Technical University of Munich

Department of Informatics

Bachelor's Thesis in Informatics

Evaluation of Graph Partitioning Algorithms for Load Balancing of Earthquake Simulations

David Jonathan Schneller

Technical University of Munich

Department of Informatics

Bachelor's Thesis in Informatics

Evaluation of Graph Partitioning Algorithms for Load Balancing of Earthquake Simulations

Evaluierung von Graphpartitionierungsalgorithmen für die Lastbalancierung von Erdbebensimulationen

Author	David Jonathan Schneller
Supervisor	Prof. Dr. Michael Georg Bader
Advisor	Carsten Uphoff, M.Sc.
Date of Submission	March 15, 2019

Eidesstattliche Erklärung

Abstract

Finite element methods can be used to discretize a given partial differential equation. This usually results in the domain model being transformed into a mesh made of polyhedral cells. When simulating the equation on the mesh on a computer cluster, load balancing helps to push the maximum out of the hardware. In order to do so, one can transform the mesh into a graph which is then partitioned. As the so-called graph partitioning problem is computationally hard to solve exactly, many heuristics have been developed for it, and some libraries with algorithms for this problem are publicly available. We examine the partitioning libraries ParMETIS, PT-SCOTCH and ParHIP for using them to load balance the computations of the earthquake simulation program SeisSol. For this, we present an overview over common partitioning libraries, and then present an interface for the afore-mentioned libraries which is integrated into the mesh-reading library PUML. Furthermore, we show how the transformation from mesh to graph in our case can be accomplished without using an external library, essentially reducing the number of dependencies of our software. To solve this problem, we developed an abstraction to iterate over all faces in a given mesh, possibly distributed over multiple compute nodes. We close with an evaluation of ParMETIS, PT-SCOTCH and ParHIP; mostly for the case that we have vertex weights, but no edge weights; conforming with the currently used graph weighting model in SeisSol.

Acknowledgments

I would like to thank my parents for their continuous support during my whole studies; as well as my advisor and supervisor for their help and assistance with any questions I had while writing this thesis. Furthermore, my thanks go to the Hanns Seidel Foundation (Hanns-Seidel-Stiftung) for their financial support, funded by means from the Federal Ministry of Education and Research (Bundesministerium für Bildung und Forschung).

Contents

1 Introduction

Load balancing is essential for achieving maximum performance on a given computer cluster. However, it is sometimes difficult to achieve. Say, we want to simulate a system described by a partial differential equation (PDE) over some time – which is in the most cases not analytically solvable. We could spatially discretize our domain model using the finite element method, basically transforming the domain into a mesh out of simple polyhedral cells, for example tetrahedra. It might still be computationally intensive to simulate the PDE, so we parallelize the our software with MPI, so that it can run on a cluster. This means that we usually split up our mesh onto the different nodes. We need to communicate the local progress between the nodes, at least between neighboring cells. Unfortunately, we introduce (at least) two problems by this: firstly, if we have to communicate a lot of data, the speed of our application will be limited by the bandwidth of the interconnect. Secondly, if the mesh is split unevenly onto our nodes, some nodes may stall while others are still computing data, before we can advance in time. So, it would be preferable to distribute the mesh as evenly as possible onto the nodes and to minimize the communication volume. A possible solution to these problems is to model the mesh as a graph. Here, we have the so-called (hyper)graph partitioning problem: given some (hyper)graph, we want to find a partitioning of all vertices into a specified maximum number of sets, so that each of the sets has roughly the same size, and that the number of edges which connect two different partitions is minimized. We call the latter the edge cut. Technically though, also other objectives are thinkable. Hence, if we try to model our cells as vertices, and our dependencies between them as edges, we get a graph or hypergraph, depending on the communication model. For cells with different computational effort, we can also weigh our graph to represent that.

Alas, the graph partitioning problem is computationally hard, or rather NP-complete. As a consequence, no exact polynomial-time partitioning algorithm may exist (unless $P = NP$). Yet, many effective heuristics have been developed to counteract this limitation. Not only have many algorithms been published, also several libraries have been written which can compute practically applicable partitions. Generally, these libraries are not only developed for graphs obtained from meshes, but also to partition graphs which have a more complex structure. So the question arises on how well the libraries perform on mesh data – and which are best to be used to partition a mesh for actual simulations or computations. Additionally, our graphs may be very large and we may want to compute partitions during runtime. So our libraries must be able to work to some degree in parallel to achieve results in a sufficiently short time. This means that in addition to the partition quality, i.e. the resulting edge cut and imbalance, the time and memory it takes to partition a mesh are important metrics. Popular libraries include METIS [1] and its distributed memory variant ParMETIS [2], as well as SCOTCH and

PT-SCOTCH [3], not to mention all the hypergraph partitioners like PHG in ZOLTAN [4] and PaToH [5] or discontinued partitioners like Chaco [6], Party [7], or Jostle [8]. Moreover, there emerge new libraries from time to time like KaHIP [9] which recently got enhanced by a distributed memory variant, called ParHIP.

This Bachelor's Thesis is going to examine different graph partitioning libraries and their partitioning quality and performance impact on earthquake simulations done with the software SeisSol. The program uses unstructured tetrahedral meshes [10] for representing earthquakes, and is parallelized in a hybrid scheme using OpenMP and MPI. This means that we are in the situation as described above, and SeisSol uses ParMETIS as of now [11] to find a suitable distribution of the mesh onto a cluster. Furthermore, SeisSol weights its vertices, but not its edges [11] (or rather: all edges are uniformly weighted). The interface to the partitioning library is currently implemented in the library PUML [12], a library to read unstructured meshes, and it is tailored completely on ParMETIS. This goes so far that ParMETIS even performs the transformation of a mesh to a corresponding graph; aside from partitioning. In the following, we will determine, if there is another, better-suited partitioning library and we will generalize the interface in PUML to easily switch between different partitioners.

This thesis is structured as follows: after introducing some basic notation, we begin with explaining where SeisSol needs and makes use of graph partitioning (chapter 2). After that, we recapitulate the graph partitioning problem itself and present the currently used approaches used to tackle the problem (chapter 3). Then, we provide a quick survey of graph partitioning libraries, describe some candidates (chapter 4) and present how to implement them in SeisSol (chapter 5). We conclude with benchmarks on tetrahedral meshes (chapter 6) as well as remarks for future work in the area (chapter 7).

2 Mesh Processing

To begin, we examine where we need graph partitioning in SeisSol in particular.

2.1 SeisSol

SeisSol is an open source earthquake simulation software developed at the Technical University of Munich and the Ludwig-Maximilian University of Munich [13]. The source code is available here [14]. It was used to create a large-scale simulation of the 1992 Landers earthquake [15] (Gordon Bell finalist paper 2014) and the 2004 Sumatra earthquake [10] (winner of the Best Paper Award at SC'17). Internally, SeisSol models earthquake areas as unstructured tetrahedral meshes. For temporal discretization, the program uses the ADER-DG method (see for example [16]). The solution is approximated using a polynomial of high order (e.g. 5 in [10]).

2.1.1 Computations

The required earthquake calculations consist mainly of two types.

First, we have the propagation of seismic waves (cf. [11]). This process consists of the wave propagation inside a cell as well as the transfer of waves between adjacent cells. The inner-cell propagation can be done fully independently from other cells, whereas the surface transfer needs the data from all (i.e. 4 at maximum) neighboring cells.

Additionally, there is the computation of dynamic rupture effects. This concerns the resulting seismic waves from the friction between two different bodies, i.e. the surface between them matters. In SeisSol, it is implemented that cell faces can be marked as dynamic rupture faces. Generally, only a small portion of faces is marked as such [10].

2.1.2 Time stepping

As described in [11], each of the cells has a different maximum possible time step, so that the simulation still stays numerically stable. It is hard to optimize, if we set the time step for each cell individually. But we also lose too much performance, if we use a single global time step for that the simulation stays numerically stable everywhere.

A compromise (resulting in a *local time stepping*, or LTS, scheme), as presented in [11], is to cluster the cells by the maximum allowed time step. Let r be an integer parameter (in SeisSol, r is often set to 2). We then build clusters $C_1, ..., C_N$ with cluster C_i containing all cells whose maximum time step is in the range $[r^{i-1}\Delta t_{min}, r^i \Delta t_{min})$ (with Δt_{min} being the smallest maximum time step over all cells). The cells in C_i are then updated with rate $r^{i-1}\Delta t_{min}$. Dynamic ruptures can be incorporated into this scheme as well [10].

2.1.3 Parallelization

Due to the sheer size of the earthquake meshes, they require a lot of computational power. Therefore, distributing the calculation onto a cluster of processors is inevitable. As stated in [10], the simulation for 500 seconds of a mesh with about 221 million cells would take 13.9 hours on the SuperMUC, Phase 2 (86016 Haswell cores).

SeisSol uses a hybrid parallelization scheme, combining MPI with OpenMP. In practice that means that we usually have one MPI process per node or CPU, and OpenMP threads to fill the rest of the cores. Sometimes, it is also beneficial to avoid one or two cores on a CPU to give the operating system a place to not interfere with the computation (cf. [10]).

The LTS as described before can be parallelized as follows: in one MPI process, all local element and dynamic rupture face integration is processed in parallel with OpenMP. For multiple MPI processes, each rank keeps a copy layer of cells from other ranks which are adjacent to rank-local cells. Likewise, each rank has a ghost layer which contains its cells which are adjacent to at least one cell from other ranks. The exchange of copy and ghost layer is done with non-blocking MPI functions, i.e. `MPI_Isend`, `MPI_Irecv` and `MPI_Wait`. To deal with the different time steps and the necessary synchronization, the cells are grouped into work items which then get scheduled [11]. For more detailed information, we refer to [11]. The dynamic rupture faces are also classified into time clusters, for more information here, see [10].

