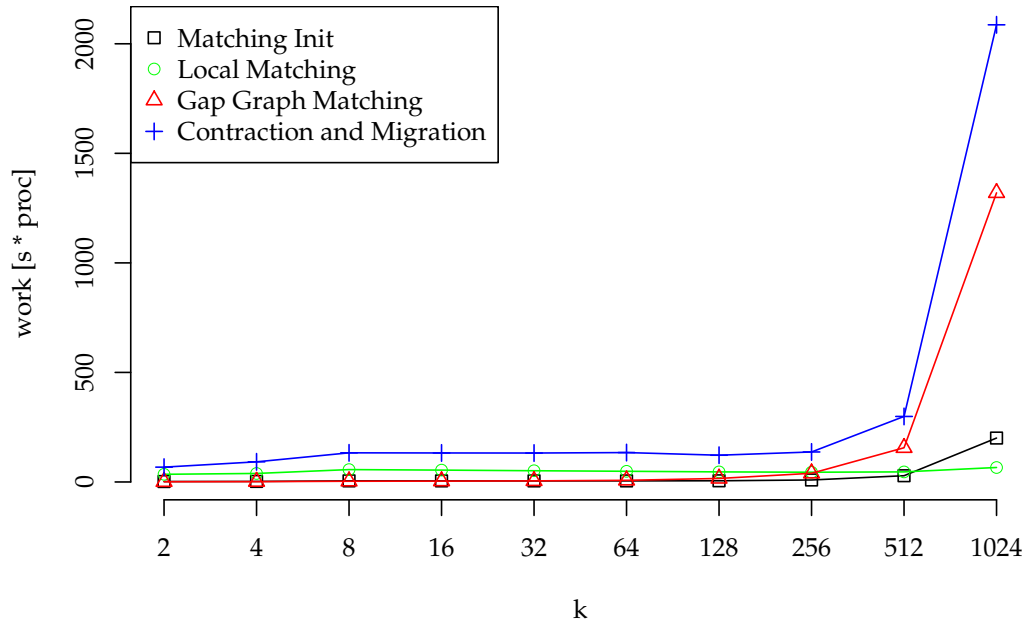
(a) Work for *weight*.
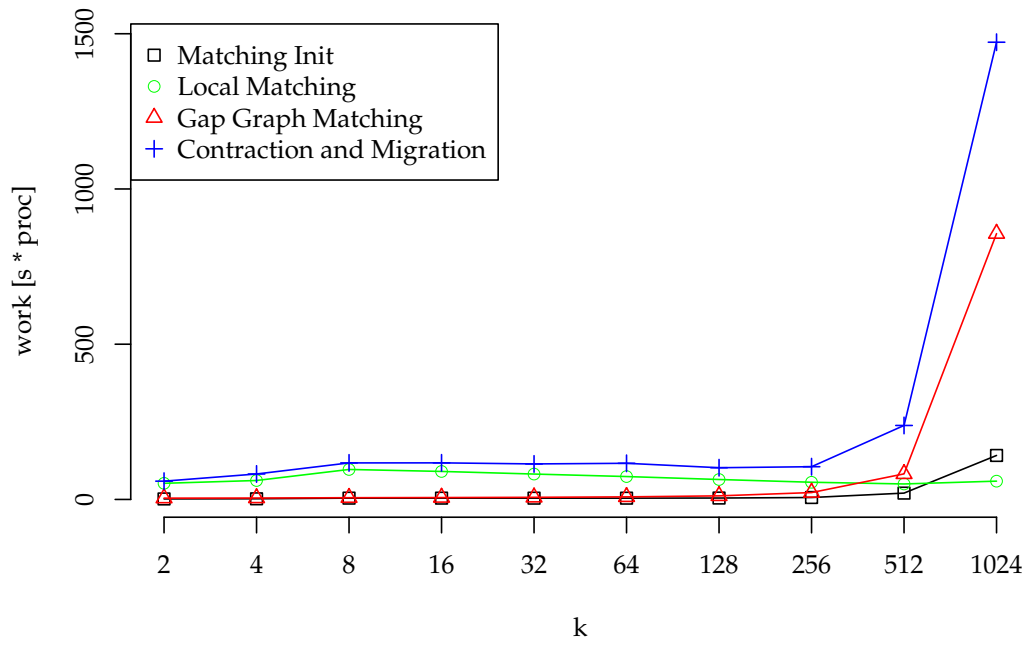
(b) Work for *inner_outer*.

Figure 6.10: Work of the matching algorithm's parts and contraction and redistribution on *eur*.
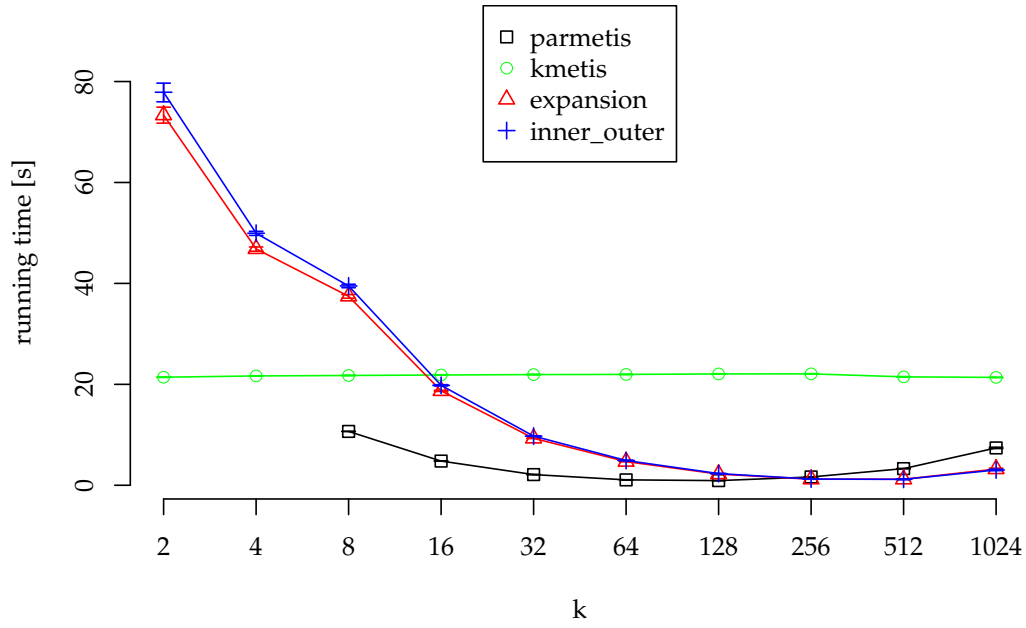
(a) Running time of coarsening without I/O.
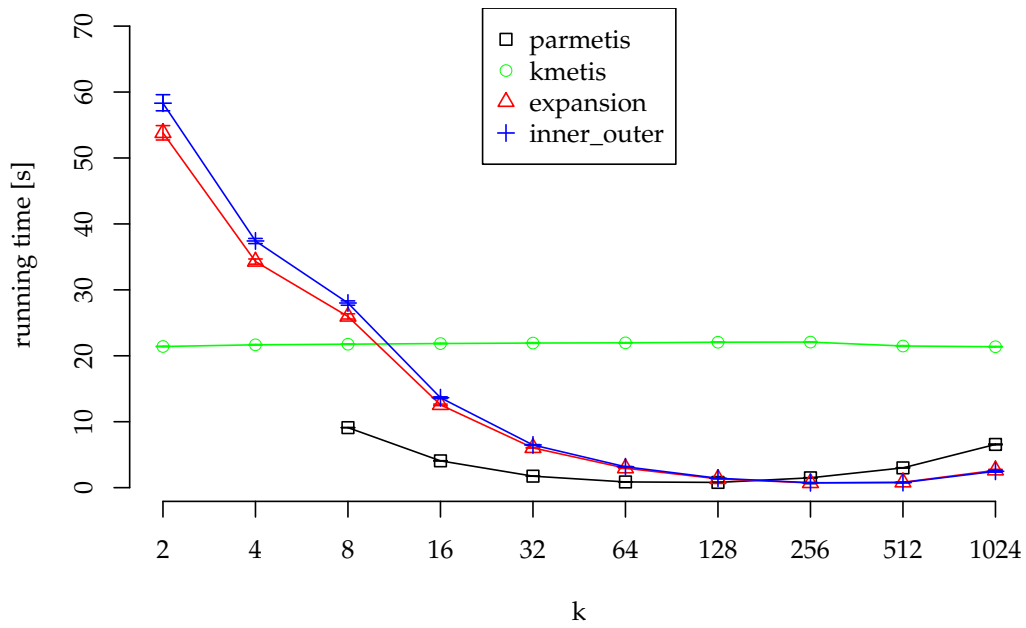
(b) Running time of coarsening without I/O, without coordinate based prepartitioning.

Figure 6.11: Running time of our parallel coarsening phase and the coarsening phase of PARMETIS on *eur*, all without initial partitioning.
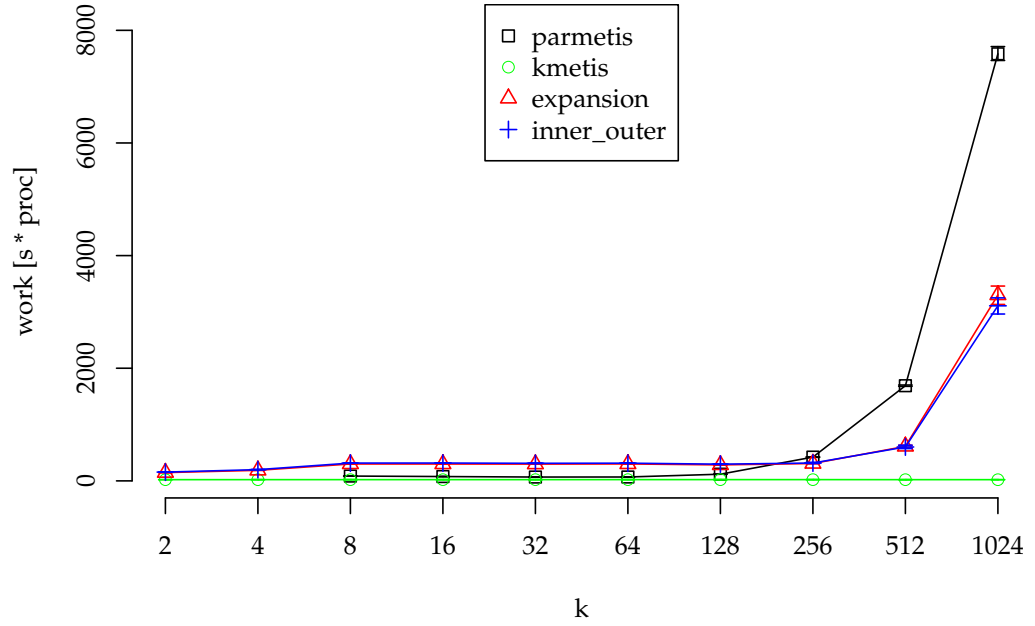
(a) Work of coarsening without I/O.
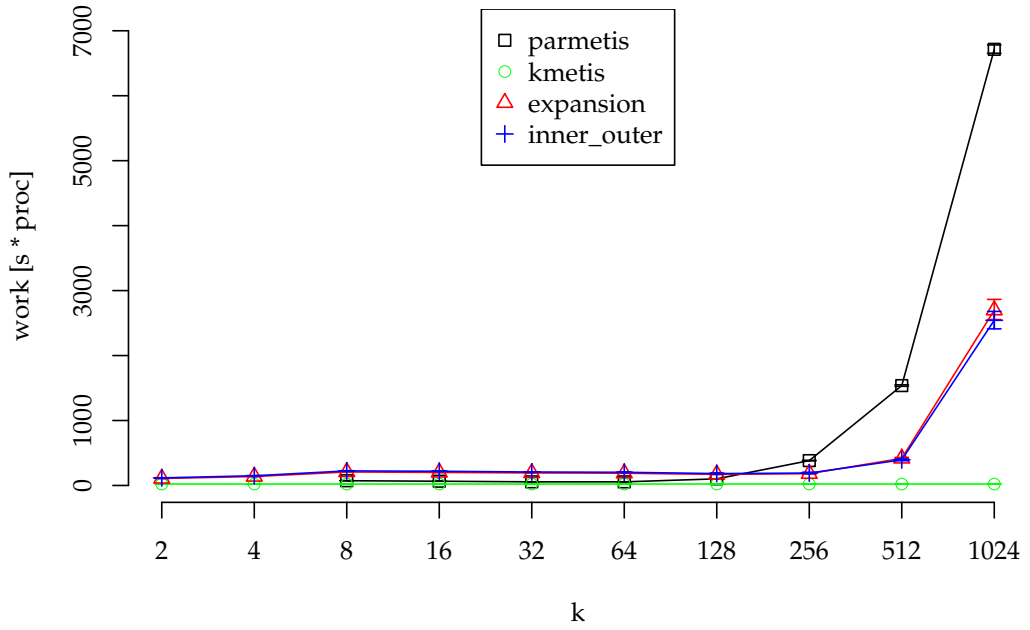
(b) Work of coarsening without I/O, without coordinate based prepartitioning.

Figure 6.12: Work of our parallel coarsening phase and the coarsening phase of PARMETIS on *eur*, all without initial partitioning.
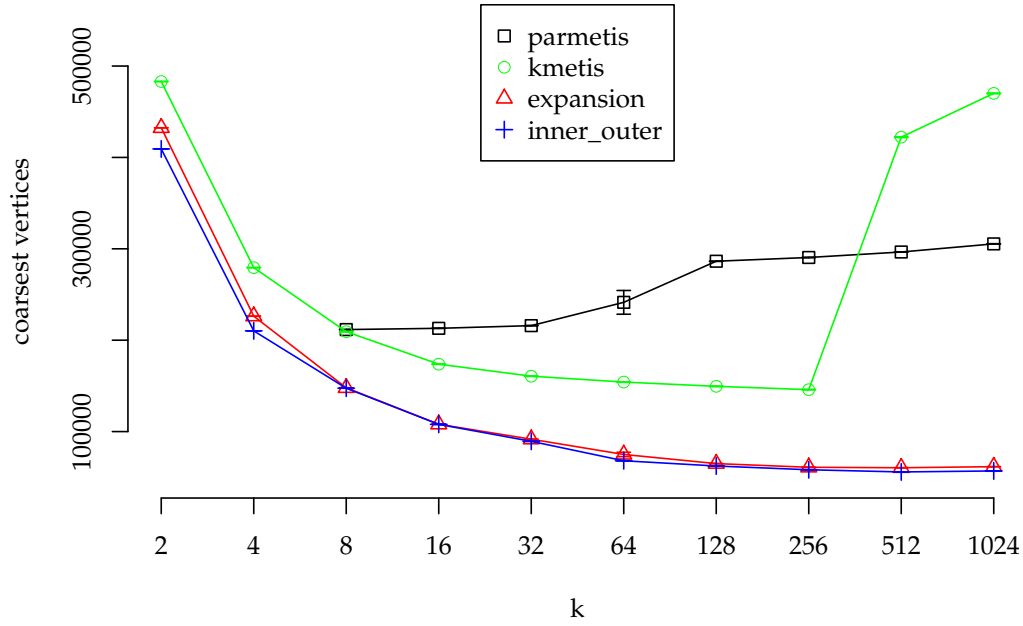
(a) Number of coarsest vertices.
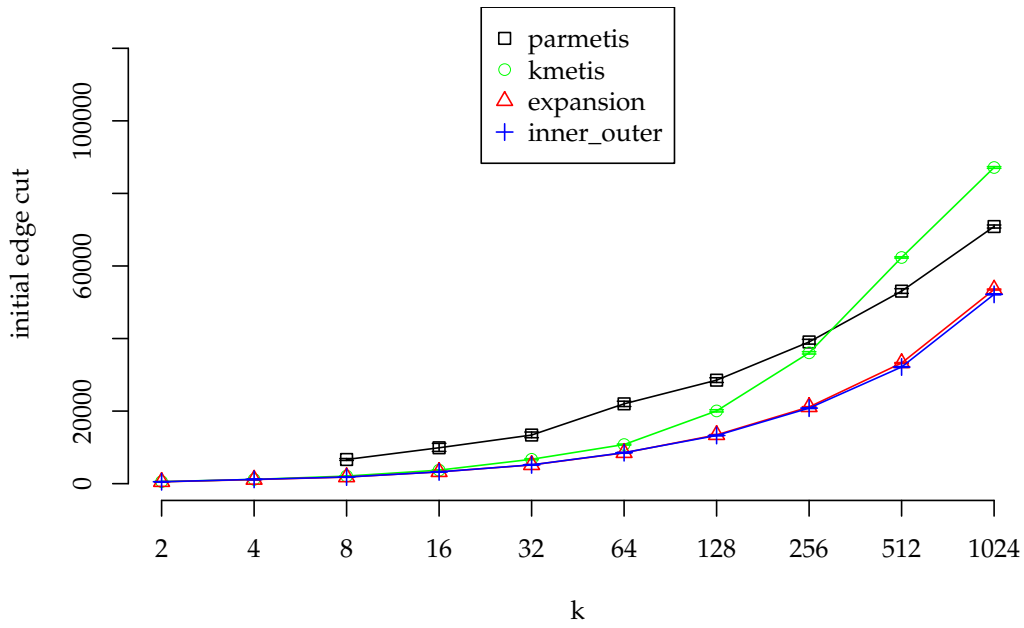
(b) Initial edge cut.