In total, this means that we always transfer whole cells to other MPI ranks.

2.1.4 PUML

As of today (2019), the meshes for SeisSol are stored in a subset of the XDMF format [17] which can be read using the library PUML [12]. The mesh file only stores information about the location of the vertices (coordinate in 3-dimensional space) and which vertices belong to which cells. For SeisSol, additional information about boundary conditions and element groups is saved. PUML reads in the vertex and cell information over all MPI ranks, so that each rank gets a roughly equal share of vertices and cells each. Faces and edges are generated by PUML. On command, PUML re-distributes it so that for each cell, all its vertices are present locally (this means of course, that a vertex may be present on multiple MPI ranks). Additionally, PUML generates faces and edges out of the vertices and assigns each cell, face, edge and vertex a global ID (unique over all MPI ranks). When given a partition assignment for the cells, PUML re-distributes the cells and the generates faces, edges etc. After that process, PUML can infer for geometric objects the associated higher- or lower-dimensional ones. For example, for a given face, PUML can infer the incident cells or the associated mesh edges or mesh vertices. An interface to ParMETIS which can be used for partitioning is also present in PUML, but not tied to the rest of the library.

Further extensions or optimizations for PUML are discussed in [18].

2.2 Notation

To work with graphs, we have to introduce a little bit of mathematical notation.

For a graph $G = (V, E)$, V denotes the set of vertices and E the set of edges. Let $n := |V|, m := |E|$. We only consider undirected graphs, i.e. we write the edges in the form $\{u, v\} \in E$, with $u, v \in V$. Moreover, we may give each vertex a weight $c : V \to \mathbb{R}_0^+$, and each edge as well: $\omega : E \to \mathbb{R}_0^+$. We may also freely extend ω to map from the set of all two-element subsets of V to $\mathbb{R}_0^+$ by setting $\omega(\{v, w\}) = 0$ for $\{v, w\} \notin E$. For $V' \subseteq V$, we may conveniently write $c(V') := \sum_{v \in V'} c(v)$, for $E' \subseteq E$ analogously $\omega(E') := \sum_{e \in E'} \omega(e)$.

We call an edge $e \in E$ *incident* to some vertex $v \in V$, if $v \in e$ (i.e. one end of e ends in v). Two vertices $v, w \in V$ are *adjacent*, if $\{v, w\} \in E$ (i.e. there exists an edge which connects v with w).

A k-partition is then a disjoint partition of V into k sets $V_1, ..., V_k$, i.e. $V_i \cap V_j = \emptyset$ for $i \neq j$ (i.e. the sets are pairwise disjoint) but $V_1 \cup ... \cup V_k = V$. We call $\omega(V_1, ..., V_k) := \omega(\{\{u, v\} \in E | u \in V_i, v \in V_j, i \neq j\})$ the edge cut of the partition, that is, the sum of the weights of all edges which are incident to vertices from two different partitions.

We also define a hypergraph $G = (V, E)$, with V as for a “simple” graph, and E being a (general) subset of the power set of V, with $|e| > 1$ for $e \in E$. Every graph is also a hypergraph.

2.3 Terminology

In the Bachelor’s Thesis, we are going to work with (unstructured) tetrahedral meshes. We will refer to the tetrahedra of our mesh as cells, as it is done within PUML (cf. subsection 2.1.4). Each cell has four faces and each face is shared with at most one other cell. Additionally, a cell has six edges, and four vertices. Of course, these edges and vertices can be shared by a multiple (not limited) number of other cells. Each vertex has a three-dimensional coordinate in space.

The so-called *dual graph* of the mesh is defined as follows: for each cell in our mesh, we insert a vertex into the dual graph. We connect to vertices in our dual graph, if the corresponding cells share a face. An example is shown in figure 2.1. Motivated by this description, we may also extend the notion of *incident* and *adjacent* on cells and faces.

To prevent confusion when using the term “vertex” and “edge” for both the mesh, the dual graph of the mesh and an arbitrary graph, we will from now on write “graph vertex”, “graph edge” and “mesh vertex”, “mesh edge” respectively, if the usage of the terms is not clear.

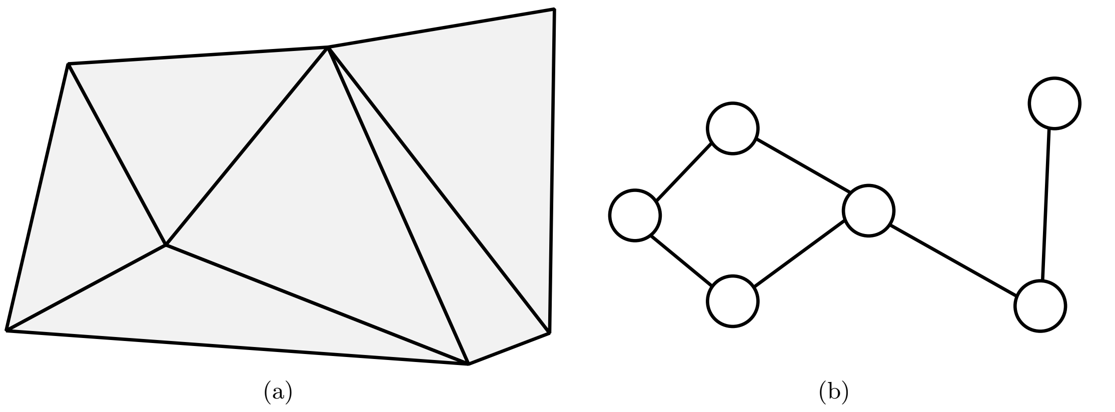

Figure 2.1: A two-dimensional example of (a) a mesh and (b) its dual graph. Note that in the two-dimensional case, mesh edges and mesh faces coincide; hence *here* we insert a graph edge for every mesh edge.

2.4 Problem Setting

The question we want to answer in this Bachelor's Thesis is now to map our n cells onto K interconnected nodes. Of course, we want to optimize the distribution so that the general runtime is reduced. We try to accomplish this by formulating two goals:

- Minimize the maximum computing time per node
- Minimize communication between the nodes

To solve this, the problem can be modeled (as it is done in SeisSol) with the dual graph of the mesh. As SeisSol only communicates values of cells over MPI, we can interpret this as sending data from one dual graph node to another dual graph node (a dual graph node equals one cell), if they are adjacent (i.e. the two cells share a common face in our mesh). The only thing that is left is to account for the different computational weights of our cells, introduced by the LTS scheme and dynamic ruptures. Let $v \in V$ be a dual graph vertex, let C_i be the cell corresponding to v. First, we deal with wave propagation: we observe that it only matters to look at the number of updates the cell does until we reach the time $r^N \Delta t_{min}$. This number is exactly r^{N-i}. The reason this is accurate enough is that at $r^N \Delta t_{min}$, all clusters are synchronized again (and again at integer multiples of that time). As each cell update is equally expensive, setting this weight for wave propagation is accurate enough. For the edge weights of the graph, we set them all uniformly to 1. This is actually how it is implemented in SeisSol as of now [14]. Dynamic ruptures are handled as follows: for each dynamic rupture face, we add αr^{N-i} to the weight, where α is an integer parameter which approximates the ratio of the computational code for dynamic rupture calculations divided by the wave propagation. This may not be the most accurate model possible for these, but it works good enough in practice [14].

With this knowledge, we can then re-formulate our goals: our first goal then equals to finding a K-partition, so that $c(V_i) \approx \frac{c(V)}{K}$ for homogeneous clusters. For heterogeneous clusters, this generalizes to $c(V_i) \approx w_i c(V)$ with $\sum_{i=1}^{N} w_i = 1, w_i \geq 0$. The second goal means to minimize the accumulated weight of edges whose endpoints lie in different V_i, i.e. minimize $\omega(V_1, ..., V_K)$ (the edge cut).

To summarize: our re-formulations transformed our load balancing issue into a graph problem, namely the graph partitioning problem (with minimizing the edge cut as objective).

3 The Graph Partitioning Problem

We define the graph partitioning problem as follows: given a weighted Graph $G = (V, E)$ (both vertex- and edge-weighted), minimize a given function $f(V_1, ..., V_K)$ subject to $c(V_i) \leq w_i c(V)(1 + \epsilon)$, by partitioning the graph in at most K different parts $V_1, ..., V_K$. We call $\epsilon \geq 0$ the allowed imbalance. A small non-zero imbalance value helps (e.g. $\epsilon = 0.01$) to possibly achieve a lower objective value while not significantly worsening the possible imbalance. In the following, we will solely use the edge cut as objective function, i.e. $f(V_1, ..., V_K) = \omega(V_1, ..., V_K)$.

3.1 Complexity

The graph partitioning problem, with minimizing the edge cut as objective, is NP-complete. This follows, for example, from [19]. We look at an unweighted (i.e. $\omega(e) = 1$ for $e \in E$, $c(v) = 1$ for $v \in V$) graph. The authors prove that determining if there exists a 2-partition V_1, V_2 with $|V_1| = |V_2|$ (or $c(V_1) = c(V_2)$) with an edge cut of at most W (with W being a positive integer), i.e. $\omega(V_1, V_2) \leq W$, is NP-complete. Any given algorithm to solve the graph partitioning problem exactly as we defined it could therefore be used to solve this problem, with $K = 2$, uniform vertex and edge weights, $w_1 = w_2 = \frac{1}{2}$ as well as $\epsilon = 0$. Additionally, there cannot exist a polynomial-time approximation algorithm with finite approximation ratio for $K \geq 3$ to find a K-partition with $|V_1| = ... = |V_K|$, except if $P = NP$ was to hold [20]. So far, these results have only spoken about general graphs. Yet, even for simpler-structured graphs which are composed of grids or trees, there generally cannot exist such an algorithm [21]. Therefore, the case for the dual graphs of meshes we view here, it most likely also hopeless.