Figure 6.13: Number of coarsest vertices and initial edge cut for KMETIS, PARMETIS and our variants *inner_outer* and *expansion* on *eur*.
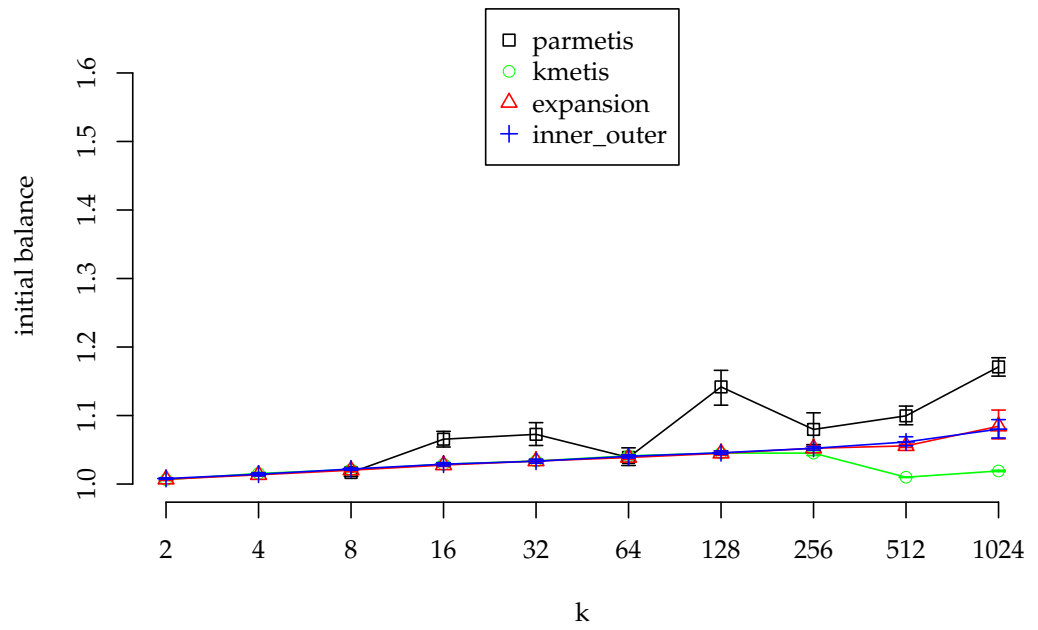
Figure 6.14: Initial balance cut for KMETIS, PARMETIS and our variants *inner_outer* and *expansion* on *eur*.
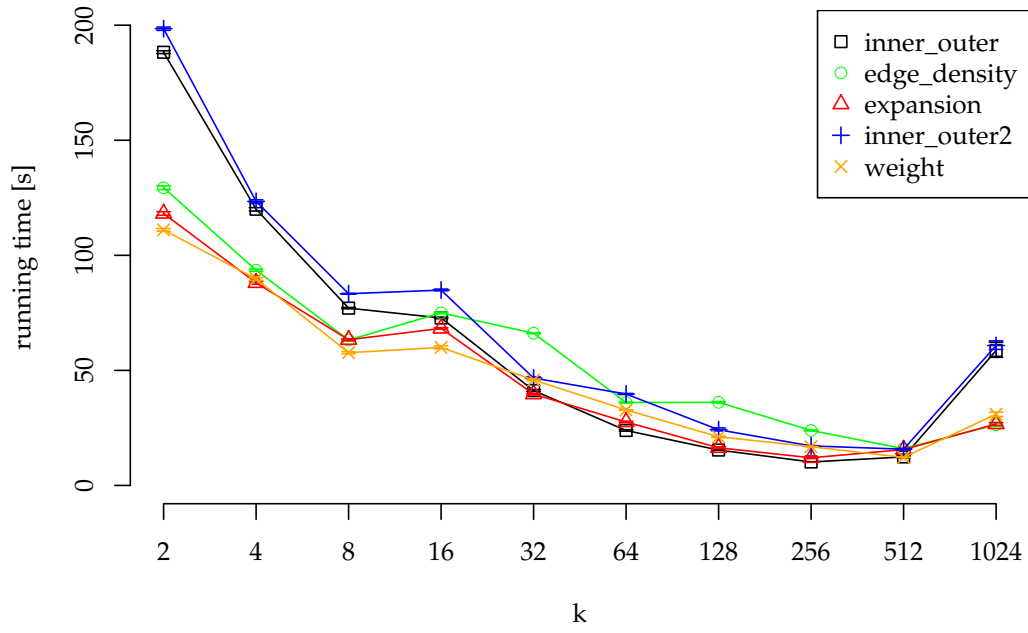
(a) Running time of coarsening with initial partitioning.
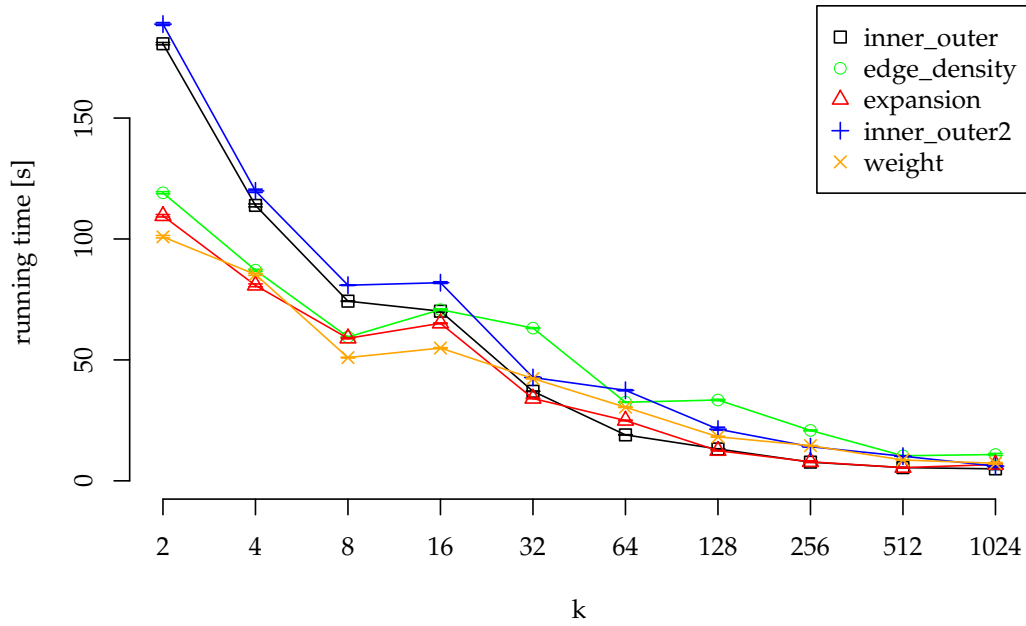
(b) Running time of coarsening without initial partitioning.

Figure 6.15: Running time of coarsening using various rating variants on *cage15*.
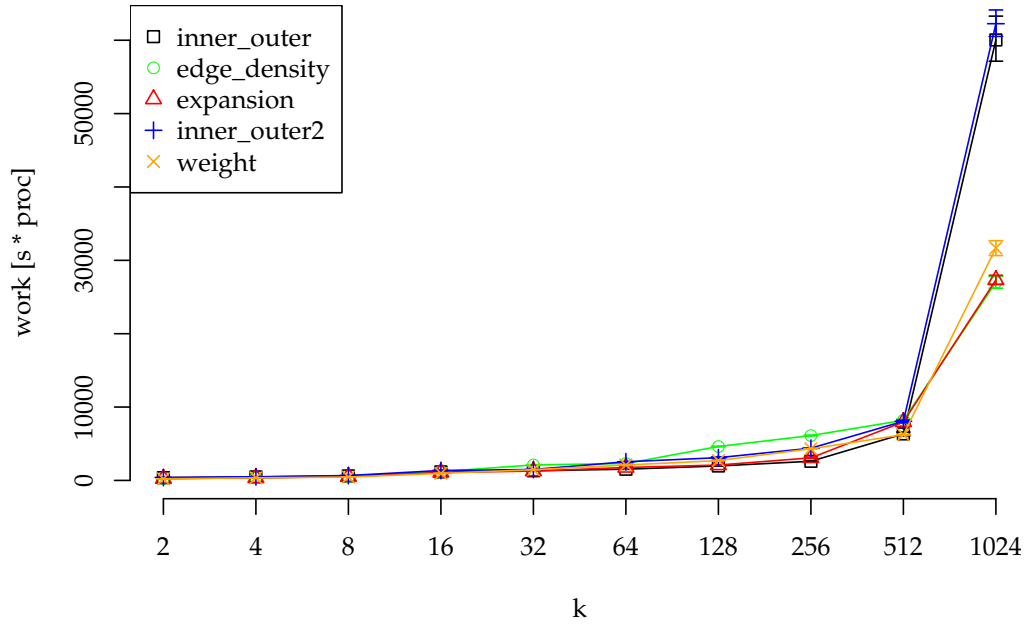
(a) Work of coarsening with initial partitioning.
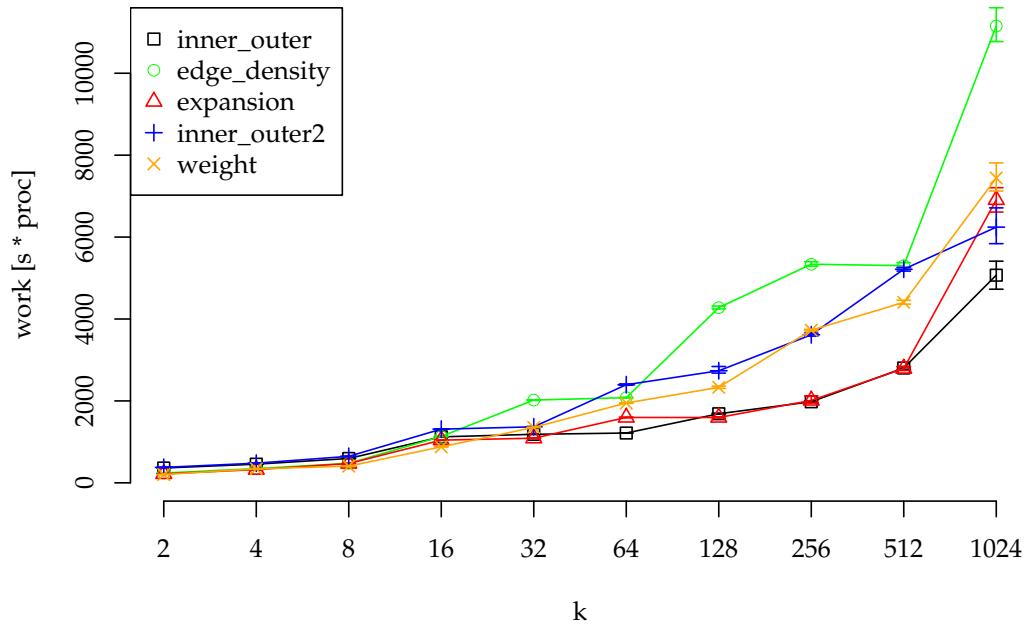
(b) Work of coarsening without initial partitioning.

Figure 6.16: Work of coarsening using various rating variants on *cage15*.
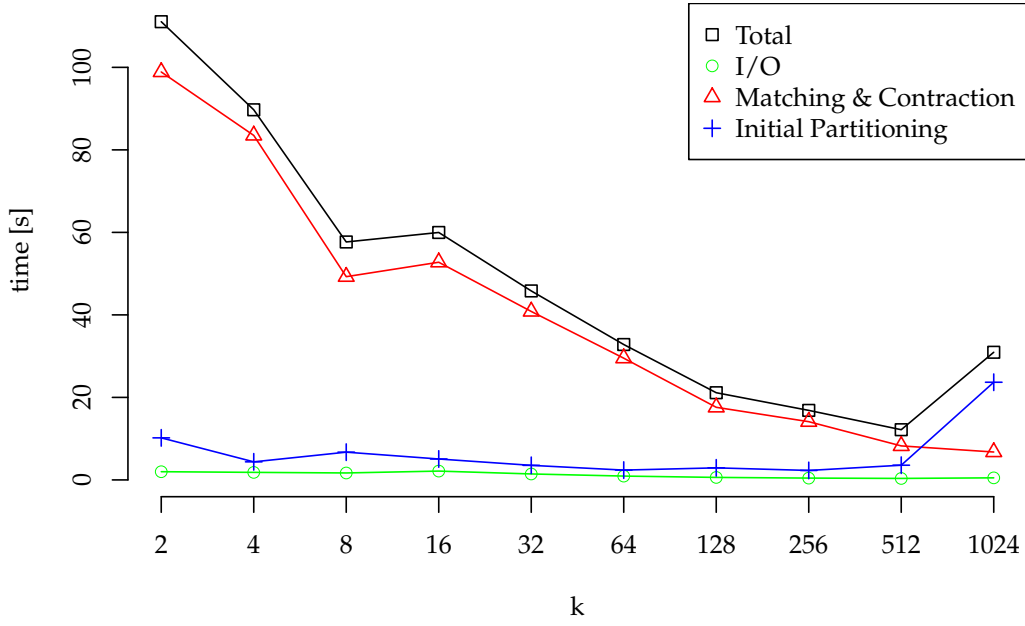
(a) Running time of the coarsening phase parts for *weight*.
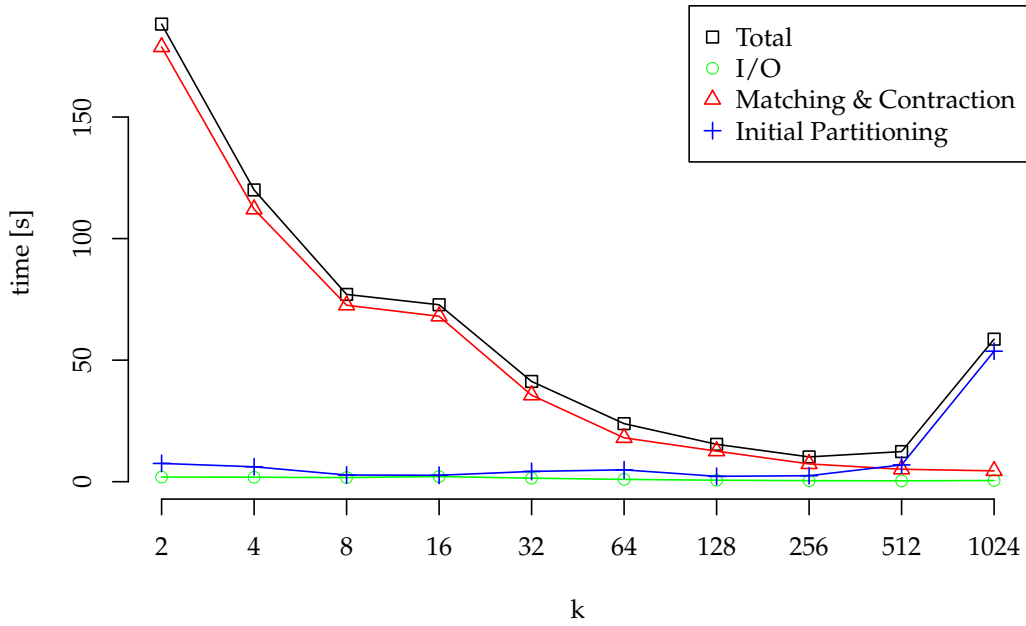
(b) Running time of the coarsening phase parts for *inner_outer*.

Figure 6.17: Running time of the parts of the coarsening phase *cage15*.
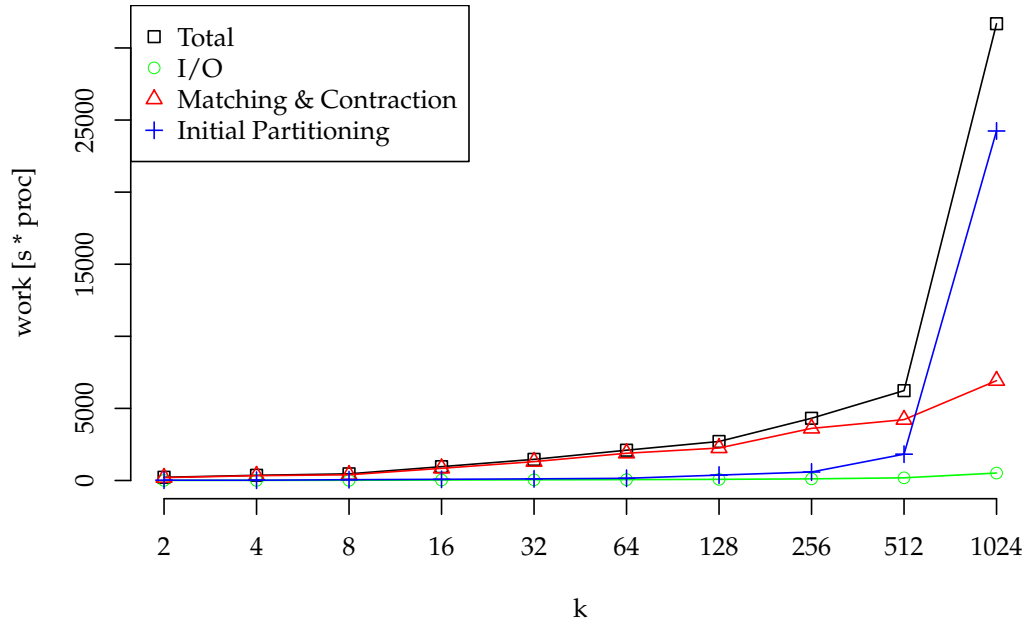
(a) Work of the coarsening phase parts for *weight*.
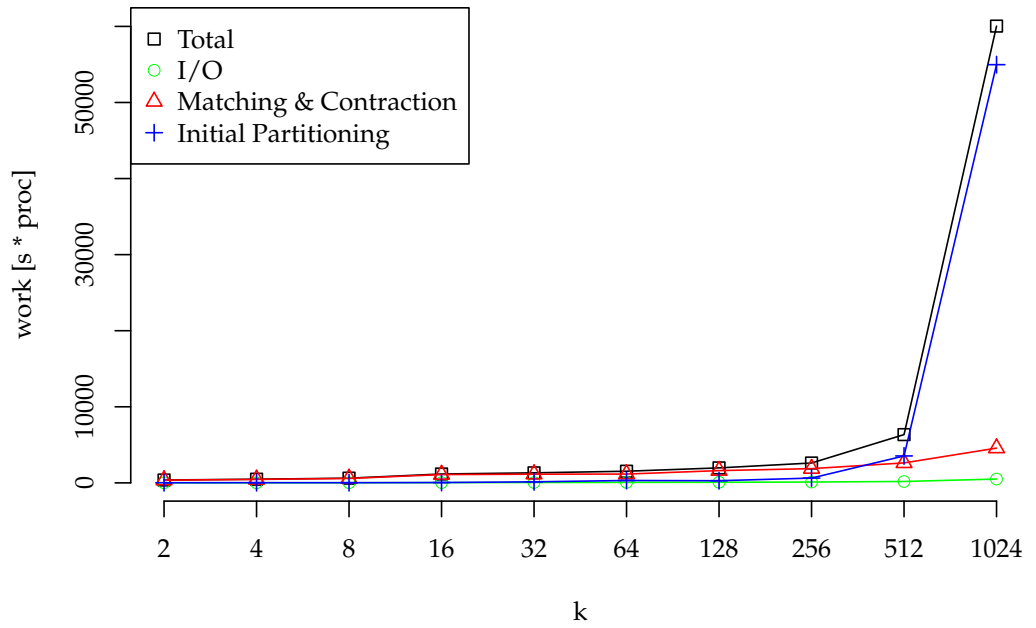
(b) Work of the coarsening phase parts for *inner_outer*.

Figure 6.18: Work of the parts of the coarsening phase *cage15*.
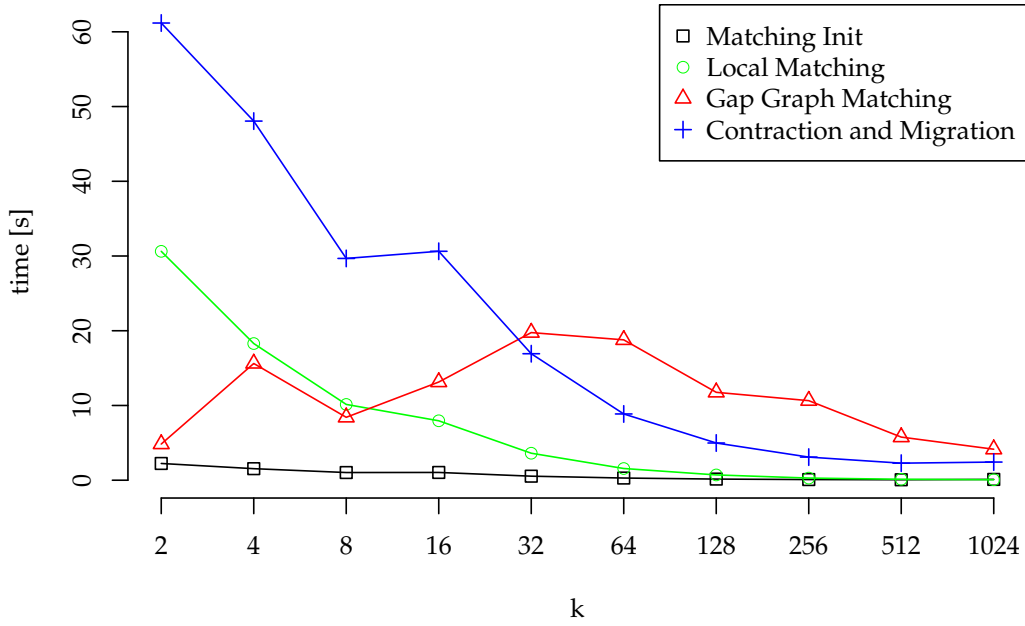
(a) Running time for *weight*.
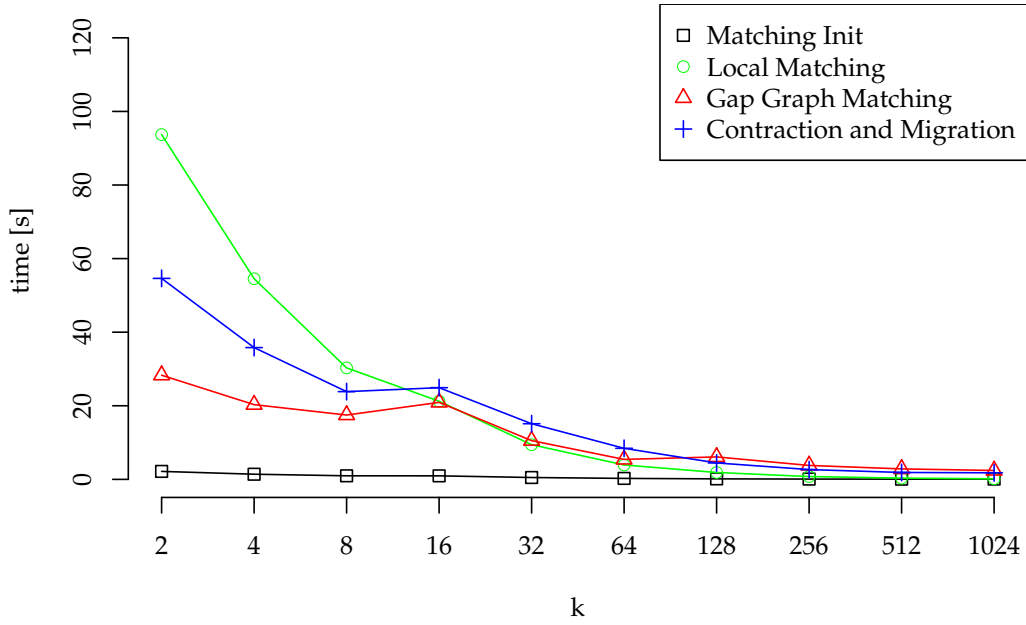
(b) Running time for *inner_outer*.

Figure 6.19: Running time of the matching algorithm's parts and contraction and redistribution on *cage15*.
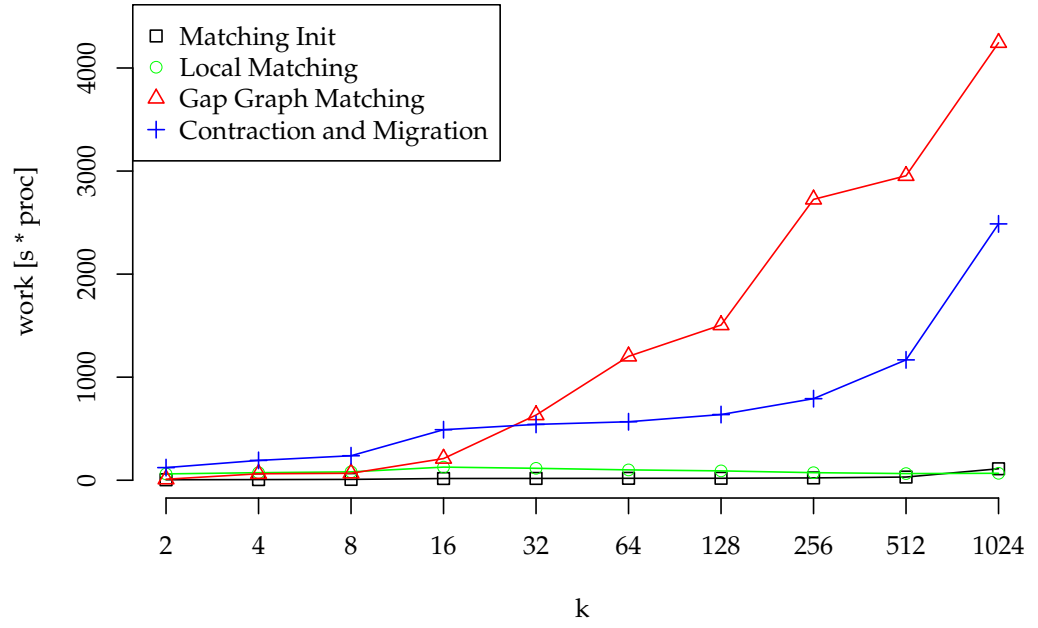
(a) Work for *weight*.
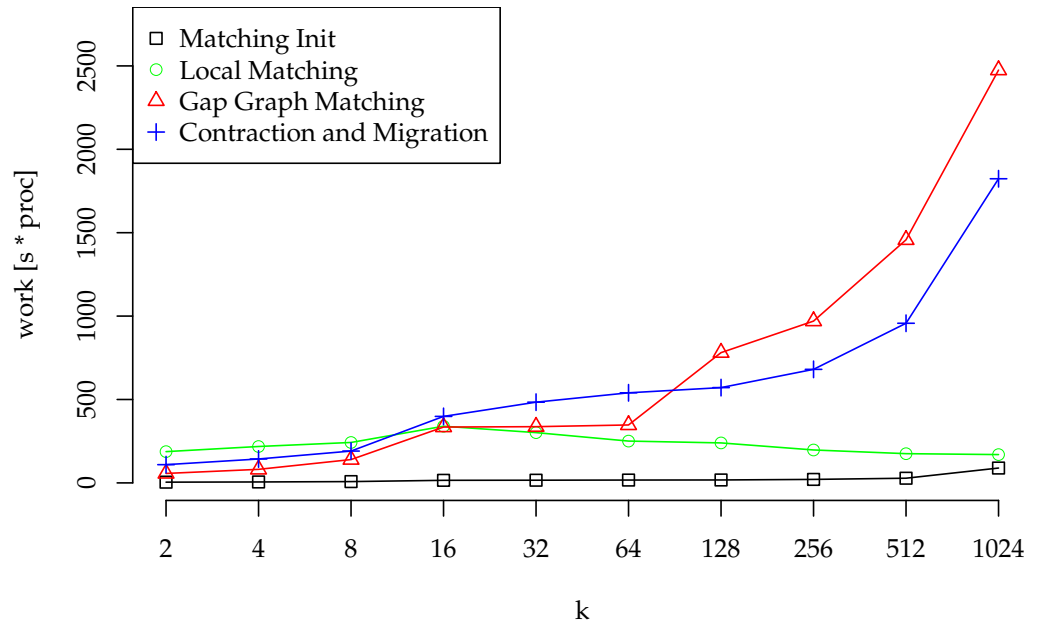
(b) Work for *inner_outer*.

Figure 6.20: Work of the matching algorithm's parts and contraction and redistribution on *cage15*.
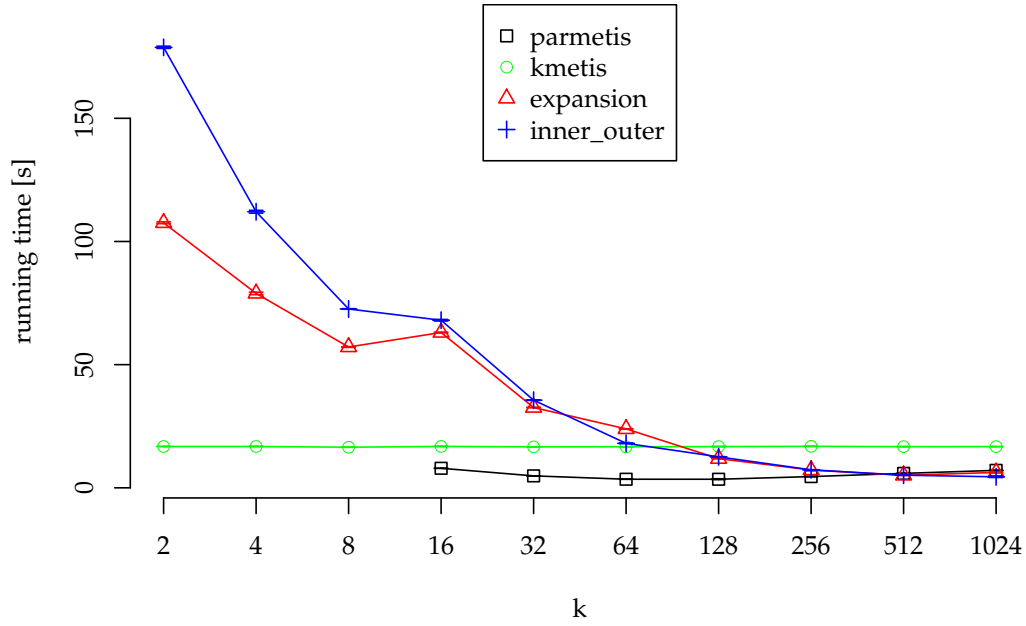
(a) Running time of coarsening without I/O.
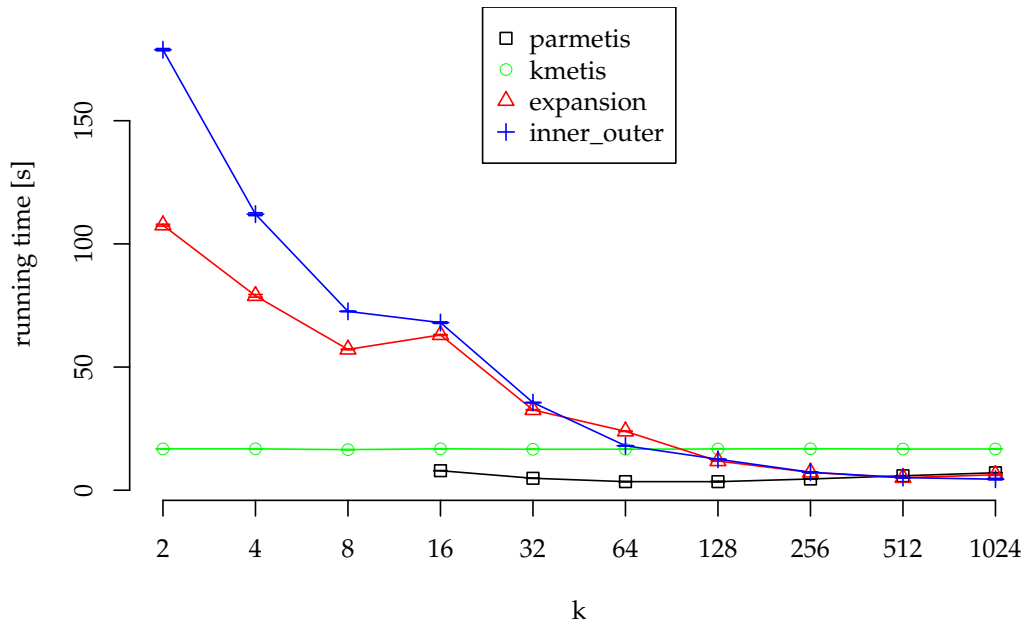
(b) Running time of coarsening without I/O, without coordinate based prepartitioning.