3.2 Heuristics

Several heuristics have been developed which compute practicable partitions or improve existing ones. As there is a vast amount of algorithms and approaches for the graph partitioning problems, we will refer to e.g. [22] for a general overview.

3.2.1 Multi-level Partitioning

Because the graph which are to be partitioned can be very large, it is often helpful to reduce the size of the graph, and partition then. The most common approach for this is multi-level graph partitioning. This method is used by many popular partitioning libraries today, e.g. [23, 24]. Because of this, we will describe it here in abstract form. It could be described as rather a meta-heuristic, as it makes use of (at least) another graph partitioning algorithm as well as partition refinement algorithms (i.e. improving existing partitions), and to some extent also use of graph clustering (i.e. grouping vertices of a graph in some way). The basic idea goes as follows [25]:

- *Coarsening*: We take our original graph G and collapse some of the edges (process is known as coarsening). For each edge $e = \{u, v\}$ we want to collapse, we remove e, u, v from the graph and insert a new vertex uv with $c(uv) = c(u) + c(v)$. Every edge which connected either to u or v is changed to connect to uv instead. We set $\omega(\{x, uv\}) = \omega(\{u, x\}) + \omega(\{v, x\})$, if $x \in V$ was connected to both u and v. Which edges to collapse depends on the used heuristic. This process may now be repeated several times until the resulting graph G' is small enough.

- *Initial Partitioning*: Then, we partition the resulting coarsened graph with a more expensive partitioning algorithm.

- *Un-coarsening and Local Refinement*: Next, we reverse our coarsening steps: each vertex now belongs to a partition. If we now split some uv into u and v, both u and v will belong to the same partition as uv did. Consequently, the edge cuts and the sizes of the partition stay the same during this split-up. We also allow for nodes to be moved to other partitions or to be exchanged with other nodes to improve the edge cut. Again, we use heuristics for doing this. In the end, we will receive a partition on G.

The multi-level method can also be used for partition refinement, e.g. if cut edges from the partition are not contracted during coarsening. This way, the contracted graph still retains the structure of the existing partition.

Moreover, to reduce the influence of randomness (as many heuristics rely on a random influence), the multi-level approach can be applied multiple times, trying different random seeds [26]. Most of the time, cut edges are not contracted any more, as with partition refinement. Such re-running of the multilevel method is called “V-Cycles”. In some variants, multiple seeds per coarsening layer are used [27] (the techniques are called “W-Cycles” and “F-Cycles”).

3.2.2 Sample Algorithms for the Multi-Level Partitioning Scheme

Because the multi-level is used in many places, there also exists an abundance of algorithms for each step. Therefore, we will name a few of them. We will outline some of them here shortly. Of course, this list of far from being exhaustive.

Coarsening

For coarsening, any algorithm which is used for clustering data can be used here. For example, the so-called “label propagation algorithm”, described in [28] which tries to find dense subsets in a graph which can then be coarsened to a single node [27].

Another approach is to use a matching, i.e. a subset of edges M, so that for each $v \in V$, there exists at most one $e \in M$ with $v \in e$. The idea is then to coarsen every edge in M. A common heuristic for example is the heavy edge heuristic, where edges with a larger weight are matched first (greedily) [29].

Initial Partitioning

When the coarsest graph has been obtained, other heuristics are applied. In some cases, another partitioning library is called in a program. In some other cases, another multi-level scheme is employed [30]. Generally, all algorithms used for initial partitioning could also be used for partitioning the original graph – but they may be very slow doing so.

Apart from that, there are also more expensive graph partitioning algorithms which can be used. For example, there exist methods which work with eigenvectors of the Laplacian matrix of a graph (called spectral methods) [29]. Furthermore, a strategy called graph growing can be used which starts with random vertices (as "seeds") and then iteratively adds vertices around them to the partition. Again, here there exist multiple strategies on how to choose the random vertices themselves and how to choose the surrounding vertices to add. Furthermore, partitioning methods could also make use of coordinates, if the vertices have some. In the case of our dual graph, we could for example take the barycenter as the coordinate of each graph vertex in space. Then, partitioning using hyperplanes or space-filling curves (see e.g. [31]) are thinkable.

Local Refinement

For local refinement, algorithms based on the idea of Kernighan and Lin are most common [32]. In this algorithm (here only shown for $K = 2$, and $|V_1| = |V_2|$), pairs of vertices from the two partition parts are chosen, so that when they are swapped, they improve the edge cut the most. This is done until no vertices are left, then the algorithm can be applied several times to improve the edge cut. Unfortunately, each pass takes $\mathcal{O}(n^2 \log n)$ operations. An improvement to linear time per pass, called the Fiduccia-Mattheyses algorithm, is described in [33]. Here, single vertices are moves instead of pairs and more sophisticated data structures are used. Even further extensions exist, see for example [29] or [27].

Another way to improve the edge cut is to only look at the vertices at the border of the partition parts (i.e. where there are edges whose incident vertices belong to two different parts) [34]. Also a possibility is to use maximum-flow-minimum-cut based methods, as described in [27].

The label propagation algorithm can also be used for local refinement [24].

4 A Survey of Graph Partitioning Libraries

In the following, we will compare some popular graph partitioning libraries. We also include hypergraph partitioners in our comparison, as regular graphs are simply a special case of these. First, we formulate some criteria which the libraries have to fulfill at least:

- The library should be actively developed. Some libraries we discovered (and that were referenced in several papers) are not available for public download any more or only accessible through sites like the Web Archive. In case of a sudden disappearance of a library, SeisSol risks to have an important component missing, therefore we will avoid such which are clearly not enhanced or fixed any further.
- The license must be compatible to SeisSol, i.e. to the Revised BSD License (3-clause BSD) (cf. [14], check e.g. `src/SeisSol.cpp`). Of course, the library should be open source in the first place. This way, it can be ensured that no portability problems arise when running SeisSol on more unknown or newer processor architectures.
- The library has to support MPI, or some other interface for distributed memory computation. As the graphs we view may be large and weighted, even local threading may not be fast enough.

The comparison will therefore be split up into three parts: Firstly, we look at the general state of development and maintenance activity done for the libraries. Secondly, we will review the partitioners themselves, and their basic features, such as MPI support or the existence of a C++ interface. Lastly, we examine the performance claims by the authors of the library, both in terms of partitioning speed and output mesh quality. Here, we do not eliminate any library any more, this is rather for a comparison between them in advance to our own tests. We chose not to reproduce the performance results in advance by ourselves as this process would be to time-consuming, and does not focus on our actual work.

4.1 State of Development

Table 4.1 shows a list of graph partitioners with links to their respective websites. We do not claim that the table covers every existing and used graph partitioning library, yet it should cover most of the popular libraries. This means, we will examine, if they are still actively developed and have a license which is compatible to SeisSol.

First, we have the METIS family, including METIS (single-threaded), mtMETIS (OpenMP), ParMETIS (MPI), and hMETIS (hypergraph partitioning). It is developed at the University of Minnesota [1], [2], [40]. In the case of METIS, mtMETIS

Name	Latest Version	Last Release	Active Development	License	Webpage	Source
METIS	5.1.0	2013-03-30	?	Apache 2.0	[1]	[35]
mtMETIS	0.6.0	2013-03-30	?	MIT	[1]	[36]
ParMETIS	4.0.3	2013-03-30	?	Custom[37]	[2]	[38]
hMETIS	2.0pre1	2007-05-25	?	Custom[39]	[40]	closed source
KaHIP	2.10	2019-01-04	✓	MIT	[9]	[41]
ParHIP	2.10	2019-01-04	✓	MIT	[9]	[41]
SCOTCH	6.0.6	2018-07-31	✓	CeCILL-C	[3]	[42]
PT-SCOTCH	6.0.6	2018-07-31	✓	CeCILL-C	[3]	[42]
Zoltan	3.83	Jan. 2016	✓	3-clause BSD	[4]	[43]
KaHyPar	1.0.0	2019-02-01	✓	GPLv3	[44]	[45]
Parkway	2.11	2006-04-08	✗	Unknown	[46]	[47]
PaToH	3.0	2006-12-05	✗	Unknown	[5]	closed source
Jostle	3.0	2002-07-08	✗	Unknown	[8]	Unavailable
Chaco	2.2	2000-05-04	✗	LGPLv2.1	[6]	[48]
Party	1.99	after 1996-9-16	✗	Unknown	[7]	Unavailable

Table 4.1: A (probably non-exhaustive) list of existing graph partition libraries

and ParMETIS, the last release happened in 2013. In the case of hMETIS, it is 2007. hMETIS is a closed source product, hence we will not review it further. The bug tracker included in the webpage (e.g. [2]) does not include many new items for years. We therefore cannot really state, if there is active development going on right now. Concerning the license of the libraries, it allows the usage for "research purposes by non-profit organizations" [37] in the case of ParMETIS – which applies to SeisSol. The license of METIS is Apache 2.0 which is compatible to SeisSol.

Next, we have the KaHIP package, developed at the Karlsruhe Institute of Technology and the University of Vienna [9]. KaHIP, which includes a distributed memory partitioner ParHIP in its latest version, was updated in 2019 (version 2.10, sometimes also referenced as 2.1). In their referenced papers [24, 49], the authors speak of future extensions (which they plan to implement), therefore, we also consider this library to be in active development. The license of KaHIP has been changed from GPL to MIT with version 2.10 which is of course compatible to SeisSol.

As the next partitioning library, we consider SCOTCH and PT-SCOTCH [3], developed at Laboratoire Bordelais de Recherche en Informatique at the University of Bordeaux 1. Both are still actively maintained; the last release happened in 2018 [42]. The CeCILL-C license is compatible with the code of SeisSol having a different license, hence we can use both SCOTCH and PT-SCOTCH.