Figure 6.21: Running time of our parallel coarsening phase and the coarsening phase of PARMETIS on *cage15*, all without initial partitioning.
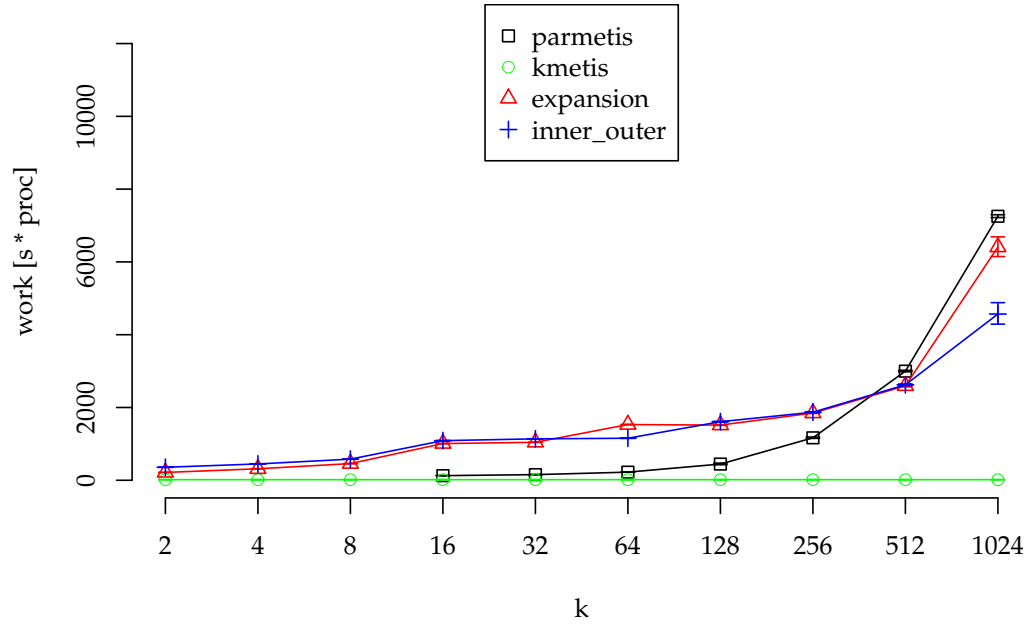
(a) Work of coarsening without I/O.
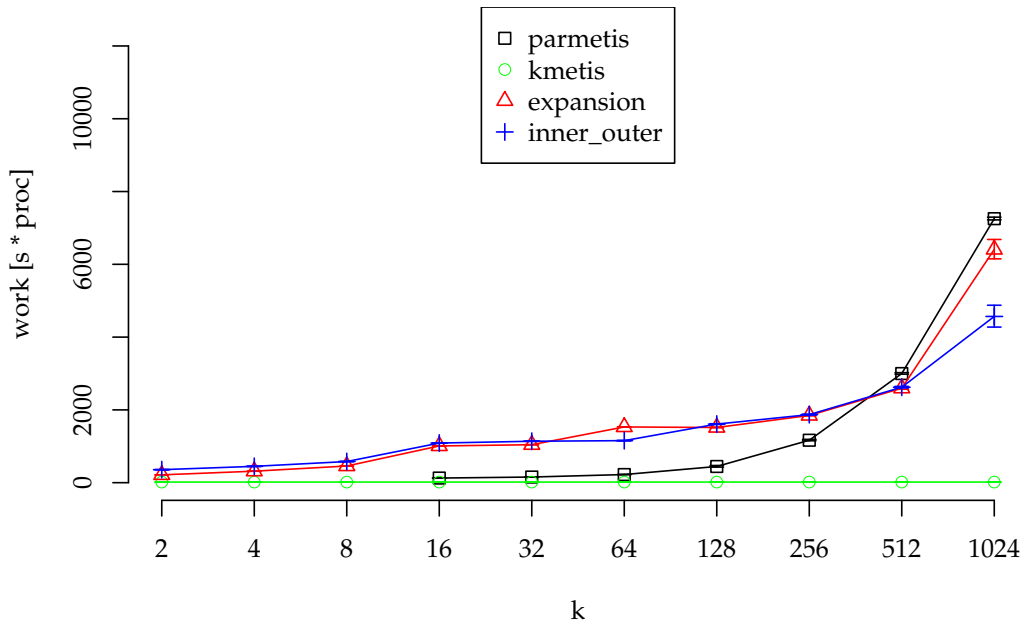
(b) Work of coarsening without I/O, without coordinate based prepartitioning.

Figure 6.22: Work of our parallel coarsening phase and the coarsening phase of PARMETIS on *cage15*, all without initial partitioning.
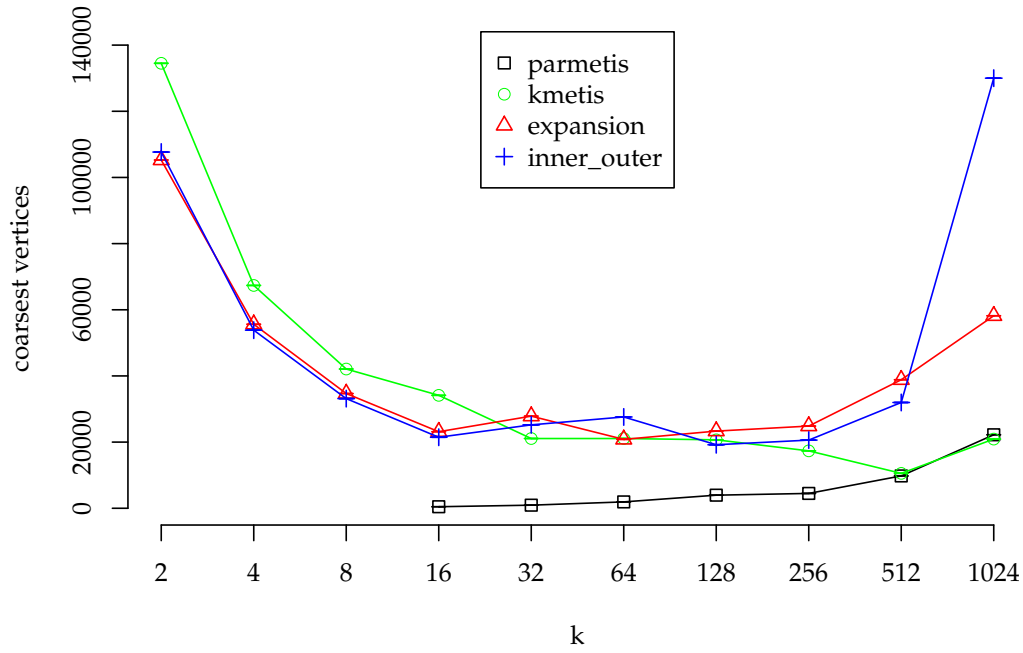
(a) Number of coarsest vertices.
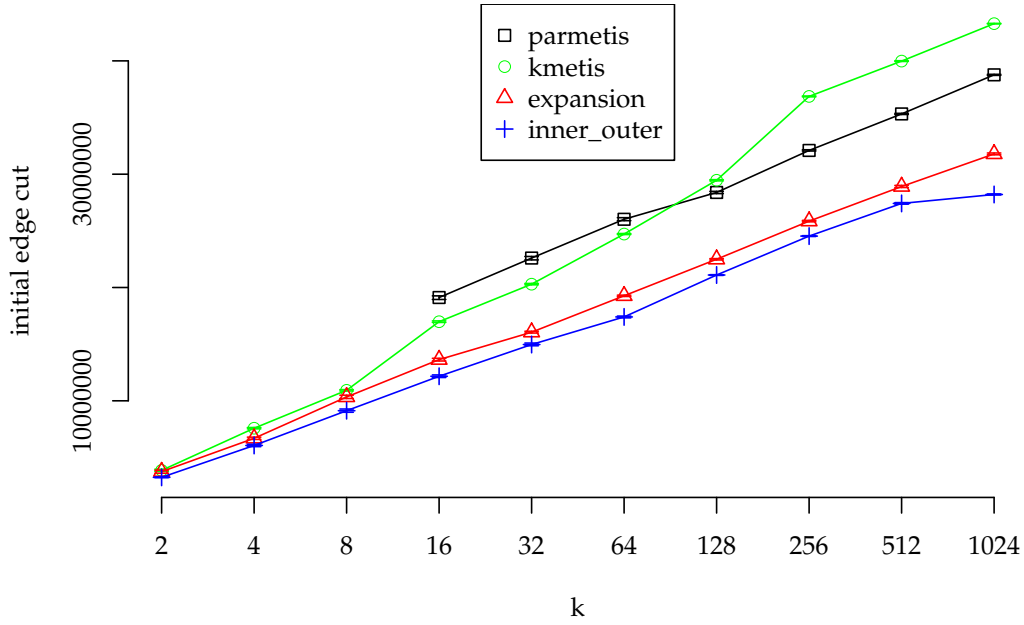
(b) Initial edge cut.

Figure 6.23: Number of coarsest vertices and initial edge cut for KMETIS, PARMETIS and our *inner_outer* and *expansion* variant on *cage15*.
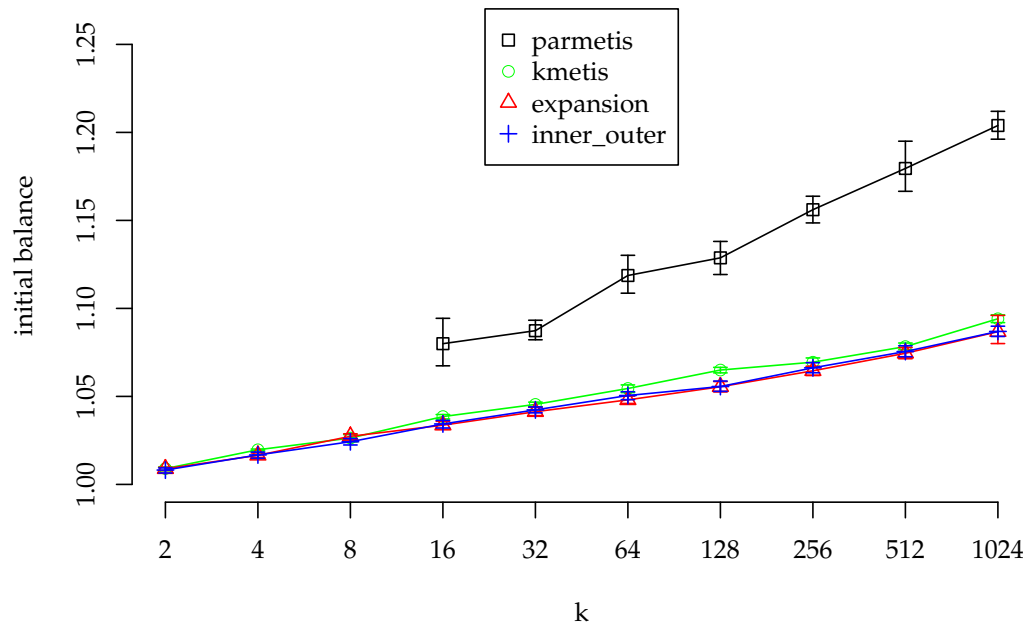
Figure 6.24: Initial balance for KMETIS, PARMETIS and our variants *inner_outer* and *expansion* on *cage15*.
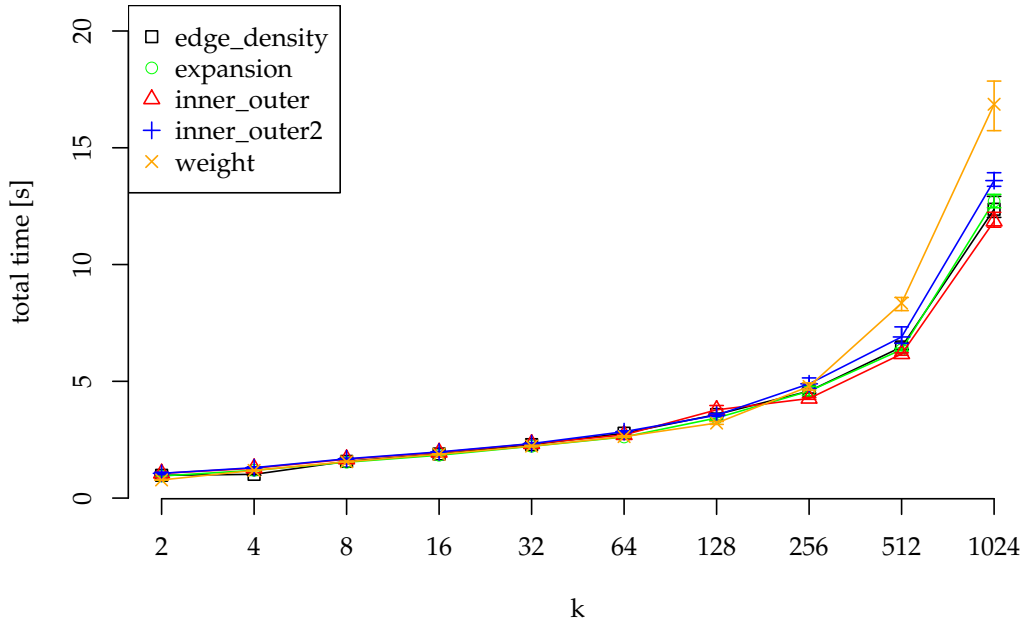
(a) Coarsening phase including partitioning.
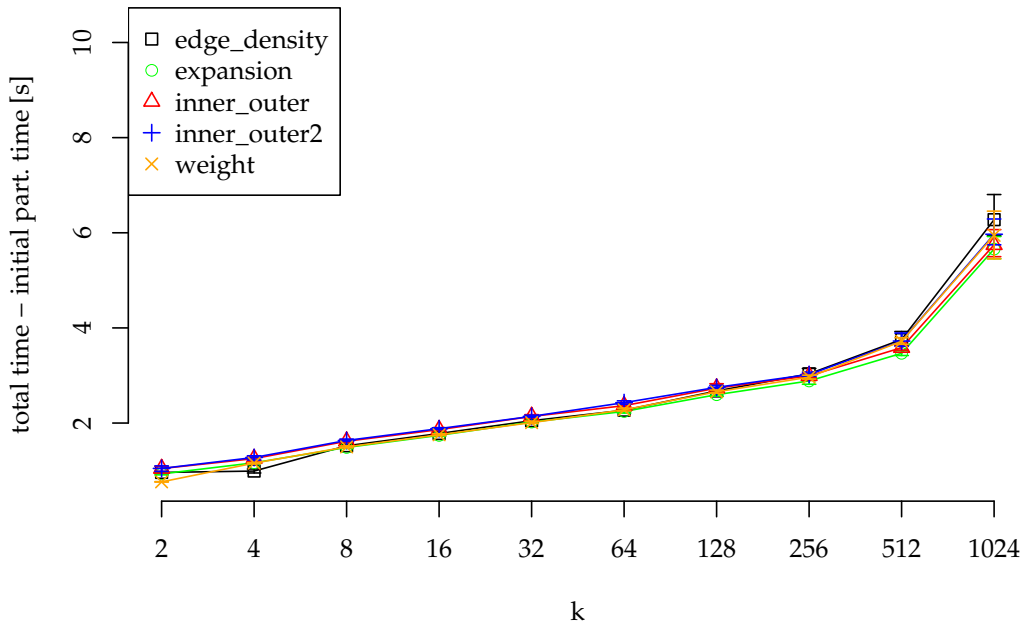
(b) Coarsening phase only, no partitioning.

Figure 6.25: Running time of the coarsening phase with and without initial partitioning on Delaunay triangulation graphs. For k processes, the Delaunay triangulation has $2^{16+\log k}$ vertices.
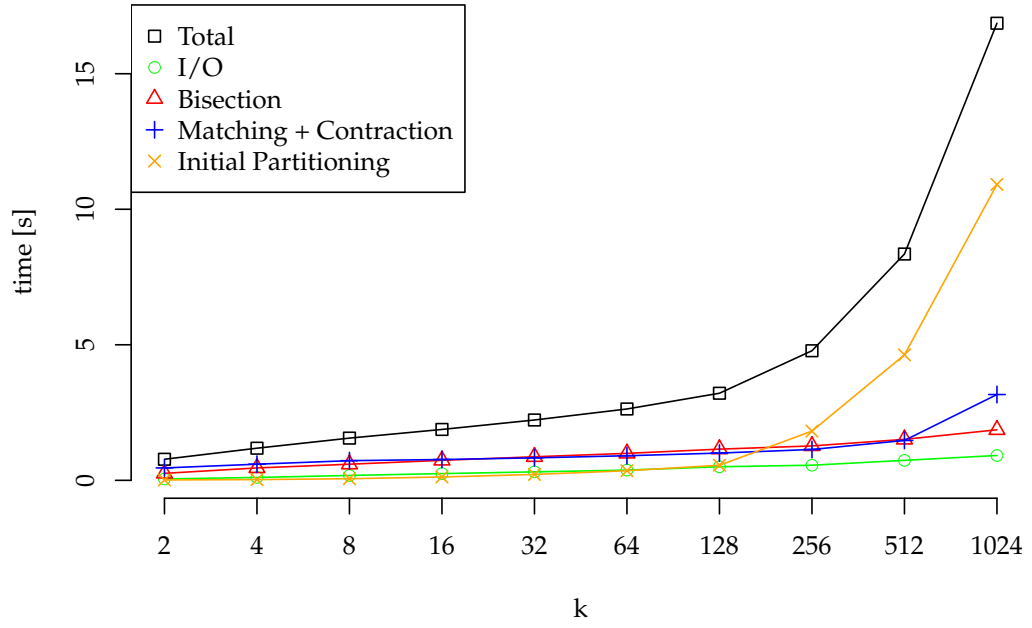
(a) Coarsening with rating *weight* broken down by step.
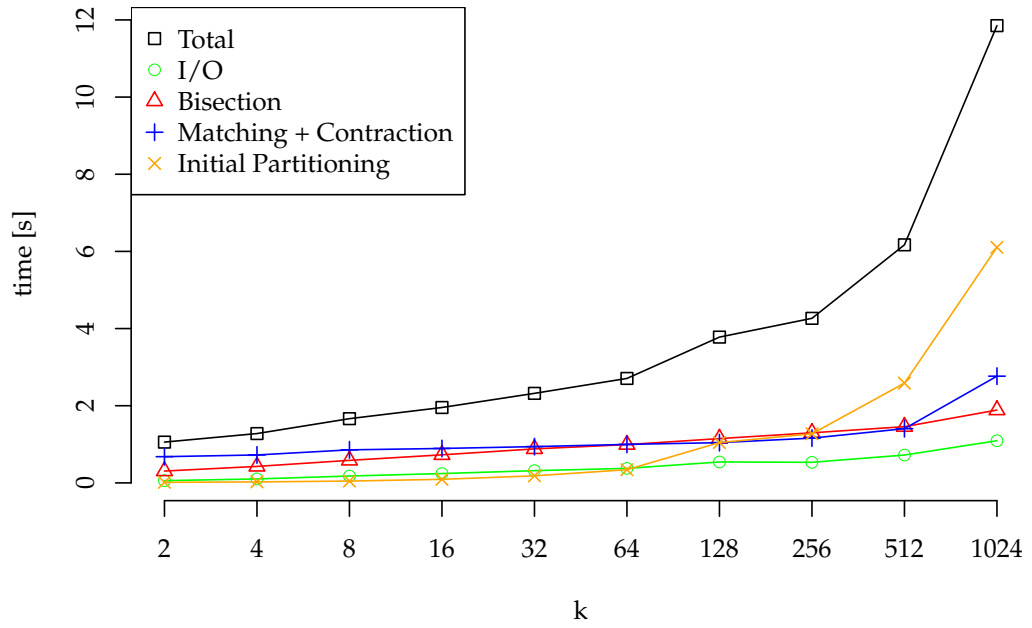
(b) Coarsening with rating *inner_outer* broken down by step.