ZOLTAN is a package with a considerable amount of interfaces to other partitioning libraries [4] which has been created at the Sandia National Laboratories. The last release happened in 2016, we still consider this to be current enough to speak of active development, though it seems to stall. License-wise, ZOLTAN is shared under the 3-clause BSD which matches nicely with the license of SeisSol.

KaHyPar is a serial hypergraph partitioner, currently actively developed [44], the first release happened in 2019. Again, it is from the Karlsruhe Institute of Technology. It is licensed under the GPL, version 3 which may cause problems, as SeisSol is not licensed under this one.

Concerning the rest of the libraries in the table: Party, Jostle and Chaco are not developed any further. The same holds for the hypergraph partitioners Parkway and PaToH. For Chaco, the latest version is more than 20 years old. Jostle has been taken down by its owner, only the manuals are still downloadable on the webpage [8]. Instead, Jostle seems to have been succeeded in some areas by a proprietary program called NetWorks which we will not review any further due to its closed source nature and because it does not cover our purpose. In the case of Party, Parkway, and PaToH, the webpage itself was only available through the internet archive. Therefore, we will not review Party, Jostle, Chaco, Parkway and PaToH in the following sections.

4.2 Feature Comparison

From now on, we will compare the partitioners, not the libraries any more. In many cases, only one partitioner is packed into a library, but this is not the case for ZOLTAN and KaHIP. KaHIP contains five partitioners: KaFFPa, KaFFPaE, KaBaPE, ParHIP, and an edge partitioner (which partitions by edges instead of vertices) since version 2.10. We will not review the edge partitioner, since it is out of scope for us. ZOLTAN includes some basic geometric methods, and also an own (hyper)graph partitioner called PHG – all are parallelized with MPI. We do not count the interfaces to ParMETIS and PT-SCOTCH here which ZOLTAN also contains, as both are autonomous partitioning libraries themselves.

Table 4.2 shows an overview over the most basic functions of all partitioners. For this, we solely compare their support for MPI or threading, as well as the existence of a C/C++ interface.

Of the viewed partitioners, METIS, mtMETIS, KaFFPa, SCOTCH, PaToH and KaHyPar did not support MPI, therefore we are not going to review them further. Though it should be noted that mtMETIS supports OpenMP. We also decided not to look further into KaFFPaE and KaBaPE. While both support MPI, they use it to partition the graph on each rank without contracting it first, leading to a run-time at least as long as when using a non-MPI algorithm [49]. Interestingly enough, SCOTCH, PT-SCOTCH did support both MPI and pthreads, making it the only partitioner to do so; yet the pthread support is still very basic, but going to be improved in future versions [50]. This ultimately leaves ParMETIS, ParHIP, PT-SCOTCH and the algorithms from ZOLTAN. Further, we present a short overview over the advantages of the remaining partitioners:

- ParMETIS can weigh the different partition parts (the weights were referred to as w_i in chapter 3). Moreover, it supports multiple vertex weights which it tries to balance at once. ParMETIS contains some methods for re-partitioning as well. ParMETIS can also handle FORTRAN arrays which start at index 1 instead of 0.

- ParHIP has the least features up to now, given it is still in development. It still lacks a proper C/C++ interface; instead it has only a command line program supplied in the git repository [41].

- PT-SCOTCH also supports weights for different partition parts. Apart from this, is offers fine-grained control over its partitioning algorithm at run-time, using so-called strategy strings. Users also may extend parts of the implemented multi-level scheme with own algorithms. PT-SCOTCH contains routines for ordering a given graph besides for partitioning it. PT-SCOTCH can handle FORTRAN arrays as well, like ParMETIS.

- ZOLTAN (speaking of the whole package) supports a variety of graph partitioning algorithms, with the afore-mentioned PHG and the geometric methods. Apart from that, it contains interfaces to ParMETIS, SCOTCH, PT-SCOTCH and PaToH. Furthermore, ZOLTAN offers methods for re-partitioning or re-distributing a graph during run-time. ZOLTAN contains an interface in FORTRAN, C, as well as a class-based interface for C++.

All the partitioners left support 64-bit wide integers for vertex weights and indices.

Name	C/C++ Interface	Threading	MPI
METIS	✓	✗	✗
mtMETIS	✗	✓	✗
ParMETIS	✓	✗	✓
KaFFPa	✓	✗	✗
KaFFPaE	✗	✗	✗[a]
KaBaPE	✗	✗	✗[a]
ParHIP	✗	✗	✓
SCOTCH	✓	✓[b]	✓
PT-SCOTCH	✓	✓[b]	✓
ZOLTAN-PHG	✓	✗	✓
ZOLTAN-Geom.Alg.[c]	✓	✗	✓
KaHyPar	✓	✗	✗

[a] Technically yes; but practically, it runs several serial partitioners on different nodes [b] Partially
[c] Geometric algorithms implemented in ZOLTAN

Table 4.2: Basic features of the viewed partitioners.

4.3 Performance Claims

We will now compare the performance claims made by the authors of the libraries. Our focus mainly lies on recent papers (i.e. in the last few years), as older publications do not resemble the state of the software anymore.

The authors of KaHIP/ParHIP compare ParHIP in [24] (2017) to ParMETIS and PT-SCOTCH. Yet, they only present the results of ParMETIS compared to ParHIP, because they claim that PT-SCOTCH yielded always worse results than the other two partitioners. In their tests, they run ParHIP in two configurations, namely "fast" (performance-oriented) and "eco" (quality-oriented). They compare both mesh-like graphs (i.e. low connectivity) and graphs with a more complex structure (e.g. web or social graphs). The maximum allowed imbalance is set to 1.03. In the tests where they compare both mesh and social graphs of a size up to about 100 million nodes (the biggest social graph has around 3.3 billion edges), and want to divide into 32 parts, ParHIP-eco wins over ParMETIS in terms of edge cut in many (but not all) cases. The machine used to test has 512 GiB of RAM and the tests run with 32 MPI processes. Especially with the graphs labeled as meshes, ParHIP-fast does not yield much worse edge cuts than ParHIP-eco. In terms of runtime, ParMETIS always outperforms both ParHIP variants when it comes to mesh-like graphs. For social graphs, sometimes ParHIP is faster, and for the biggest social graphs, ParMETIS fails to partition them, due to lack of memory. Apart from this test, they also performed a scaling study using a family mesh-like graphs (called `rgg` and `del` [51]) up to 1 billion nodes, scaling them over the number of MPI processes, diving into 16 parts. Again, the authors compare ParHIP-fast with ParMETIS. Here, ParMETIS again was considerably faster, with the run-time decreasing about linearly with the number of MPI processes, for `rgg25`, `rgg27`, `rgg29`, `rgg31` and `del25`. For `del27` and `rgg31`, there are only two data points, but there the run-time of ParMETIS increases about 10 times when going from 1024 to 2048 MPI processes. In contrast, the runtime of ParHIP-fast began to rise at about 512 MPI processes for almost all mesh sizes. Again, ParMETIS was unable to process the larger mesh-like graphs, failing to process `del29` and `del31`.

In [52] (2013), the authors of ParMETIS and mtMETIS compare against PT-SCOTCH. They use three different test systems (based on Intel Xeon and AMD Opteron processors), and also test a dual graph of a mesh with roughly 1 million vertices. Yet, they only present comparison results for the edge cut per number of cores, when taking the average over several runs of all used meshes. Here, PT-SCOTCH performs worse than ParMETIS and mtMETIS on a little amount of cores first, but yields the best edge cuts comparably when using 32 cores (the maximum here); though there is no big difference here any more. Yet, the edge cut is still worse in comparison to a serial execution of METIS. Concerning memory consumption, here both PT-SCOTCH and ParMETIS scale similarly, with ParMETIS using slightly less memory than PT-SCOTCH. When it comes to the partitioning time, ParMETIS outperforms PT-SCOTCH by being roughly 4 to 5 times faster on all three test machines, considering the dual graph spoken of before.

4.4 Chosen Libraries

In the end, we decided to implement ParMETIS, ParHIP and PT-SCOTCH. Because ParMETIS is currently used in SeisSol, it is obviously included and will be viewed as the base for our comparison. We had to barely change any code to make it work. ParHIP was finally included because it seemed to be a possible new contender for the task, especially managing large graphs really well. The decision for PT-SCOTCH fell, because it had the compatibility interface to ParMETIS.

This mainly leaves only ZOLTAN with PHG and the geometric partitioning methods out. The reason we ultimately decided to ignore ZOLTAN for now is that we wanted to try out ParHIP. If we only tested ParMETIS and PT-SCOTCH, we could have implemented only ZOLTAN, and access the two libraries over the unified interface. But as there are no ZOLTAN bindings to ParHIP, we either would have programmed this ourselves, or add some special case in our interface for ParHIP. Combining that with the fact that the interface for some libraries also lacks some features [4], we decided against this way.

We would have included ZOLTAN independently from a ParMETIS and PT-SCOTCH interface, if I had more time for this Bachelor's Thesis.

4.5 Partitioning Libraries in Detail

We will now present the chosen libraries in more detail.

4.5.1 ParMETIS

The METIS graph partitioning library and its distributed memory version ParMETIS are developed by George Karypis et al. at the University of Minnesota. Version 1.0.0 of ParMETIS was released in 1997 [2]. The basic and original ParMETIS algorithm is described in [23] (comments to these also exist in [34]), further extensions are provided in [53].

Software Packages

METIS, ParMETIS, mtMETIS, and hMETIS are each distributed separately and all have a C++ interface. All variants use the CMake build system, and have a single header which contains all necessary function definitions for using the libraries.

We will now focus on ParMETIS specifically [37]. For using ParMETIS, we have to include both the METIS-header `metis.h` and the ParMETIS header `parmetis.h`. The METIS-header also contains compiler variables which define the size of the used integer types for indices and weights.