Figure 6.26: Running times of the steps of the coarsening phase on the Delaunay triangulations. For k processes, the Delaunay triangulation has $2^{16+\log k}$ vertices.
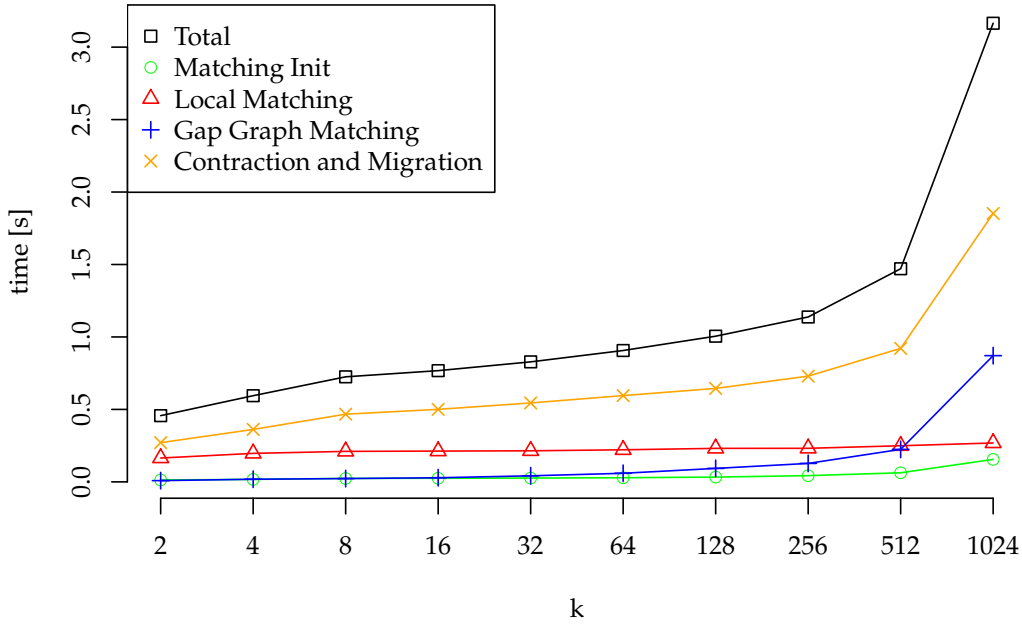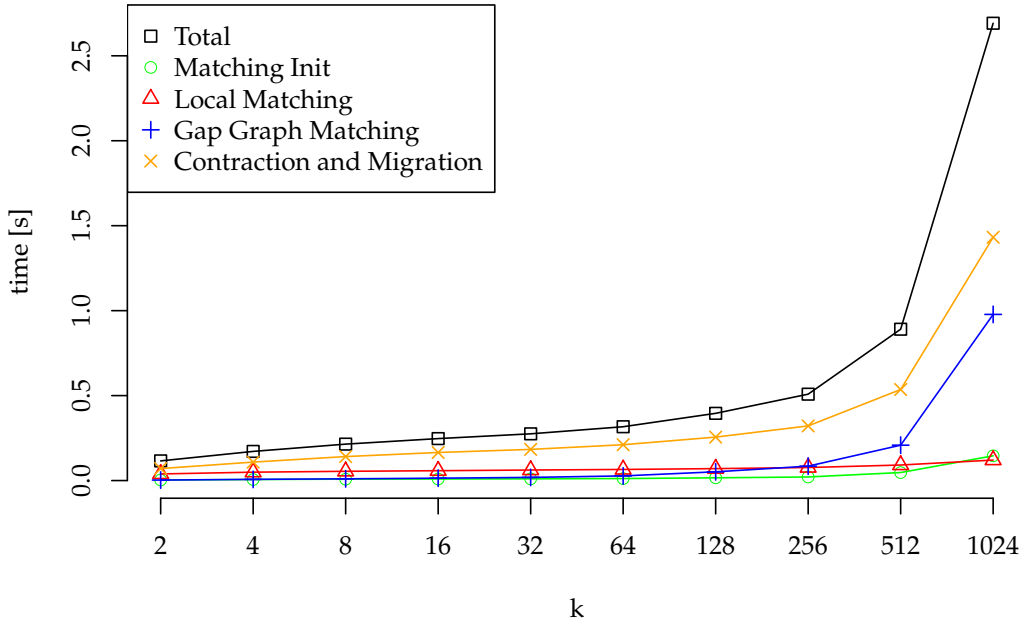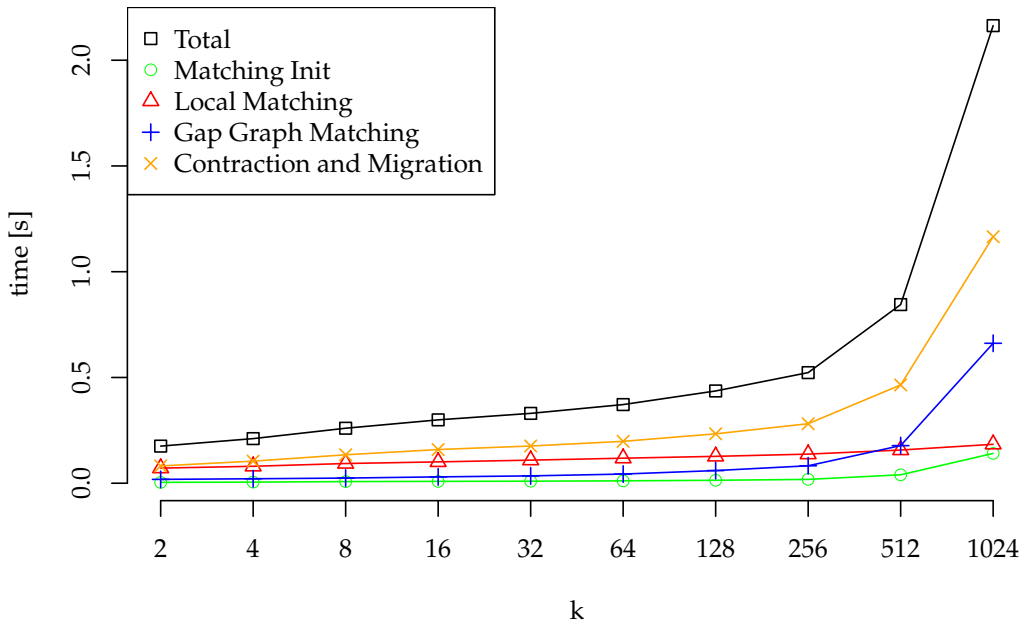
(a) Running time for the contraction and migration step's parts for *weight*.
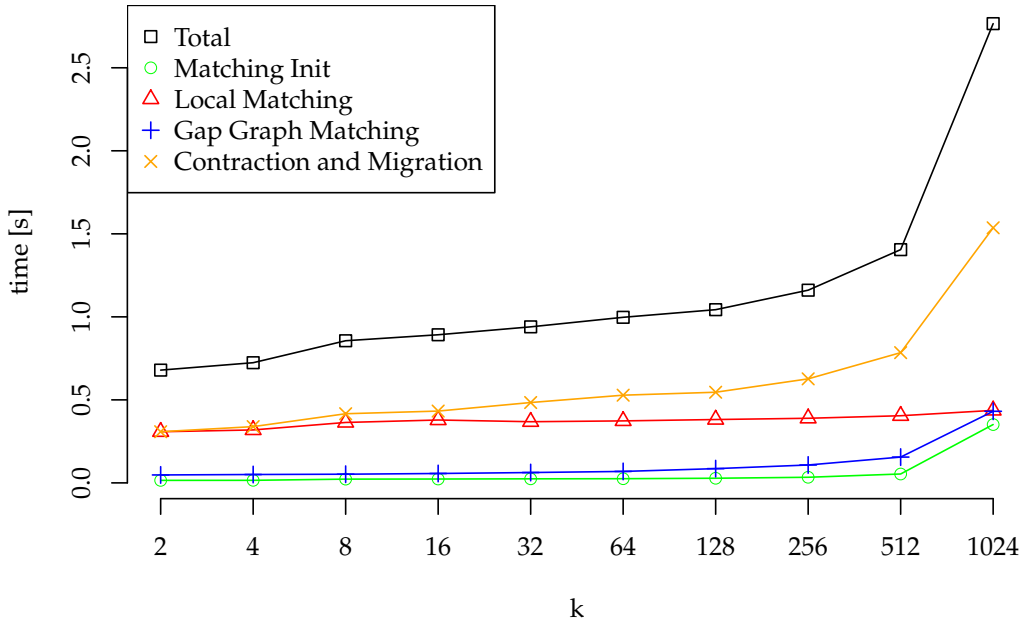
(b) Running time for the contraction and migration step's parts for *inner_outer*.

Figure 6.27: Running time for the contraction and migration step's parts for *weight* and *inner_outer*. For k processes, the Delaunay triangulation has $2^{16+\log k}$ vertices.
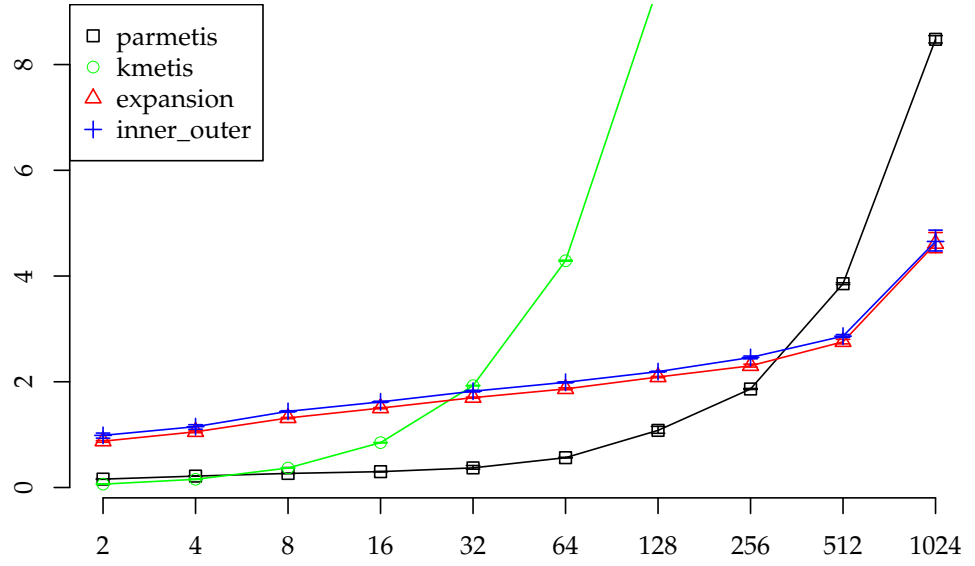
(a) Running time without I/O.
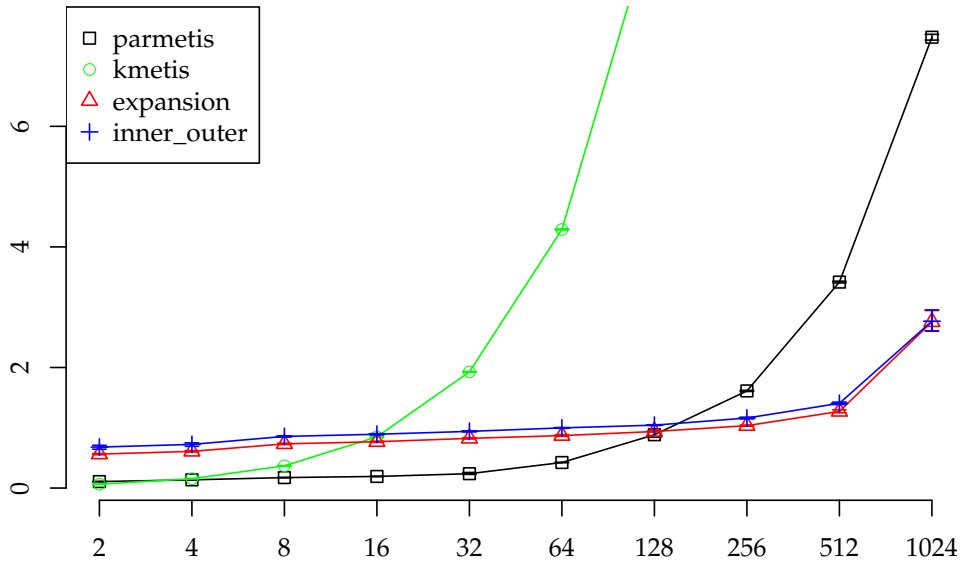
(b) Running time without I/O and coordinate based prepartitioning.

Figure 6.28: Running times of our parallel coarsening phase and the coarsening phase of METIS on the Delaunay triangulation graphs. All times exclude the initial partitioning. For k processes, the Delaunay triangulation has $2^{16+\log k}$ vertices.
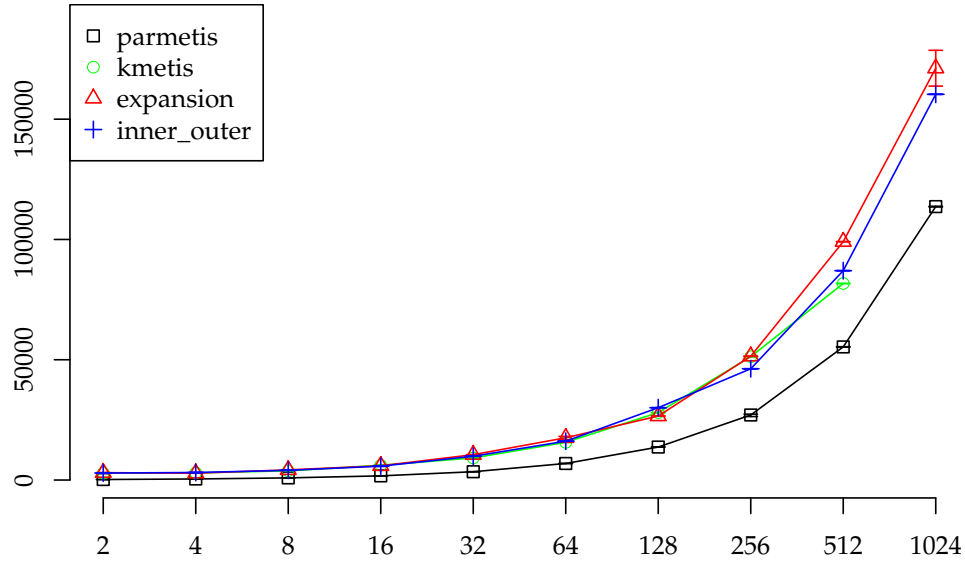
(a) Number of coarsest vertices.
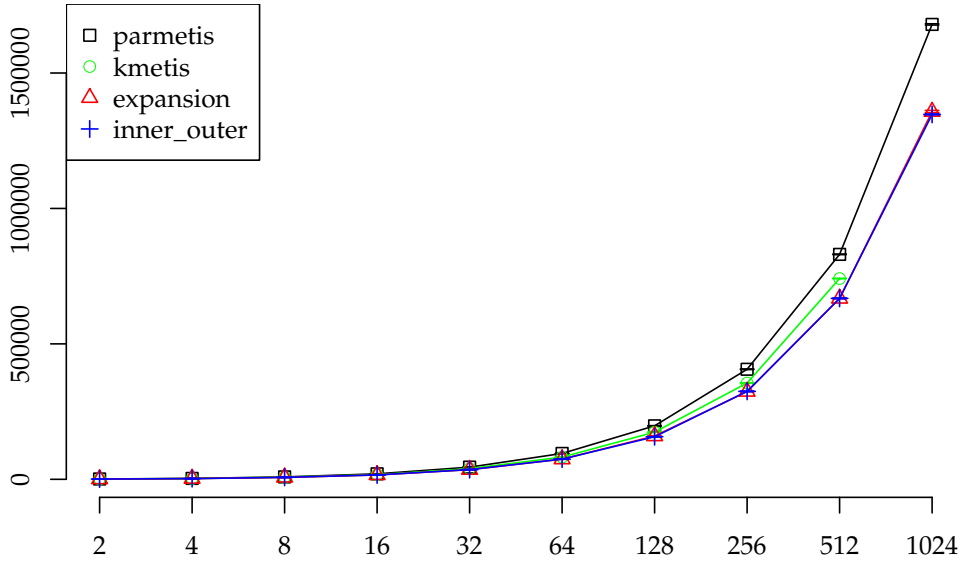
(b) Initial edge cut.