All methods are prefixed with `ParMETIS_V3_`. `ParMETIS_V3_PartKway` partitions a graph a multi-level approach, and it said to yield high partitioning quality.
`ParMETIS_V3_PartGeom` uses a geometry-based approach which is said to be quick, but results in worse partitions. Moreover, it does not accept vertex nor edge weights. `ParMETIS_V3_PartGeomKway` first computes a geometry-based partition which is then refined using the multi-level approach. It is said to be quicker than `ParMETIS_V3_PartKway`, still yielding comparable partition quality. Aside from that, ParMETIS contains methods to refine a partitioned mesh with `ParMETIS_V3_RefineKway` and to re-partition a mesh, while not changing the partition mapping too much, with `ParMETIS_V3_AdaptiveRepart`.
`ParMETIS_V3_Mesh2Dual` is a helper method which can be used in our case if we want to generate the dual graph out of mesh vertices and mesh edges.
`ParMETIS_V3_PartMeshKway` exists to partition a given mesh. Effectively, it internally calls first `ParMETIS_V3_Mesh2Dual` and then `ParMETIS_V3_PartKway` on the resulting dual graph. `ParMETIS_V3_PartMeshKway` can handle vertex weights, but does not support edge weights.

Parameter format

The structure of ParMETIS arguments is also used by the interface to KaHIP [54], and (PT)SCOTCH at least provides compatibility methods using them, so we will explain them here in more detail.

In general, each vertex has a global ID (program-wide) and a local ID (rank-wide). The vertices on each rank are numbered beginning with 0. For the global ID, we take the local ID and add the number of vertices which reside on all ranks which are lower than the rank we are currently on. Edges are stored in adjacency lists. Furthermore, the (undirected) edges are stored as two directed edges, i.e. if v and w are adjacent, w is in the adjacency list of v and vice-versa. A list of common arguments in ParMETIS follows. It should be noted that `idx_t` is an integer datatype which can be 32 or 64 bits long, depending on the `metis.h` header.

- `idx_t* vtxdist` should point to an array of size $N+1$, if N is the size of our MPI communicator. `vtxdist[i]` then specifies the minimum vertex ID which is present on rank i. This way, `vtxdist[i+1]-vtxdist[i]` equals the number of vertices on rank i. `vtxdist` should be the same array on all ranks.

- `idx_t* adjncy` stores the adjacency lists for each local vertex (in ascending order w.r.t. the local ID of these). The adjacency list of a vertex consists of the global IDs of its neighbors, with no order specified. The size of this array is therefore the combined length of all local adjacency lists. In our case, it is less or equal than $4n$, with n being the number of vertices.

- `idx_t* vidx` specifies the position of its adjacency list in `adjncy`, in a similar fashion as `vtxdist` does with the number of vertices per rank. The array is of size n (with n as before).

- `idx_t* vwgt` stores the vertex weights. For a single vertex weight, `vwgt[i]` contains the weight for the vertex with local ID i. It is also valid to supply `nullptr`, if only uniform vertex weights are used. For multiple vertex weights, the position $M \cdot i + j$ in `vwgt` contains the weight of vertex i in the j-th vertex weighting system, with M being the number of vertex weights. Hence, this array if of size nM.

- `idx_t* adjwgt` stores the edge weights. `adjwgt[i]` contains the weight of the edge which is defined through `adjncy`. It is therefore of the same size as `adjncy`. `adjwgt` can also be a `nullptr`, if only uniform edge weights are used.

- `idx_t* part` stores the resulting partition assignment. `vwgt[i]` contains the (zero-based) partition for the vertex with local ID i. It has the same size as `vidx`.

Partition weights are supplied as an array of `double` or `float` which sums up to 1. The weights may differ for each different vertex weight. There is also a separate imbalance array which allows to specify an individual imbalance for each planned partition and vertex weight.

4.5.2 ParHIP

KaHIP (short for **Ka**rlsruhe **Hi**gh Quality **P**artitioning) is developed by Christian Schulz and Peter Sanders at the Karlsruhe Institute of Technology (KIT) and the University of Vienna [9]. Since version 2.0, the distributed memory partitioner ParHIP is part of the package.

Partitioners

We will provide a short overview over some papers for some partitioner algorithms provided in KaHIP.

- *KaFFPa*: The **Ka**rlsruhe **F**ast **F**low **Pa**rtitioner. The basic algorithms are found in [55], [49] and [30].
- *KaFFPaE*: **KaFFPa E**volutionary. It uses KaFFPa at its base. The basic algorithms for this are found in [27].
- *KaBaPE*: The **Ka**rlsruhe **Ba**lanced **P**artitioner **E**volutionary. The basic algorithm can be found in [49].
- *ParHIP*: **Par**allel **Hi**gh Quality **P**artitioner. A distributed memory partitioner which makes use of KaFFPaE for initial partitioning. More information on the algorithms can be found in [24].

Software Package

For all of the before-mentioned algorithms, there exist command-line tools within the KaHIP package. ParHIP and KaFFPa have a command line program on their own, KaBaPE shares a command line program with KaFFPaE. Apart from that, the KaHIP package contains an edge partitioning algorithm (also as distributed memory variant), as mentioned before, and some helper programs, e.g. to run the label propagation algorithm [54]. Only KaFFPa has an exposed C++ interface as of version 2.10. As a build system, KaHIP uses scons, and additionally some bash scripts for easier compiling. An installation system is still missing.

4.5.3 PT-SCOTCH

SCOTCH and PT-SCOTCH are developed at Laboratoire Bordelais de Recherche en Informatique (LaBRI) at the University of Bordeaux 1, prominently by François Pellegrini. The first papers date back to 1994, the first release of PT-SCOTCH was in 2007 [3]. Information on the algorithms present in SCOTCH and PT-SCOTCH can be found in [56], [50], and [34].

Software Package

Both SCOTCH and PT-SCOTCH are distributed in one package. It uses makefiles directly without any additional build system. It also supplies several makefiles which defined parameters for different systems. Settings like threading or the integer length can be set using compiler variables. The header for PT-SCOTCH (`ptscotch.h`) needs the header for SCOTCH to properly work (`scotch.h`).

As mentioned, both packages contain METIS and ParMETIS compatibility methods which cover some of the ParMETIS methods with an identical signature. Yet, these methods ignore some of the input values, for example, the seed value does not have an effect, and the edge cut is not computed (cf. source code at [42], referring to `ParMETIS_V3_PartKWay`).

When the compatibility methods are not used, PT-SCOTCH provides an own interface for C which mimics object orientation. There are several structures, like `SCOTCH_dgraph` or `SCOTCH_strat`. Each of these can be initialized with some parameters and filled with data (e.g. `SCOTCH_dgraph` contains pointers to the actual graph data for PT-SCOTCH), and have to be cleaned up after partitioning. Furthermore, parts of the multi-level algorithm in SCOTCH can be replaced with user-implementations [57]. In general, PT-SCOTCH and SCOTCH have two main algorithms implemented: mapping of data onto a target architecture (which equals what we treat here), and re-ordering of sparse matrices [56]. Each of these have their own set of methods (with "map" or "order" in their name). For supporting partition weights, PT-SCOTCH provides many "architectures" on how topologically our partitions are connected, including the possibility to specify the complete graph as topology. The actual weights are inserted as integers.

Aside from that, both PT-SCOTCH and SCOTCH both support the afore-mentioned strategy strings for exact control of the partitioning process [50, 56]. For ease-of-use, default strategies which focus on certain aspects (like quality, speed or balance) are also available.

5 Implementation in SeisSol and PUML

After we have determined three partitioning libraries to work with, we now implement an interface for them into the library PUML. Every class in the PUML library is inside the `puml` namespace. We will omit it in this section to ease readability.

5.1 Current State

As of March 2019, SeisSol uses the ParMETIS library for partitioning.

5.1.1 Reading PUML Meshes In SeisSol

We will now describe the mesh reading process for PUML meshes in SeisSol [14]. The reading process happens in the file `Geometry/PUMLReader.cpp`, class `PUMLReader`. Firstly, PUML reads the mesh (as well as boundary and group data) from the file. Next, SeisSol makes PUML generate faces and edges which are required for calculating the vertex weights, using the method `PUML::generateMesh()`. These are determined after this step in the file `Initializer/time_stepping/LtsWeights.cpp`, class `LtsWeights`. To calculate the maximum wave speed (which in turn is used to determine the LTS cluster), SeisSol depends on a material file, supplied in the YAML file format [58]. The material file is parsed with the easi library [59]. After the vertex weights have been generated, SeisSol issues the actual partitioner call. To determine the partition weights, SeisSol runs a small benchmark (called Mini SeisSol in code) to determine the performance of each MPI rank (this is actually done before the instance of `PUMLReader` is created). Using the resulting partition, PUML then re-distributes the cells (`PUML::partition()`) and then again calls `PUML::generateMesh()` to re-generate faces and edges. Finally, SeisSol generates mesh information and the actual simulation can begin.

5.1.2 Graph Partitioning

The call to ParMETIS itself is issued in the file `PartitionMetis.h` in the PUML library, in the method `partition` of the class `PartitionMetis`. This `PartitionMetis` class itself is completely separate from the rest of the PUML library. In `Geometry/PUMLReader.cpp`, the program first creates a `PartitionMetis` class and then invokes the `partition` method. The resulting data pointer is then passed to the local `PUML` object holding the cell and vertex data. Concerning ParMETIS, the code calls `ParMETIS_V3_PartMeshKway`. This method can effectively be split up/replaced by two other ParMETIS methods, cf. subsection 4.5.1.

5.2 Face Iteration Abstraction

When working with PUML, SeisSol iterates over all cells and each of their side faces [14]. For each cell and given face, the program then computes some data, also considering the data from the cell on the other side of the face (it it exists). When considering the dual graph and ignoring faces which have only one incident cell, this is equivalent to interpreting the graph as directed (i.e. we use 2-sized tuples as edges, not sets of size 2), and then iterating over each directed edge. It may sound trivial on with a single process or a single node, but when working with meshes which are distributed over multiple MPI ranks, communication is needed. In this case, it effectively means that for a single calculation, we have to transfer a temporal ghost layer to other ranks to compute the necessary information.

As an example, one might want to compute edge weights for the dual graph, by taking the sum of the incident vertex weights. For this, we need to compute a sum for each edge in our dual graph, or equivalently, each face in our mesh. The sum depends on the incident graph vertices, or mesh cells. If both of the incident cells lie on the same rank, this poses no problem. But, if the incident cells lie on two different ranks, we will need to communicate the weight of the respective other graph vertex, before we can compute the edge weight. In case we use a format similar to that of ParMETIS, we will need the resulting edge weight value on both ranks. This can be realized by transferring the vertex weights to the respective other rank, and then compute the edge weight locally on each of the two ranks.

We here present a direct abstraction (with a few adjustments) for this concept of iterating over all faces.

To establish the basics, each cell is present on exactly one rank. In contrast, cell faces, cell edges and cell vertices might be present on multiple ranks, in a way that for each cell we have the faces, edges and vertices of the cell also locally on the same rank. In the case of a cell face, this means that it is shared between at most two different ranks. For data transfer with MPI, it is beneficial to focus on the cell faces instead of cells themselves. We may differentiate between three types of cell faces (cf. figure 5.1):

- *Inner faces*: a face which has two incident cells which are on the same rank
- *Shared faces*: a face which has two incident cells, but they are on different ranks
- *Boundary faces*: a face which has only one incident cell

Of these three, only *shared faces* are present on more than one rank. Subsequently, only these need to be handled with MPI.

Speaking in a more abstract form about the face iteration: the goal is to execute a given function $h(f, c_1, c_2)$ with c_1, c_2 being adjacent cells (c_2 may also be a placeholder value NONE if no other incident cell to f than c_1 exists) and f being the face incident to both c_1 and c_2. We want to compute h for all possible such triples f, c_1, c_2. When using MPI communication, we may not have c_2 present on the same rank as c_1 (except if we introduced never-updated ghost layers and only needed the information of the cells which lie at an MPI boundary; which is not always the case, see SeisSol code for more [14]), and we do not know how h processes its input values, including c_2. Therefore, we split up the call $h(f, c_1, c_2)$ into two other functions, $fh(f, c_1, v_2)$ and $ch(f, c_2)$. $ch(f, c_2)$ will be executed remotely for *shared faces* and locally for *inner faces*, and the result will be stored in v_2. As v_2 is a variable, we may transfer it with MPI and then execute $fh(f, c_1, v_2)$ locally. For that, f and c_1 are present on our local rank, as before. In total, $ch(f, c_1, ch(f, c_2))$ should behave like $h(f, c_1, c_2)$. To handle *boundary faces*, we say that v_2 is NONE, if c_2 is NONE.

We may also change the cell-centric iteration scheme, as used in SeisSol, to a face-centric iteration scheme. Then, instead of iterating over all cells and then all side faces, we simply iterate over all faces directly. It is a little easier to describe the algorithm this way, as the MPI-related part is already face-centric. Let f be a face, c_1 and c_2 be the incident cells (or c_2 is NONE). Assume for now that c_2 is not NONE. Then we need to execute $h(f, c_1, c_2)$ and $h(f, c_2, c_1)$. In the cell-centric iteration scheme, we would handle c_1, and under this context f and the incident c_2. Then later, we would do the same with the roles of c_1 and c_2 swapped. Consequently, we also need to do this in the face-centric iteration. Here, we handle f (instead of the cells) and directly execute $h(f, c_1, c_2)$ and $h(f, c_2, c_1)$ after another. With MPI and distributed memory, this boils down to: if f is an *inner face*, we execute $fh(f, c_1, v_2)$ and $fh(f, c_2, v_1)$ with $v_1 = ch(f, c_1)$, $v_2 = ch(f, c_2)$ which we can simply compute locally. For *shared faces*, we execute $fh(f, c_1, v_2)$ on the rank which has c_1 and $fh(f, c_2, v_1)$ on the rank which has c_2, i.e. v_1 and v_2 have to be transferred. $v_i = ch(f, c_i)$ is computed on the rank which has the respective cell c_i (with i being 1 or 2). *Boundary faces* (here, we assume c_2 is NONE) can simply be handled with a call $fh(f, c_1, \text{NONE})$.

The differences and advantages/disadvantages of the change from cell to face-centric are mostly cosmetic or non-effective, if we assume that the iteration order is irrelevant (which it mostly is for our applications). To be more specific, the iteration of all faces requires less calls to the PUML-internal hash maps than in the cell based method. There, we get a face of a cell internally by first computing all face vertices and then looking up the face in a hash table. The advantage of the face-centric iteration is that such hash map calls are not needed, as a face in PUML stores its incident cells. In some cases however, SeisSol needs to know from a face on which side of a cell it lies (numbered from 0 to 3) [14]. To get this information when knowing just cell and face requires some hash map calls, and an iteration over all side faces of the cell; i.e. it is very cheap to find this out. The cell-centric scheme gives us this information for one cell at least, and hence allows a more beautiful formulation in these cases; the face-centric scheme gives us no information about the cell sides. Empirically speaking, there should be no noticeable performance difference between face and cell-centric schemes.

5.2.1 The Algorithm

We may now give an algorithm for the abstraction; from the perspective of a single rank in an MPI application (cf. algorithm 5.1): we are given the functions fh and ch which are the same for all ranks. Additionally, we keep a list on each rank for each target rank

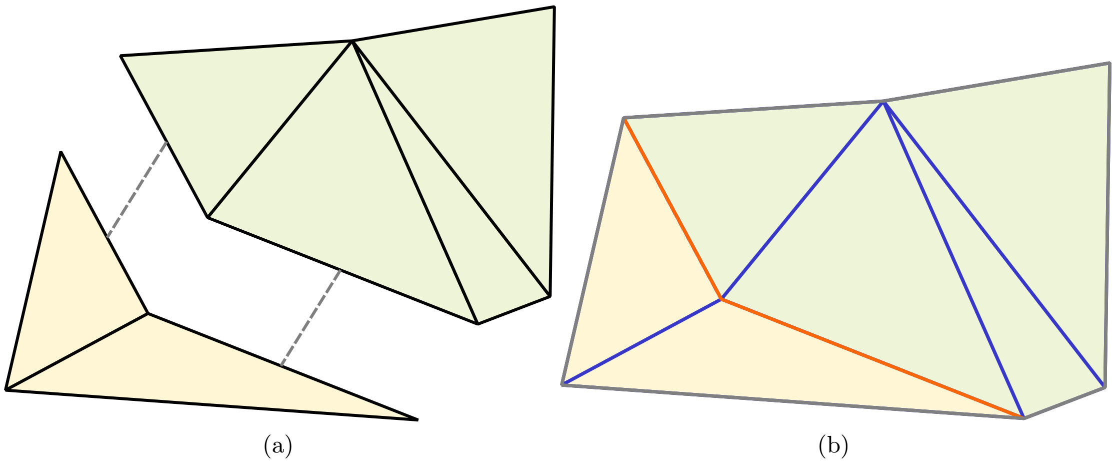

Figure 5.1: An example distribution of a two-dimensional mesh (cf. figure 2.1a) on two MPI ranks. Figure (a) shows the distribution. The color represents the rank the cell resides on. Faces which are represented on both ranks are connected using a gray dotted line. Figure (b) then shows how the faces (which are also edges here) are classified. The gray faces are boundary faces, the blue faces are inner faces, and the red-orange faces are shared faces.

(we call it $T[r]$ with r being a rank). Firstly, we then iterate over all faces and process *boundary* and *inner faces* immediately. If we encounter a *shared face*, we add it to $T[r]$ with r being the other rank the face is present on. We then sort each $T[r]$ by the global face id (which is unique over all faces over all ranks). The idea to do this is already present in SeisSol [14]. Next, we compute an array $D[r]$ which stores the values of ch of the local cells of $T[r]$. This array $D[r]$, we then transfer to the other rank r for each r. Consequently, we receive an array $R[r]$ from each other rank r as well, as every rank transfers data to us. We do not need to transfer any information about how many faces or which faces we transfer. If we have two ranks r_1 and r_2, both have the same set of faces shared with each other: suppose f is a face, c_1 and c_2 are incident to it, and c_1 lies on r_1 and c_2 on r_2. Then r_2 needs the information $ch(f, c_1)$ from r_1 and vice-versa. So, f will be inside $T[r_1]$ on r_2 and $T[r_2]$ on r_1 (remember: each rank has its own T, D and R arrays). As a result, these two arrays are the same. The sorting of $T[r]$ then also voids the need to transfer the global IDs of the *shared faces*. After this is done, we may simply handle all *shared faces* using $D[r]$ which we just received. For those, all that is left is to compute fh for them.