Figure 6.29: Number of coarsest vertices and initial edge cut for KMETIS, PARMETIS and our variants *inner_outer* and *expansion* on the Delaunay triangulations.
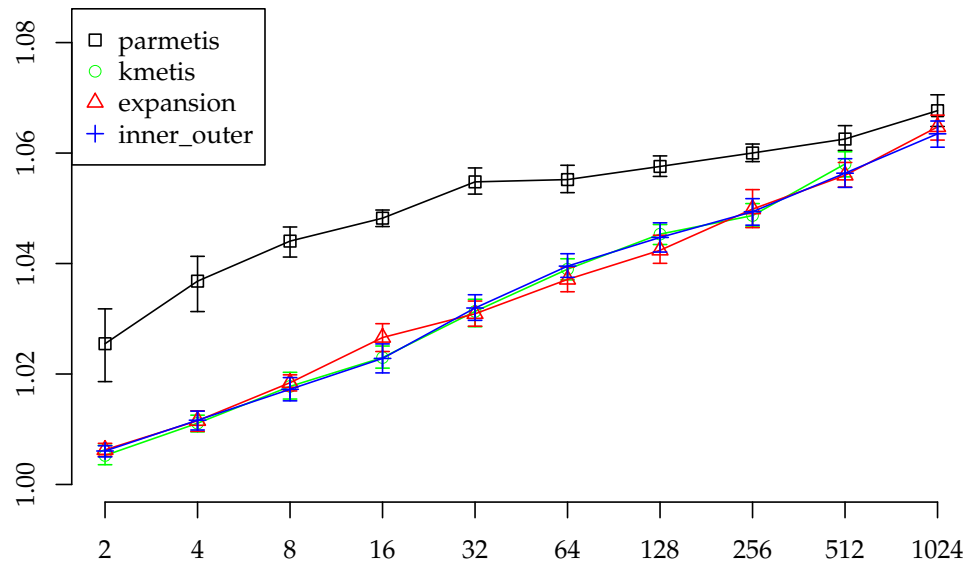
Figure 6.30: Initial balance for KMETIS, PARMETIS and our variants *inner_outer* and *expansion* on the Delaunay triangulations.
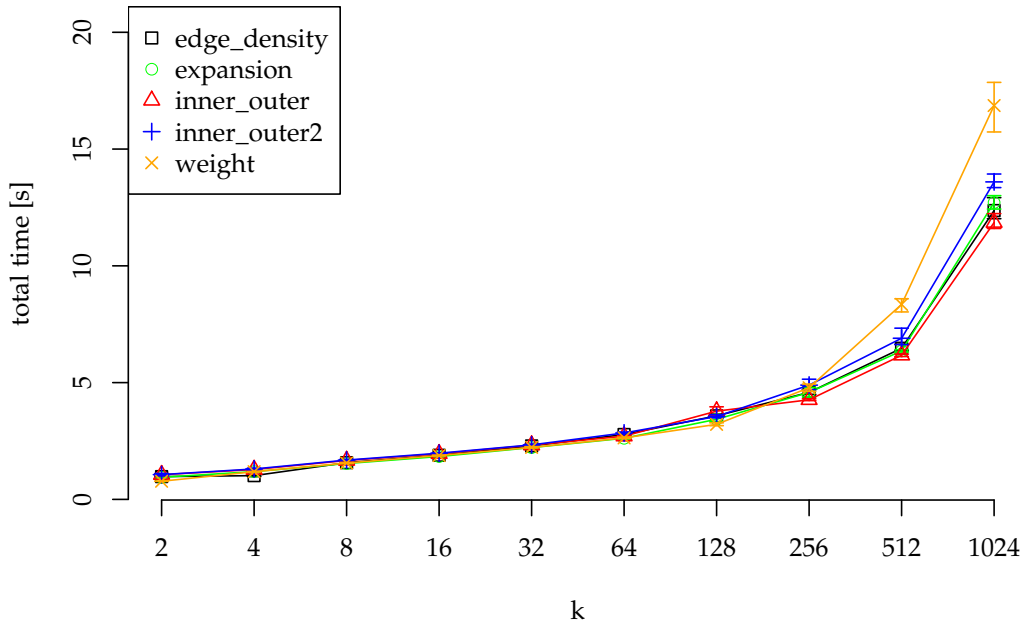
(a) Coarsening phase including partitioning.
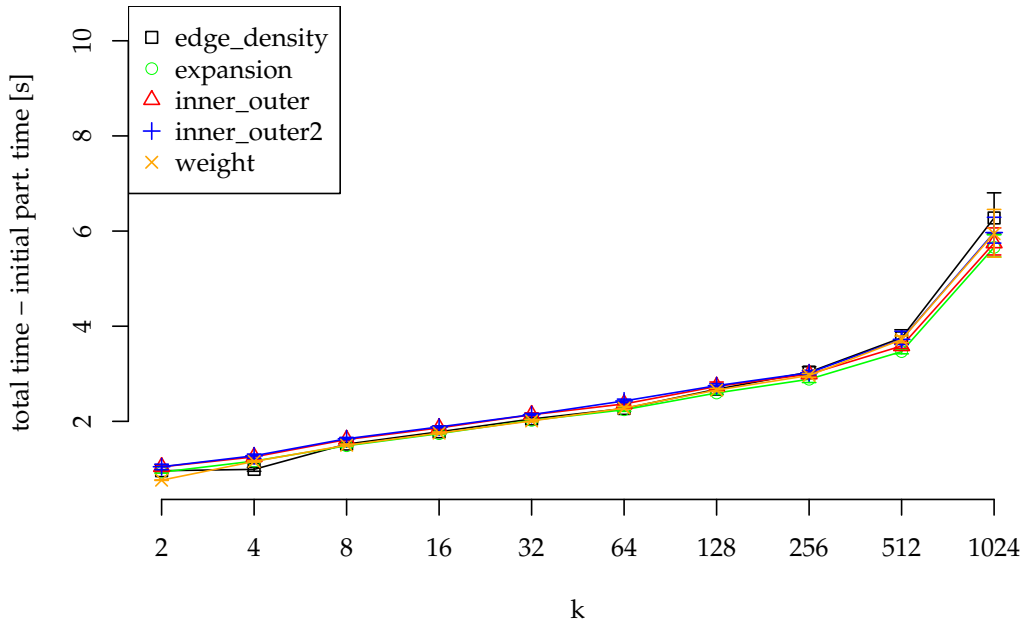
(b) Coarsening phase only, no partitioning.

Figure 6.31: Running time of the coarsening phase with and without initial partitioning on random geometric graphs. For *k* processes, the random geometric graph has $2^{14+\log k}$ vertices.
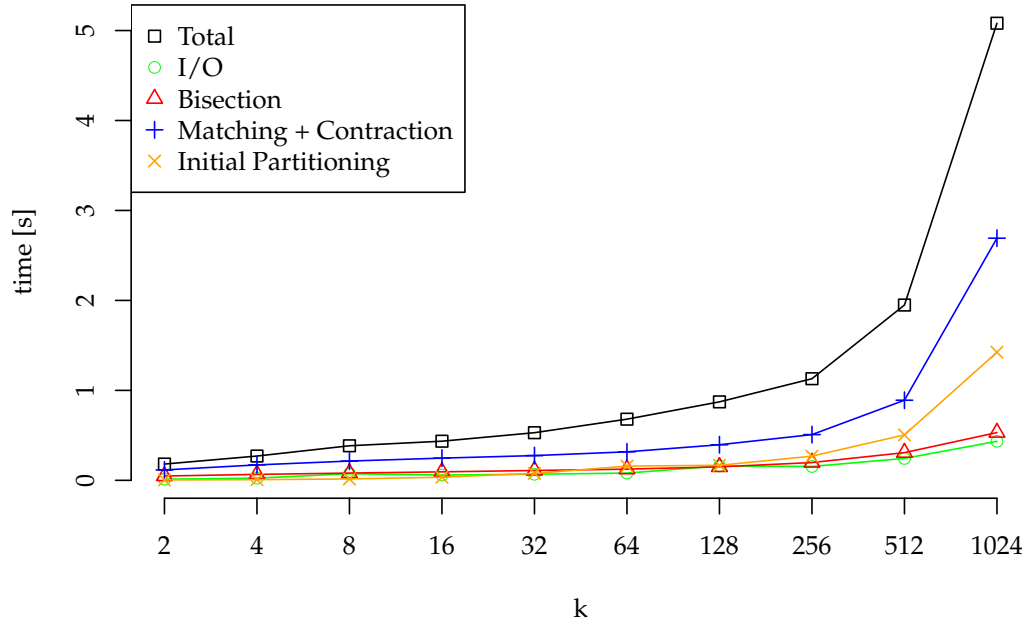
(a) Coarsening with rating *weight* broken down by step.
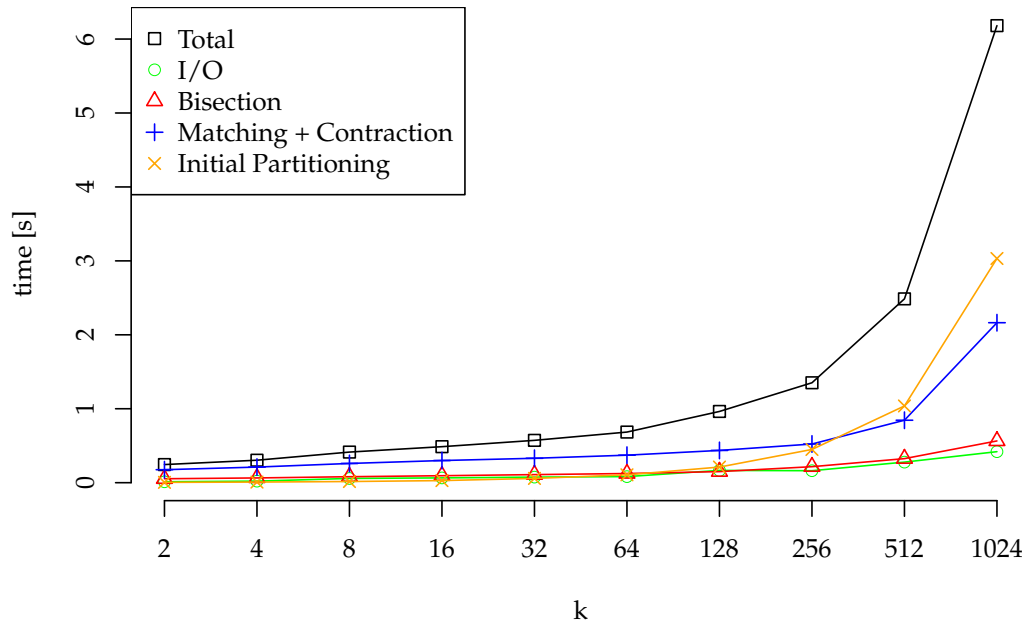
(b) Coarsening with rating *inner_outer* broken down by step.

Figure 6.32:   Running times of the steps of the coarsening phase on the random geometric graphs. For k processes, the random geometric graph has $2^{14+\log k}$ vertices.

(a) Running time for the contraction and migration step's parts for *weight*.

(b) Running time for the contraction and migration step's parts for *inner_outer*.

Figure 6.33: Running time for the contraction and migration step's parts for *weight* and *inner_outer*. For k processes, the random geometric graph has $2^{14+\log k}$ vertices.
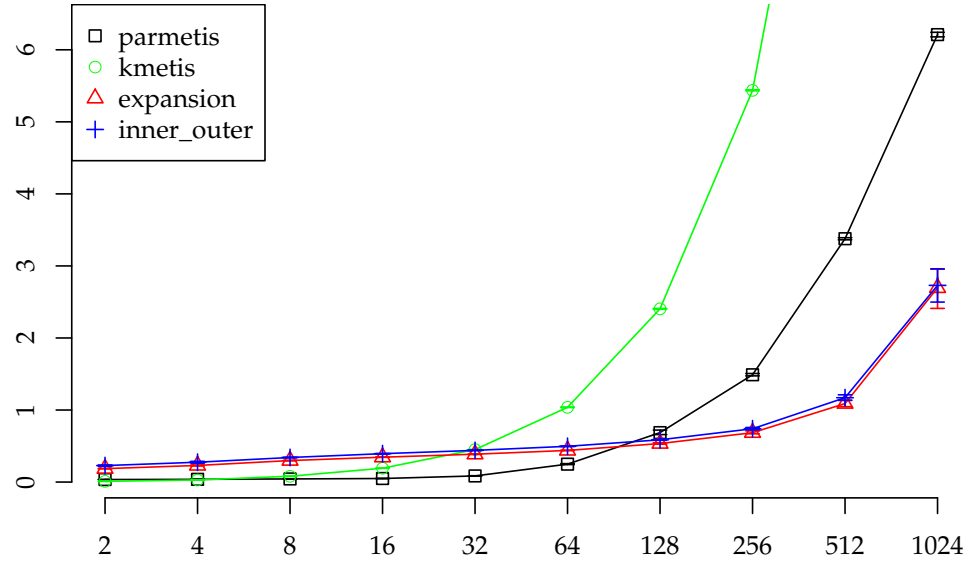
(a) Running time without I/O.
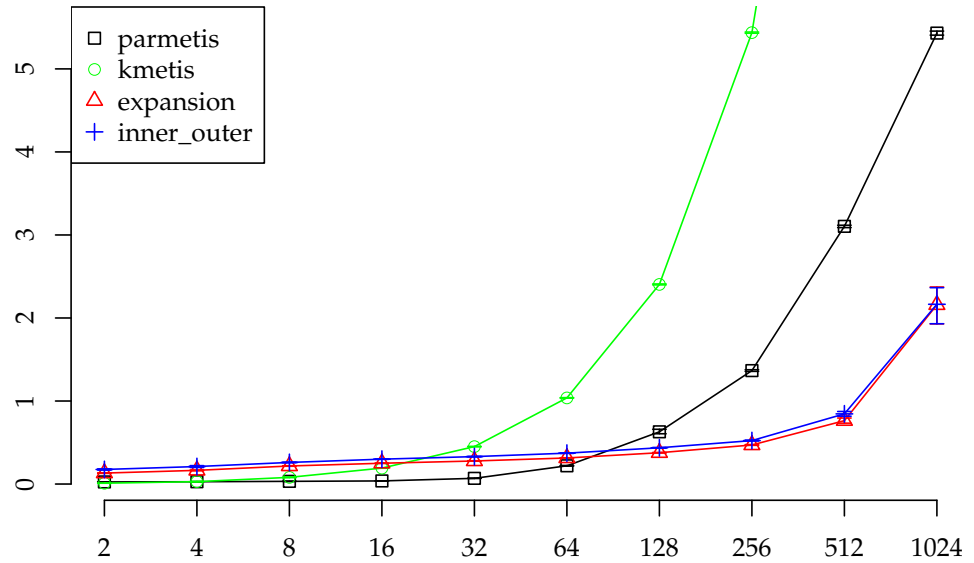
(b) Running time without I/O and coordinate based prepartitioning.

Figure 6.34: Running times of our coarsening phase using the edge rating *inner_outer* and the time spent by KMETIS and PARMETIS. All times exclude the time spent in initial partitioning. For k processes, the graph has $2^{14+\log k}$ vertices.
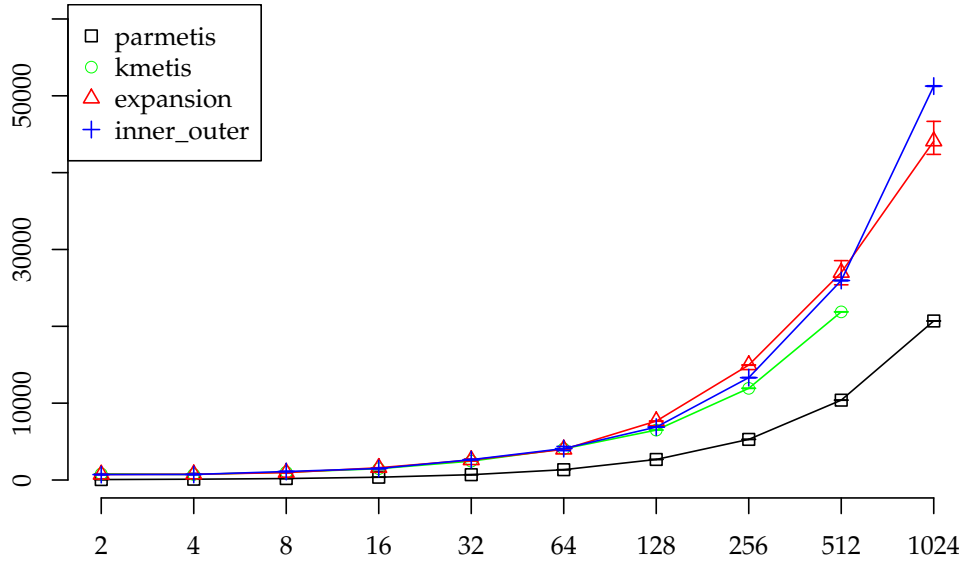
(a) Number of coarsest vertices.
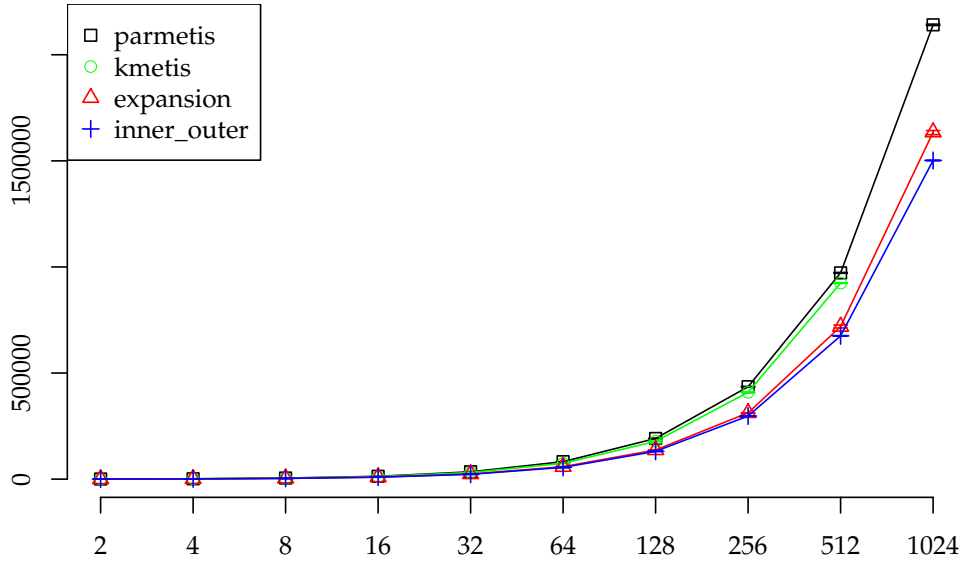
(b) Initial edge cut.

Figure 6.35: Number of coarsest vertices and initial edge cut for KMETIS, PARMETIS and our *inner_outer expansion* variants on the random geometric graphs.
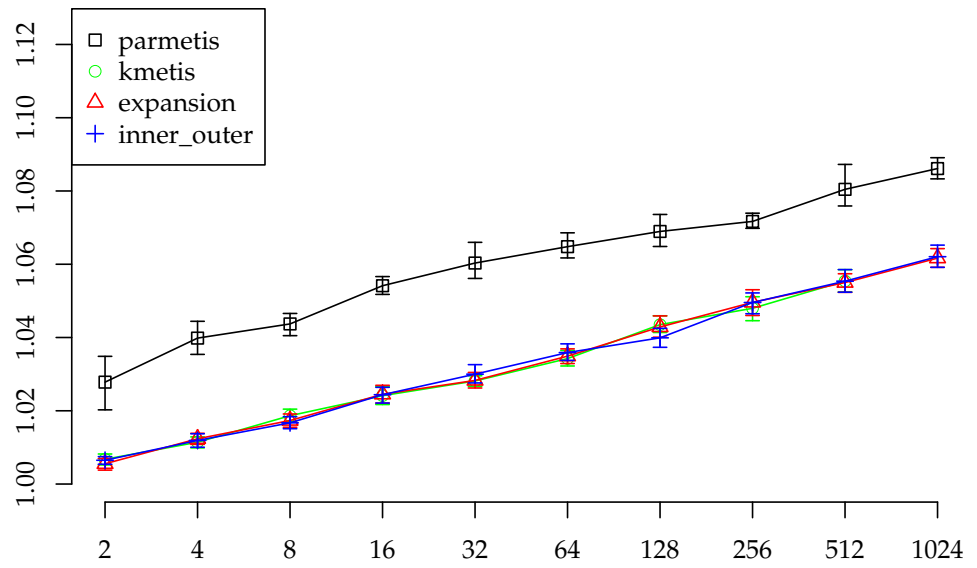
Figure 6.36: Initial edge cut for KMETIS, PARMETIS and our variants *inner_outer* and *expansion* on the random geometric graphs.

7. Discussion and Future Work

7.1. Discussion

We have presented a scalable parallel coarsening phase for a parallel multi-level graph partitioner.

For the local edges, we use the GPA ([49]) approximate maximum weight matching algorithm that yields high quality matchings. Instead of using the "classic" objective function *weight* to optimize in the matching algorithm, we proposed to use other ones we call *edge ratings*. We presented the edge ratings *inner_outer* and *expansion* that yield better initial partitions in terms of edge cut and balance than the other edge ratings we proposed. They also yielded better initial edge cuts than KMETIS in many cases.

Note that the comparison of the initial partition of our algorithm with the one of KMETIS in terms of edge cut might be biased by a higher coarsest vertex count of KMETIS. A direct comparison can only be made when using the coarsening phase in a complete graph partitioner (also see below). However, since the coarsest vertex counts yielded by our variants are very close to each other, we are confident that the comparison is valid.

We performed an experimental evaluation of the coarsening phase and compared it with KMETIS and PARMETIS. In terms of initial edge cut, our algorithm yielded better results than KMETIS in many cases when using the edge ratings *inner_outer* and *expansion*. We also considered the running time on large graphs from various classes. Our code requires a lower running time than PARMETIS for large numbers of processes, depending on the graph. The break-even with KMETIS occurs with fewer proceses.

An uncoarsening phase and an integration with the refinement code from [62] has also been implemented. However, because of time constraints, we are not able to present results for a complete graph partitioners yet.

7.2. Future Work

While our graph coarsening algorithm shows good scalability and quality of the initial partition, there are still a lot of open points:

Most prominently, the integration with the refinement phase from [62] should be completed. This will give the possibility to compare the effect of different edge rating variants on the final partition. Also, we expect the resulting parallel graph partitioner to yield high-quality graph partitions both in terms of edge cut and balance.

Our matching algorithm shows relatively good scaling behaviour in our experiments. However, it is a synchronous algorithm and thus suboptimal.

Recently, Pothen et al. proposed an asynchronous matching algorithm for which they could show good scalability for up to 8192 processes in [26]. First, a comparison of our synchronous variant of the Manne-Bisseling matching algorithm with their asynchronous algorithm would be interesting. Second, it would be interesting to integrate their algorithm with our partitioner to improve scalability further.

When partitioning sparse graphs, the gap graph will contain few edges. In these cases, a simpler parallel matching algorithm like the ones described in Section 3.2.6 could be faster because of lower initialization times. This should be investigated. Of special interest is the point where it could make sense to switch from a simple to a more sophisticated algorithm.

Our distributed contraction algorithm also is synchronous. It might be worth investigating a schema that allows overlapping communication and computation.

Another open question is the initial partitioning. We evaluated the sequential implementations PMETIS, SCOTCH and PARTY.

All processes run the same code on the same graph. This means that time is wasted.

On the one hand, it might be good to do the same trick that PARMETIS uses: Each process only evaluates the branches of the recursive bisection algorithm that it needs to compute its part of the initial partition. On the other hand, the initial partitioning algorithm could be run on all processes with different seeds. The best partition of the coarsest graph could then be broadcasted and used in the rest of the program. Sanders proposed this in [61].

We could also consider the DIBAP algorithm for the initial algorithm. While the focus of DIBAP is not on execution speed, it yields high quality partitions. This could be lead to better final partitions and the time/quality tradeoff should be analyzed.

Yet another point for further research is comparing the effect of using different matching algorithms. Instead of GPA, the GREEDY algorithm or even the simple but fast HEAVY-EDGE-MATCHING algorithm could be used.

Inspired by the diffusive methods by DIBAP, it might also be of interest to select the edges to contract using a diffusion or random walk based approach ([61]). There already is an experimental implementation in the sequential code that proved to be very slow. For already strongly coarsened graphs, however, it might be an interesting option.

A. Zusammenfassung

Das Partitionieren von Graphen ist in der Parallelverarbeitung ein wichtiges Problem zur Modellierung von Lastbalancierung. Im einfachsten Fall betrachtet man ungewichtete, ungerichtete Graphen $G = (V, E)$. Die Knotenmenge V ist so in k Blöcke zu partitionieren, dass jeder Block der Partition die gleiche Zahl von Knoten beinhaltet. Dabei ist die Zahl der Kanten (der *Wert des Kantenschnittes*) zwischen Knoten in unterschiedlichen Blöcken zu minimieren.

Das Problem kann auf natürliche Weise auf Graphen mit Knoten- und Kantengewichten erweitert werden.

Da das Problem $\mathcal{NP}$-vollständig ist, werden für reale Anwendungen Approximationsalgorithmen betrachtet. Seit der Mitte der 1990er Jahre basieren die erfolgreichsten Verfahren und Implementierungen auf einem Mehrlevelansatz.

Dabei wird der Graph zunächst durch Kontraktion von Kanten immer weiter verkleinert, man spricht vom Vergröbern des Graphen. Der gröbste Graph wird dann von einem, möglicherweise langsameren, Algorithmus direkt partitioniert. Danach wird der Graph nach und nach wieder verfeinert und die entstehende Partition mittels lokaler Suche verbessert.

In der Literatur werden sowohl sequentielle als auch parallele Algorithmen bzw. Programme zur Partitionierung von Graphen betrachtet.

In meiner Diplomarbeit stelle ich eine skalierbare, parallele Vergröberungsphase für einen parallelen Mehrlevel-Graphpartitionierer vor.

Der algorithmische Schwerpunkt dieser Arbeit liegt auf der Auswahl der zu kontrahierenden Kanten. Es ist üblich, die Kanten mit Algorithmen zur Suche von Paarungen auszuwählen. Dabei wurde festgestellt, dass es vorteilhaft ist, schwerere Kanten auszuwählen. Bisherige Arbeiten verwenden sehr einfache Algorithmen.

Ich benutze in meiner Vergröberungsphase den Approximationsalgorithmus GPA von Maue und Sanders. Für diesen haben die Autoren des Algorithmus gezeigt, dass er bei verhältnismäßig kleiner Laufzeit sehr gute Ergebnisse liefert.

Neben der Verwendung eines starken Algorithmus zum Finden von Paarungen ist die Wahl der Zielfunktion für die Bewertung von Kanten meines Wissens in dieser Arbeit neu. Bisherige Verfahren verwenden das Gewicht von Kanten als Bewertungsfunktion. In meiner Implementierung betrachte ich, wie in der Aufgabenstellung von Peter Sanders vorgeschlagen, andere Zielfunktionen.

Für die Partitionierung des gröbsten Graphen habe ich verschiedene existierende sequentielle Bibliotheken zur Partitionierung von Graphen untersucht. Eine unter ihnen, SCOTCH, liefert die konsistentesten Ergebnisse bezüglich Balance der entstehenden Partition. Diese wird benutzt um den gröbsten Graphen zu partitionieren.

Zusätzlich habe ich eine parallele Verfeinerungsphase implementiert und mit der

lokalen Suche zur Verbesserung von Partition von Christian Schulz kombiniert. Dies geschieht über eine Schnittstelle, die in Zusammenarbeit mit dem Autor der lokalen Suche entstand. Der so entstehende vollständige Graphpartitionierer habe ich aus Zeitgründen noch nicht experimentell auswerten können.

Das Ergebnis der Vergröberungsphase ist ein partitionierter, vergröberter Graph, die *initiale Partition*. Der Kantenschnitt dieser Partition dieses Graphens benutze ich als Metrik für die Qualität der vorangegangen Vergröberung. In einer ausführlichen experimentellen Auswertung habe ich die Qualität der Vergröberung sowie die Skalierbarkeit der Vergröberungsphase betrachtet.

Es stellt sich heraus, dass durch die Wahl der Zielfunktion die Qualität der initialen Partition gegenüber KMETIS verbessern lässt. Ein Teil des Ergebnisses ist auch die Rangfolge der vorgeschlagenen Zielfunktionen nach der Qualität der initialen Partitionen, die mit ihnen erzeugt wurden.

Die Skalierbarkeit habe ich zunächst auf zwei großen Graphen betrachtet. Der erste ist der Graph des Straßennetzes von Europa. Der zweite entsteht aus einer Matrix, die aus einer Anwendung in der Bioinformatik stammt.

Danach betrachtete ich sie auf zwei Familien von Graphen: Zum einem auf Delaunay Triangulierungen und zum anderen auf geometrischen Zufallsgraphen. Zum Erzeugen dieser Graphen habe ich jeweils ein Programm geschrieben.

Ich habe meine Vergröberungsphase mit dem sequentiellen KMETIS und dem parallelen PARMETIS verglichen. Für mittlere Prozessorzahl ist meine Vergröberungsphase schneller als die vom sequentiellen Programm KMETIS auf einem Prozessor. Für große Prozessorzahlen (ab 256 oder 512, je nach Graph) ist sie zudem schneller als die von PARMETIS mit der gleichen Zahl von Prozessoren.

B. Graph Generators

The aim of this thesis was to create a *scalable* coarsening phase for a multilevel graph partitioning algorithm.

One focus was on graphs with geometric points attached to the vertices because these graphs can be preprocessed efficiently. The largest graph in Walshaw's graph partition archive [73] is *auto* with 448 thousand vertices and 3.315 million edges. The largest graph from this set for which we could obtain coordinates is *ocean* with 143 thousand vertices and 410 thousand edges.

This raises the question for large graphs which have coordinates attached to their vertices. For our weak scaling studies, we also needed generators that generate graphs for a given size, i.e. the number of vertices.

In Section B.1, we will describe our generator for random geometric graphs and in Section B.2, we will describe the generator for Delaunay Triangulation graphs.

B.1. Random Geometric Graph Generator

Random geometric graphs are a class of geometric graphs where each vertex is attached to point in an euclidean space. The points are connected if their distance is below a given value d. We are considering random geometric graphs with an equal distribution of points in the unit sqaure $[0, 1] \times [0, 1]$. Figure B.1 shows an example for a random geometric graph.

Unit disk graphs are a type of random geometric graph, too. Here, randomly placed points in the unit disk (disk around $(0, 0)$ with radius 1) are considered. They are considered to model wireless communication systems, e.g. see [31, 34].

In this section, we will identify the terms vertex and point. Without loss of generality, we assume that the points are unique.

Conceptually, the generator is given the number of points n to create and a radius r. It then creates n uniformly random vertices on points in the unit square $[0, 1] \times [0, 1]$. A pair of vertices is connected if their distance is not greater than r.

B.1.1. Algorithm and Supporting Data Structures

A naïve implementation of a unit disk graph generator would compare all point pairs, requiring $\Theta(n^2)$ time. Since we are generating the points uniformly at random, we can use the following improvement (from [61]).

We create a tiling of squares (cells) with side length r as shown in Figure B.2. The number of cells in each row and column is $k = \lceil 1/r \rceil$. The tiling begins in $(0, 0)$ and
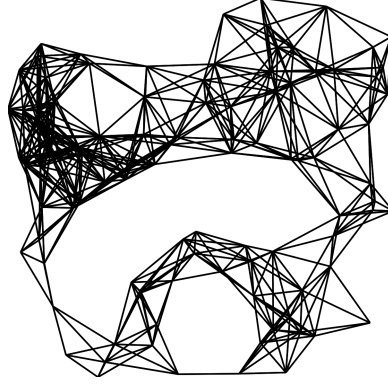
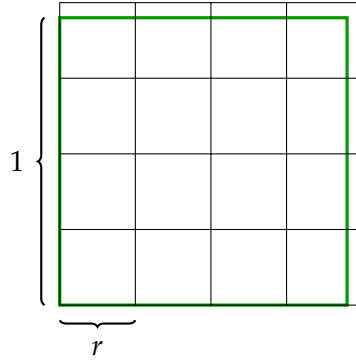Figure B.1: Random geometric graph in the unit square with 100 points and $d = 0.2$.

Figure B.2: The grid and measures. The unit square is shown in green, the grid is shown in black.

overlays the unit square. The cells are numbered $0, \ldots, k^2 - 1$, row-wise from left to right, then bottom to top.

The points are sorted by their cell number in an array *points*. The index of the first point in cell i is given by *point-splitters*. Note that this, again, gives a CSR data structure.

The cell index of a point can be determined in constant time using division and multiplication. After generating n vertices, we can sort them in expected time $\mathcal{O}(n \cdot \log n)$ using Quicksort. Then, we can generate the *point-splitters* in $\mathcal{O}(n + k^2)$ time. Thus, building the grid data structure takes $\mathcal{O}(k^2 + n \cdot \log n)$.

Consider each cell C and consider all neighbouring cells C' of C. Then, compute the distance of each vertex u from C to each vertex v from C'. If the distance is not greater than r then add the edge (u, v) to the set of edges.

Each cell has an expected number of vertices of $\mathcal{O}(n^2/k^4)$. There are k^2 cells. Thus, this step takes expected time $\mathcal{O}(n^2/k^2)$.

This algorithm runs in expected time $\mathcal{O}(k^2 + n \cdot \log n + n^2/k^2) = \mathcal{O}(1/r^2 + n \cdot \log n + n^2 \cdot r^2)$.

[34] states that the random disk graph is almost always connected if $r = c\sqrt{(\ln n)/n}$ if $c > 1.1$. In our experiments, we use random geometrics graph in the unit square and a value for r of $0.55\sqrt{(\ln n)/n}$. The hope here is that the points in the disk around $(0.5, 0.5)$ with radius 0.5 are almost always connected. The points in the square outside should be "somewhat" connected. The graph should have one big component and maybe few smaller components, and some isolated vertices, as we could observe in some of the graphs from Table 6.1.

For this value of r, the expected running time is $\mathcal{O}(n/\ln n + n \cdot \log n + n \cdot \ln n) = \mathcal{O}(n \cdot \log n)$
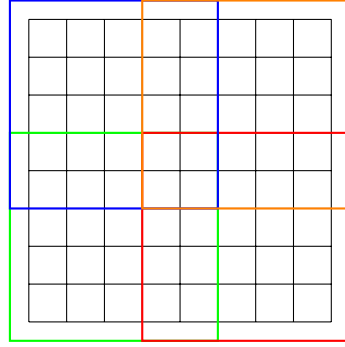
B.1.2. Parallelization

Figure B.3: Distribution of an 8×8 grid to four processes.