To avoid re-calculating $T[r]$, we may also pre-compute it. In this case, we ignore *shared faces* in the first iteration over all faces. In case that we have only one rank present, the whole algorithm then does exactly the same as if it was run as a local loop, as there are no *shared faces* then.

```
Data: $fh(f, c_1, v_2)$ and $ch(f, c_2)$ with $f$ being a cell face, $c_1, c_2$ being cells and
      $v_2$ being the return type of $ch$
Let $T[r]$ be a list for each rank $r$;
foreach Faces $f$ do
    if $f$ is an inner face then
        Let $c_1$, $c_2$ be the cells which are incident to $f$;
        Execute $fh(f, c_1, ch(f, c_2))$;
        Execute $fh(f, c_2, ch(f, c_1))$;
    else if $f$ is a boundary face then
        Let $c_1$ be the incident cell to $f$;
        Execute $fh(f, c_1, \textsc{none})$;
    else if $f$ is a shared face then
        Let $r$ denote the other rank where $f$ is present;
        $T[r]$.add($f$);
end
foreach Rank $r$ do
    Sort $T[r]$ by global face ID;
end
Let $D[r], R[r]$ be two lists for each rank $r$ which have the same size as $T[r]$;
foreach Rank $r$ do
    foreach Face $f$ at index $i$ in $T[r]$ do
        Let $c_1$ be the cell incident to $f$ which is present on the local rank;
        $D[r][i] = ch(f, c_1)$;
    end
end
Transfer $D[r]$ and receive $R[r]$ for each rank $r$;
foreach Rank $r$ do
    foreach Face $f$ at index $i$ in $T[r]$ do
        Let $c_1$ be the local cell which is incident to $f$;
        Execute $fh(f, c_1, R[r][i])$;
    end
end
```

Algorithm 5.1: The face iteration abstraction. It performs some function fh on all faces in the mesh, taking into account that the incident cells might be on different MPI ranks. In this case, we pre-compute the cell-specific data we need for the call of fh and transfer it to the other rank.

5.2.2 Implementation

We implemented algorithm 5.1 in the file `FaceIterator.h` in the PUML package which contains the class `FaceIterator`. When constructed, this class creates $T[r]$ for each rank r. For this, it simply requires a `PUML` object. To iterate over all faces, we provide the method `iterate` which takes the following arguments in its base form:

- A function called `external_cell_handler` of type `T(int,int)` with `T` being a type from a template parameter. The two arguments specify the local face and cell id, respectively. This function corresponds to *ch* in algorithm 5.1.
- A function called `face_handler` of type `void(int,int,const T&)`, `T` and the two integer arguments as above. The last parameter is the result some call to `external_cell_handler`. This function corresponds to fh in algorithm 5.1, in the case of *shared* or *inner faces*.
- A function called `boundary_face_handler` of type `void(int,int)`, similar to `face_handler`, but missing the third argument. This equals fh of algorithm 5.1 in the case of *boundary faces*. With C++17, it will be possible to unify this argument with the one before by using `std::optional`. If boundary faces are irrelevant (i.e. when looking at the dual graph of the mesh – then only shared and inner faces will form dual graph edges)
- An `MPI_Datatype` which should correspond to `T`.

The execution then closely follows algorithm 5.1, with fh equal to `face_handler` and *ch* being `external_cell_handler`. This means that we transfer array of type `T` over MPI which is why the datatypes of the MPI transfer type and `T` have to match.

To ensure this, we introduced a template-based conversion from C++ type to MPI datatype (class `MPITypeInfer<T>` in `FaceIterator.h`). For primitive types like `int` or `long`, we will directly return the corresponding MPI type (in this case `MPI_INT`, `MPI_LONG`). To also support more complex types, we implemented support for `std::pair` or `std::array`, caching an MPI datatype which is created on first use. For both `std::pair`, we issue a call to `MPI_Type_create_struct` and the `offsetof` macro [60]. For `std::array`, we use `MPI_Type_contiguous`.

To transport the data with MPI, cf. transfer of $D[r], R[r]$ in algorithm 5.1, we flatten the respective arrays in our code, so that we have a single array for data to transfer and a single array for the data to receive. Concerning the MPI communication itself, there are two possibilities: either we use a single `MPI_Alltoallv` call to communicate all data or, if we only have few neighbors, we can use one `MPI_ISend` and `MPI_IRecv` per corresponding rank. In SeisSol, the latter pattern is used more often. For a more detailed discussion of the advantages and disadvantages of either strategy, we refer to [18]. The user can decide which pattern to use; for this purpose we have added a flag to the `FaceIterator` constructor.

On top the above-mentioned arguments for `iterate`, we provide further variants. For example, we provide a method which applies a cell handler on every cell (not only the external ones) and then calls the face handler with the results. Building on top of that, we can replace the cell handler by a `std::vector` or pointer which we supply to the method instead (and is internally replaced by an access to the data structure). Then, the cell i has its data at position i in the `std::vector` or pointer.

The whole `FaceIterator` itself is completely separate from the PUML core (`PUML.h`). In fact, it could be distributed as a single header file with PUML as its only dependency. It is important that `PUML::generateMesh()` is called before creating a `FaceIterator`, the class itself does not check that. In all occurrences of the face iteration is SeisSol right now, this is already the case before the iteration starts.

5.2.3 Considerations

For the future, the first and the last loop in algorithm 5.1 may be parallelized. Though this only can be considered, if fh has no loop-carried dependencies and the order in which the faces are traversed is irrelevant. To simplify the iteration routines, a removal of the `boundary_face_handler` parameter might be possible, as the functionality of it may be also accomplished when simply iterating over all edges locally.

Moreover, currently the internal data structures used for storing information about $T[r]$ use `int`, because `MPI_Alltoallv` does so with its size and displacement parameters. This *could* lead to an integer overflow when considering much larger meshes (than e.g. 221 million cells), so there is no concern right now.

5.3 Creating an Interface for Partitioning Libraries

We mainly focused on enhancing and modifying the PUML library.

In general, all partitioning class in the following depend on the topology of the cells, i.e. if we have tetrahedra or hexahedra. This means that each of the following classes has a template parameter `Topo` which can be either `TETRAHEDRON` or `HEXAHEDRON`. From now on, we will omit this template parameter for better readability.

To support multiple partitioning libraries, we first of all introduced a new base class called `PartitionBase` and made `PartitionMetis` inherit from it. Then, we renamed `PartitionMetis` to `PartitionParmetis`.

Additionally, we introduced a new header `Partition.h` which automatically allows the selection of a partitioning library. It consists of a class `Partition` which contains a static method `get_partitioner(std::string name)`, accepting the name (as string) of a partitioning library or configuration. It returns an `std::unique_ptr` to a subclass of `PartitionBase`. In the end, the user should just include `Partition.h` and run the `get_partitioner` method to get a partitioner – or a `nullptr`, if the name was incorrect.

Table 6.3 presents a list of possible partitioner names. As of now, each new partitioner has to be added by hand into `Partition::get_partitioner`.

5.3.1 Dual Graph Creation

Next, we examine the conversion from mesh to dual graph a little closer, as well as the partitioning method used. For that, we split `ParMETIS_V3_PartMeshKway` up into `ParMETIS_V3_Mesh2Dual` and `ParMETIS_V3_PartKway` in `PartitionMetis`. This way, we could now also support edge weights, if ever needed. Next, we pulled the logic of `ParMETIS_V3_Mesh2Dual` completely out of `PartitionMetis`. For the graph representation, we instead created a new class `PartitionGraph` (file `PartitionGraph.h`). The reason for this was that other partitioners did not have a comparable graph generation method, and to keep the partitioner-related files free from other partitioner dependencies. To completely eliminate the dependency on ParMETIS for all partitioners, we decided to re-implement the functionality of `ParMETIS_V3_Dual` for our specific structure. For this, we use the face iteration abstraction as described in section 5.2. Hence, `PUML::generateMesh()` has to be run before the graph creation, but this is guaranteed if local time stepping is used, as the face iteration is already used there.

The `PartitionGraph` stores the data in an adjacency list format similar to that of METIS/ParMETIS (cf. parameters `vtxdist`, `vidx`, and `adjncy` from section 4.5.1; called `vertex_distribution`, `adj`, and `adj_disp` in `PartitionGraph`), all as `std::vector`. Additionally, `PartitionGraph` holds a vector with data like `vtxdist`, but for edges instead of vertices (called `edge_distribution`). Besides that, the class provides useful information such as the local and global vertex as well as edge count. As of now, `PartitionGraph` also stores vertex and edge weights. This might be sub-optimal, as it is simply a copy of data which already has been put together into an array; but this way, all data is in one data structure. To define potential edge weights, `PartitionGraph` builds on the face iteration abstraction. By iterating over the faces again in the order (here, the order matters) in which we created the adjacency lists, it is possible to define for each entry in the adjacency list a weight with the data from the two adjacent vertices. For this, `PartitionGraph` provides several methods called `forall_local_edges` which accomplish exactly that. Internally, it is a simple wrapper around a `FaceIterator`. To support geometry-based algorithms, `PartitionGraph` provides a method called `geometric_coordinates` which fills a given vector with the barycenters of the local cells.

5.3.2 Adoption of Partitioning Libraries

Each of the graph partitioners was adopted in a single header which contains a class that inherits `PartitionBase`. As a downside, every time we include such a header, we also include the headers of the underlying partitioning library. But as PUML is constructed as a header-only library, there is no other choice to that (except violating that principle).

To subclass `PartitionBase`, the method `partition` has to be overridden; it is declared `virtual` in `PartitionBase`. During the course of development, the signature to the method changed to `void partition(int* partition, const PartitionGraph& graph, double* nodeWeights, int nparts, double imbalance, int seed)`. The first parameter is the output partition, as before. The `graph` parameter supplies the graph to partition – including vertex and edge weights. `nodeWeights` specifies the partition weights, as present in ParMETIS. The last three parameters are the number of parts to partition in, the maximum allowed imbalance in ParMETIS format, and the seed to use. Additionally, we added an overload for this method which replaces the pointers with `partition` and `nodeWeights` with `std::vector`. Another overload returns a vector with the partition result data, instead of requiring an existing one as input.

Every header for a partitioning library is included in `Partition.h` (and registered in `Partition::get_partitioner`), if a compile-time variable is defined. For example, the header `PartitionParmetis.h` is included, if the preprocessor variable `USE_PARMETIS` exists.

For the libraries that we implemented, it was common for us to firstly convert the data from the supplied `PartitionGraph` object into the partitioning library-specific data types. After that, the partitioning library could be called.