The parallelization of the second part of the algorithm described in Section B.1.1 is straightforward, even on a distributed memory machine: Assuming a number $P = p^2$ of processes, each process owns a square of $k/p \times k/p$ cells. If each process also knows the neighbouring cells, it can generate the edges for the vertices in its own cells independently of the other processes.

First, each process generates n/p points. Second, the points are distributed to their owners. Third, each process sorts the local vertices by the cell number they fall into, exchanges the vertex counts with all other processes and is then able to assign globally unique identifiers to all local vertices. Fourth, each process exchanges the vertices in the top, left, bottom and border cells with its neighbour in this direction. Now, each process can generate the outgoing edges for all local vertices independent of all other processes.

The expected time for local processing is $\mathcal{O}(n/p \cdot \log(n/p))$. Additionally, an upper bound on the expected time spent in communication $T_{\text{all-to-all}}(n/p, p) + T_{\text{all-to-all}}(1, p) + 4T_{\text{point-to-point}}(n/(k \cdot p) + 2)$. $T_{\text{all-to-all}}(\ell, c)$ is the time for an all-to-all communication with messages of length ℓ to c communication partners. $T_{\text{point-topoint}}(\ell)$ is the time for a point-to-point communication with a message length of ℓ.

B.2. Delaunay Triangulation Graph Generator

Delaunay triangulations ([11]) are triangulations of point sets that are of interest in computational geometry and various applications such as CAD systems. A triangulation of n points in the plane can be constructed in time $\mathcal{O}(n \cdot \log n)$.

Delaunay triangulations have various interesting properties. For example, the Delaunay triangulation is the dual graph of the Voronoi diagram of a given set of points and thus it is unique. Additionally, it maximizes the smallest angle in all triangles for all triangulations given a point set.

Our generator works as follows: First, we create a set of n random points in the unit square. Then, we are using the Delaunay triangulation construction algorithm from CGAL ([28]) to generate the Delaunay triangulation of the random point set and export the triangulation as a graph.
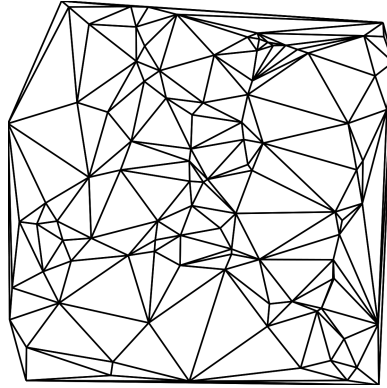

Figure B.4: Example of a Delaunay triangulation of 100 points in the unit square.

Any triangulation fulfills the following property ([11]): Given n points with k points on the convex hull, the number of edges m is given by $m = 3n - 3 - k$. All generated Delaunay triangulations of random point sets we generated had an average degreee of very close to 3. The reason for this is the surface effect: If the points are equally distributed then k is very small against n.

C. Acronyms

This appendix collects the abbreviations used in this thesis.

FM	Fidducia-Mattheyses Heuristic
KL	Kernigham-Lin Heuristic
CSR	Compressed Sparse Row
DT	Delaunay Triangulation
GPA	Global Paths Algorithm
I/O	Input / Output
MPI	Message Passing Interface
PRNG	Pseudo Random Number Generator
RGG	Random Geometric Graph
STL	Standard Template Library

C. Acronyms

D. Parallel Machine and Implementation Details

D.1. Description of the Parallel Machine

This chapter describes the parallel cluster we used.

We ran our experiments on the *IC1* (*Institutscluster 1*) ([66]) at the Steinbuch Centre for Computing, Karlsruhe Institute of Technology.

The IC1 consists of 2 login nodes and 200 compute nodes. Each node has 2 Intel Xeon X5355 with 4 cores clocked at 2.66 Ghz, thus a total of 8 cores per node, and 16 GiB of main memory. We can allocate up to 128 compute nodes and create a parallel machine of 1024 cores and 2 TiB of main memory.

The nodes are connected with a Infiniband 4X DDR interconnect. [66] gives a bandwidth of up to 1 300 MiB/s and a latency below 2 ms for point-to-point communication.

Some dedicated nodes are responsible for the storage system. The maximal read performance per node is given as 600 MiB/s and the total maximal read performance is 6 200 MiB/s.

D.2. Implementation Details

The implementation of our coarsening and uncoarsening phase was written in C++. It makes use of the datastructures of the STL where it makes sense. The sequential and parallel code, including wrappers for partitioning libraries consists of roughly 15 500 lines of code (determined using David A. Wheeler's 'SLOCCount').

We compiled the program using GCC 4.3.1 with the compiler flags `-DNDEBUG -O3`. OpenMPI 1.3.1 ([64, 51]) was used for the parallelization. The program was built using the built tool SCONS. Valgrind ([57]) was a very helpful tool for debugging.

The program writes the metrics to stdout which is written into log files by the machine's job management systems. To ease the extraction of data from these log files, some lines have the special prefix "`log>`" and contain JSON ([9]) encoded information. A short Python script extracts this information and builds a CSV file from this. The CSV file is then imported into R ([60]) for further analysis.

D. Parallel Machine and Implementation Details

Bibliography

[1] Noga Alon. Spectral techniques in graph algorithms. In *LATIN '98: Proceedings of the Third Latin American Symposium on Theoretical Informatics*, pages 206–215, London, UK, 1998. Springer-Verlag.

[2] Jon Louis Bentley. Multidimensional binary search trees used for associative searching. *Commun. ACM*, 18(9):509–517, 1975.

[3] M. J. Berger and S. H. Bokhari. A partitioning strategy for nonuniform problems on multiprocessors. *IEEE Trans. Comput.*, 36(5):570–580, 1987.

[4] M. Blum, R.W. Floyd, V.R. Pratt, R.L. Rivest, and R.E. Tarjan. Time bounds for selection. *J. Comput. System Sci.*, 7(4):448–461, 1973.

[5] S.H. Bokhari, T.W. Crocket, and D.M. Nicol. Parametric binary dissection. Technical Report 93-39, Institute for Computer Application in Science and Engineering NASA Langley Research Center, 1993.

[6] Noureddine Bouhmala. An experimental comparison of different graph coarsening schemes. Technical Report ORWP 99/01, École Polytechnique Fédérale De Lausanne, Chaire de Recherche Opérationelle, 1999.

[7] C. Chevalier and F. Pellegrini. PT-Scotch: A tool for efficient parallel graph ordering. *Parallel Comput.*, 34(6-8):318–331, 2008.

[8] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. *Introduction to Algorithms, Second Edition*. The MIT Press, September 2001.

[9] Douglas Crockford. Javascript object notation. `http://www.json.org`.

[10] Timothy A. Davis. University of florida sparse matrix collection. *NA Digest*, 92, 1994.

[11] Mark de Berg, Otfried Cheong, Marc van Kreveld, and Mark Overmars. *Computational Geometry: Algorithms and Applications*. Springer, Berlin, 3rd edition, April 2008.

[12] Maurice de Kunder. Geschatte grootte van het geïndexeerde world wide web. Master's thesis, Tilburg University, March 2007.

[13] Roman Dementiev, Peter Sanders, Dominik Schultes, and Jop F. Sibeyn. Engineering an external memory minimum spanning tree algorithm. In Jean-Jacques

Lévy, Ernst W. Mayr, and John C. Mitchell, editors, *3rd IFIP International Conference on Theoretical Computer Science (TSC2004)*, pages 195–208, Toulouse, France, 2004. Kluwer.

[14] R. Diestel. *Graphentheorie*. Springer, third edition, 2006.

[15] Hristo Djidjev. A scalable multilevel algorithm for graph clustering and community structure detection. *Workshop on Algorithms and Models for the Web Graph, Lecture Notes in Computer Science*, 2006.

[16] Doratha E. Drake and Stefan Hougardy. Linear time local improvements for weighted matchings in graphs. In *In International Workshop on Experimental and Ecient Algorithms (WEA), LNCS 2647*, pages 107–119. Springer-Verlag, 2003.

[17] Charbel Farhat and Michel Lesoinne. Automatic partitioning of unstructured meshes for the parallel solution of problems in computational mechanics. *International Journal for Numerical Methods in Engineering*, 36(5):745–764, 1992.

[18] C.M. Fiduccia and R.M. Mattheyses. A linear-time heuristic for improving network partitions. *Design Automation, 1982. 19th Conference on*, pages 175–181, June 1982.

[19] P. Fjallstrom. Algorithms for graph partitioning: A survey, 1998.

[20] Harold N. Gabow and Robert E. Tarjan. Faster scaling algorithms for general graph matching problems. *J. ACM*, 38(4):815–853, 1991.

[21] Marco Gaertler. Clustering. *Network Analysis*, pages 178–215, 2005.

[22] Giorgio Gallo and Stefano Pallottino. Shortest path algorithms. *Annals of Operations Research*, 13(1):1–79, 12 1988/12/27/.

[23] Robert Geisberger. Contraction hierarchies: Faster and simpler hierarchical routing in road networks. Master's thesis, Universität Karlsruhe (TH), 2008.

[24] Stefan Goedecker and Adolfy Hoisie. *Performance Optimization of Numerically Intensive Codes*. Society for Industrial Mathematics, 2001.

[25] Michel Gondran and Michel Minoux. *Graphs and Algorithms*. Wiley-Interscience Series in Discrete Mathematics. Wiley, Chichester, 1984.

[26] Mahantesh Halappanavar, Florian Dobrian, and Alex Pothen. Matchings in massive graphs on terrascale computers via approximation. *To appear*, 2009.

[27] Bruce Hendrickson and Robert W. Leland. A multi-level algorithm for partitioning graphs. In *Supercomputing*, 1995.

[28] Susan Hert and Michael Seel. dD Convex Hulls and Delaunay Triangulations. *CGAL-3.2 User and Reference Manual*, 2006.

[29] Vincent Heuveline, Björn Rocker, and Peter Sanders. Personal communication, May 2009.

[30] Jaap-Henk Hoepman. Simple distributed weighted matchings, 2004.

[31] M.L. Huson and A. Sen. Broadcast scheduling algorithms for radio networks. *Military Communications Conference, 1995. MILCOM '95, Conference Record, IEEE*, 2:647–651 vol.2, Nov 1995.

[32] Amos Israeli and A. Itai. A fast and simple randomized parallel algorithm for maximal matching. *Information Processing Letters*, 22(2):77 – 80, 1986.

[33] Amos Israeli and Y. Shiloach. An improved parallel algorithm for maximal matching. *Information Processing Letters*, 22(2):57 – 60, 1986.

[34] Xingde Jia. Wireless networks and random geometric graphs. In *ISPAN*, pages 575–580, 2004.

[35] Stefan E. Karisch, Franz Rendl, and Jens Clausen. Solving graph bisection problems with semidefinite programming. Technical report, INFORMS Journal on Computing, 1997.

[36] George Karypis and Vipin Kumar. Analysis of multilevel graph partitioning. In *Supercomputing '95: Proceedings of the 1995 ACM/IEEE conference on Supercomputing (CDROM)*, page 29, New York, NY, USA, 1995. ACM.

[37] George Karypis and Vipin Kumar. Multilevel graph partitioning schemes. In *ICPP (3)*, pages 113–122, 1995.

[38] George Karypis and Vipin Kumar. Parallel multilevel k-way partitioning scheme for irregular graphs. *SC Conference*, 0:35, 1996.

[39] George Karypis and Vipin Kumar. Parallel multiway graph partitioning. *Proceedings of IPPS '96*, 1996.

[40] George Karypis and Vipin Kumar. A coarse-grain parallel formulation of multilevel *k*-way graph-partitioning algorithm. In *Proc. 8th SIAM Conference on Parallel Processing for Scientific Computing*, 1997.

[41] George Karypis and Vipin Kumar. A fast and high quality multilevel scheme for partitioning irregular graphs. *SIAM JOURNAL ON SCIENTIFIC COMPUTING*, 20(1):359–392, 1998.

[42] George Karypis and Vipin Kumar. Multilevel k-way partitioning scheme for irregular graphs. *Journal of Parallel and Distributed Computing*, 48:96–129, 1998.

[43] George Karypis and Vipin Kumar. A parallel algorithm for multilevel graph partitioning and sparse matrix ordering. *Journal of Parallel and Distributed Computing*, 48:71–95, 1998.

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

[45] Robert Leland and Bruce Hendrickson. An empirical study of static load balancing algorithms. In *Proc. Scalable High-Performance Comput. Conf.*, 1994.

[46] L. Lov'asz and M.D. Plummer. *Matching Theory (North-Holland mathematics studies)*. Elsevier Science Ltd, 1986.

[47] M Luby. A simple parallel algorithm for the maximal independent set problem. In *STOC '85: Proceedings of the seventeenth annual ACM symposium on Theory of computing*, pages 1–10, New York, NY, USA, 1985. ACM.

[48] Fredrik Manne and Rob H. Bisseling. A parallel approximation algorithm for the weighted maximum matching problem. In *In Proc. Seventh Int. Conf. on Parallel Processing and Applied Mathematics (PPAM*, 2007.

[49] Jens Maue and Peter Sanders. Engineering algorithms for approximate weighted matching. In *Experimental Algorithms*, pages 242–255, 2007.

[50] Kurt Mehlhorn and Peter Sanders. *Algorithms and Data Structures: The Basic Toolbox*. Springer, Berlin, May 2008.

[51] Message Passing Interface Forum. *MPI: A Message-Passing Interface Standard, Version 2.1*, June 2008.

[52] H. Meyerhenke, B. Monien, and T. Sauerwald. A new diffusion-based multilevel algorithm for computing graph partitions of very high quality. *IEEE International Symposium on Parallel and Distributed Processing*, pages 1–13, April 2008.

[53] Henning Meyerhenke. Personal communication, July 2009.

[54] G.L. Miller, S-H. Teng, W. Thurston, and S.A. Vavasis. Automatic mesh partitioning. In A. George, J.R. Gilbert, and J.W.H. Liu, editors, *Graph Theory and Sparse Matrix Computation*, volume 56 of *The IMA Volumes in Mathematics and its Applications*, pages 57–84. Springer, 1993.

[55] B. Monien and S. Schambeger. Graph partitioning with the party library: Helpful-sets in practice. In *Proceedings of the 16th Symposium on Computer Architecture and High Performance Computing (SBAC-PAD'04)*, pages 198–205. IEEE Computer Society Press, 2004.

[56] Burkhard Monien, Robert Preis, and Ralf Diekmann. Quality matching and local improvement for multilevel graph-partitioning. *Parallel Computing*, 26(12):1609 – 1634, 2000. Graph Partitioning and Parallel Computing.

[57] Nicholas Nethercote and Julian Seward. Valgrind: A framework for heavyweight dynamic binary instrumentation. In *Proceedings of ACM SIGPLAN 2007 Conference*

on Programming Language Design and Implementation (PLDI 2007), pages 89–100, San Diego, California, USA, 2007.

[58] François Pellegrini and Jean Roman. Scotch: A software package for static mapping by dual recursive bipartitioning of process and architecture graphs. *High-Performance Computing and Networking*, pages 493–498, 1996.

[59] R. Preis and R. Diekmann. *The PARTY Partitioning-Library, User Guide - Version 1.1.* University of Paderborn, Technical Report tr-rsfb-96-024, September 1996.

[60] R Development Core Team. *R: A Language and Environment for Statistical Computing*. R Foundation for Statistical Computing, Vienna, Austria, 2009. ISBN 3-900051-07-0.

[61] Peter Sanders. Personal Communication, April 2009.

[62] Christian Schulz. Scalable parallel refinement of graph partitions. Master's thesis, Universität Karlsruhe (TH), May 2009.

[63] Horst D. Simon. Partitioning of unstructured problems for parallel processing. *Computing Systems in Engineering*, 2:135–148, 1991.

[64] The Open MPI Project. OpenMPI 1.3.1. `http://www.open-mpi.org`, 2009.

[65] Top500.org. Top500 Supercomputing Sites. `http://www.top500.org`, May 2009.

[66] Steinbuch Centre for Computing Universität Karlsruhe (TH). InstitutsCluster User Guide. `http://www.rz.uni-karlsruhe.de/rz/docs/IC/ug/ugic.pdf`, September 10 2008.

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

[68] A. van Heukelum, G. T. Barkema, and R. H. Bisseling. DNA electrophoresis studied with the cage model, 2001.

[69] C. Walshaw and M. Cross. Parallel Optimisation Algorithms for Multilevel Mesh Partitioning. *Parallel Comput.*, 26(12):1635–1660, 2000.

[70] C. Walshaw and M. Cross. JOSTLE: Parallel Multilevel Graph-Partitioning Software – An Overview. In F. Magoules, editor, *Mesh Partitioning Techniques and Domain Decomposition Techniques*, pages 27–58. Civil-Comp Ltd., 2007. (Invited chapter).

[71] C. Walshaw, M. Cross, R. Diekmann, and F. Schlimbach. Multilevel mesh partitioning for optimising domain shape. Technical report, INT. J. HIGH PERFORMANCE COMPUT. APPL, 1998.

[72] C. Walshaw, M. Cross, and M. G. Everett. Parallel dynamic graph-partitioning for unstructured meshes. Tech. Rep. 97/IM/20, Comp. Math. Sci., Univ. Greenwich, London SE10 9LS, UK, March 1997.

[73] Chris Walshaw. Partition Archive. `http://staffweb.cms.gre.ac.uk/˜c.walshaw/partition/`, April 2009.