We also included a dummy header which does nothing, called `PartitionDummy`.

ParMETIS

After our refinement, we implemented the partitioning with `ParMETIS_V3_PartKway` and `ParMETIS_V3_PartGeomKway`. We decided against testing `ParMETIS_V3_PartGeom` which simply applies a geometric algorithm, because it could not take any parameters concerning vertex or edge weights. To switch between the different implementations, we simply introduce a flag which is set in the constructor – if the user wants it.

We left an implementation which stays as close as possible to the original code before the transformation, in the header `PartitionParmetisLegacy.h`. As a result, this implementation does not support edge weights.

PT-SCOTCH

For implementing PT-SCOTCH, we could have used the ParMETIS compatibility methods. Yet, the compatibility methods completely ignore seed and imbalance values. They do not give access to the strategy parameters either which can be used to configure PT-SCOTCH more in-depth, like enforcing as much balance as possible or preferring speed over quality (or vice-versa). We therefore used the SCOTCH library methods directly, as these give us more control over the graph partitioning process. In general, the method calls are still similar as in the source code of the ParMETIS compatibility methods. That is, we first create a `SCOTCH_dgraph` structure and fill it with data. Next, we initialize our strategy (`SCOTCH_strat`) with one of the default strategies which we let the user choose (cf. `SCOTCH_stratDgraphMapBuild`). Then, we add the partition weights (`SCOTCH_arch`, architecture definition; using method `SCOTCH_archCmpltw`), and then we can partition with a call to `SCOTCH_dgraphMap`. After this is done, we then simply clean up.

ParHIP

To implement ParHIP, we had to write the interface to the library ourselves: as of March 2019, there was no interface available in the public repository [41]. We oriented our code at the existing code for the command line program called `parhip`, cf. the file `parhip.cpp`. Furthermore, we tried to mimic the code used for the interface to KaFFPa as closely as possible. The method signature orients at the corresponding KaHIP methods:
`void parhip(int* vtxdist, int* xadj, int* adjncy, int* vwgt, int* adjwgt, int npart, int seed, int mode, double imbalance, int* edgecut, int* part, MPI_Comm comm)`. The first five parameters are the same as in calls to ParMETIS. `npart`, and `seed` set the number of parts to partition in and the random initial seed, respectively. `mode` specifies a partition configuration which is a combination of either ultrafast, fast, and eco and one of mesh and social. The modes are the same as in the `parhip` application, i.e. eco provides the best quality, but is slower, and ultrafast is the fastest configuration, but yields the worst results. `imbalance` (maximum allowed imbalance) is stored as in ParMETIS, i.e. as $1 + \epsilon$ (e.g. if we want a maximum allowed imbalance of $\epsilon = 1\%$, we supply `1.01` here). `edgecut`, `part`, and `comm` are then similar to the ParMETIS call again. The calling process roughly goes as follows:

- We generate the internal configuration (class `PPartitionConfig`) according to the presets (ultrafast, fast, eco, strong; though the latter is not implemented as of today) and also set the random number generators. Note that `srand` is set for this purpose here.
- Next, we generate the actual graph out of the given input data. It is saved in an instance of the class `ParallelGraphAccess`. The generation goes analogously to the graph reading process in the file `parallel_graph_io.cpp` for METIS-like input files.
- Then, we start the actual partitioning, using the `distributed_partitioner` class.
- After that, we finish by writing out the partition and the edgecut.

We considered it possible to extend the library to support node weighting (as done in ParMETIS), yet we did not pursue this idea due to the given time constraints, and the possible long testing and verification phase.

5.3.3 Build System

As SeisSol uses `scons` with a small helper framework, we adopted both ParHIP and PT-SCOTCH to these. ParMETIS was already present, as it is already used within SeisSol. The way it is implemented right now, the system currently looks for the headers and the static libraries in the default paths, i.e. the `C_PATH` and `LIBRARY_PATH` environment variables.

6 Evaluation

We now present the results we obtained from the tests of our interface. For this, we make the partitioning libraries partition several dual graphs of meshes which could be used by SeisSol for an earthquake simulation. We will examine the resulting partitions, change parameters, and observe the effects. In the end, we will shortly compare the statements made in the papers of section 4.3 to our findings.

6.1 Test Setup

Firstly, we describe our testing procedure and environment.

6.1.1 Test Program

Our test program first reads a PUML file and generates the mesh data for it. If we want to, we can also supply a material file which can be used to derive vertex and edge weights. After that, we call the partitioning library we want to use and wait for the process to finish and the result data to be written. The result data (stored in a JSON file) contains raw information about the size and cut of each partition to each other partition as well as some deduced properties like the total edge cut etc. We also made it possible to switch each partitioner with command-line parameters, without re-building the program.

The test program receives its input configuration (e.g. partitioner to use, mesh to use etc.) as JSON files. It is also possible to supply multiple JSON files as arguments. We traverse the input configurations from left to right, and always maintain the current configuration (as JSON object). When we read a new JSON file (containing a JSON object), we combine it with the existing configuration object using the JSON merge patch format [61]. This effectively means that the new file can override or add properties to our current configuration JSON object. To shorten test time (especially for seed tests and large meshes), we also allow to process an array of configurations in one program run. The obvious disadvantage is that potential memory leaks or other unseen effects might harm the later partitioning runs. If we input a JSON array file as parameters, we build to Cartesian product with all existing configurations, built from the already read parameters. JSON files which only consist of a JSON object are interpreted as array of size 1. We start with a single empty JSON file, before we read the first parameter.

To support the vertex weights described in section 2.4, we had to adopt some of the code from SeisSol. We copied `Initializer/time_stepping/LtsWeights.cpp` and `Geometry/PUMLReader.cpp`, as well as their header files to our test software. Because of

Node processor type	Intel Xeon Phi 7210-F
Processors per node	1
Cores per node	64
DDR4 memory per node	96 GB
HBM memory per node	16 GB
Interconnect type	Intel OmniPath
Interconnect bandwidth	25 GB/s (2 links)

Table 6.1: Specifications of the CoolMUC3 cluster; for more, see [62]

this, we needed to import easi [59] which we did as GIT submodule. easi then required some other GIT submodules we needed to add to our project as well. Now we could load the SeisSol material files which are stored in the YAML file format [58].

For measuring time, we issue a call to `MPI_Wtime` right before and right after we call `PartitionBase::partition` (of a subclass of `PartitionBase`). The peak memory consumption is measured by calling `getrusage` before and after the partitioning (but before/after the call to `MPI_Wtime`). Then, we use the `ru_maxrss` attribute of the filled `struct rusage`. This of course conflicts with the array processing described above which is why we do not use it when measuring memory.

The output file of a run contains information about partitioning run-time, edge cut, and resulting imbalance. It also stores the configuration it was called with. Additionally, we store for each pair of parts the accumulated weight of edges which connect them and for each rank how much peak memory before and after the partitioning routine call happened.

In the end, we adapted the scons file from the existing test software which was used to test PUML itself, and added the new partitioning libraries as well as easi to the script.

6.1.2 Test Environment

For running our tests, we used the Linux Cluster of the Leibniz Computing Center (LRZ).

Hardware and Software Environment

We ran the datasets on the CoolMUC3 system of the Linux Cluster. Table 6.1 shows the exact specifications of the used test machines [62]. The CoolMUC3 cluster consists of nodes with a 64-core Intel Xeon Phi CPU of the Knights Landing series. Each node is equipped with 96 GB of DDR4 RAM and 16 GB HBM memory. We will only make use of the DDR4 RAM for our tests. Intel OmniPath serves as interconnect for the cluster.

Software-wise, slurm served as workload scheduler. As Linux distribution, SUSE Linux Enterprise Server 12 SP3 was used.

Building

We use the Intel compiler (icc) version 17.0.6 20171215. As environment variables, we set the following (i.e. we do not include specific optimizations like AVX512 for the Knights Landing processors):

```
export FC=mpiifort
export CC=mpiicc
export CXX=mpiicpc
export MPIFC=mpiifort
export MPICC=mpiicc
export MPICXX=mpiicpc

export FFLAGS="-O3"
export CFLAGS="-O3"
export CXXFLAGS="-O3"
```

All partition libraries are installed into a local directory. Version-wise, we used the most recent versions for all partitioners, i.e. ParMETIS 4.0.3 (with METIS 5.1.0), KaHIP 2.10, and PT-SCOTCH 6.0.6.

ParMETIS We installed ParMETIS aside from METIS. The build process went without any special occurrences. To configure ParMETIS (and METIS) for 64 bit indices and weights, we had to modify `metis.h` and change the values of the preprocessor variables `IDXTYPEWIDTH` and `REALTYPEWIDTH` to `64` (only the former is necessary to support larger graphs and vertex weights).

ParHIP We added our own ParHIP interface to the scons files already present in the KaHIP package. To compile, we then ran `parallel/build.sh`. Still, the header and the compiled library had to be copied manually into the installation directories (under `bin/` and `lib/`, respectively). The latter was located under `parallel/parallel_src/optimized_nooutput` after compiling.

PT-SCOTCH From the provided makefile supplements in `/src/Make.inc`, we used `Makefile.inc.x86-64_pc_linux2.icc.impi`, configured for ICC and Intel MPI. We additionally specified `-DINTSIZE64` and `-DIDXSIZE64` to configure for 64-bit integers. We compiled PT-SCOTCH without multi-threading for the algorithms (the number of threads is currently only defined at compile time; we did not specify it), so performance might increase that way.

Test Software For compiling our test software, we simply ran scons, with our local installation directory in `C_PATH`, `LIBRARY_PATH`, and `LD_LIBRARY_PATH`. The build process worked flawlessly, after we installed ParMETIS, ParHIP and PT-SCOTCH. (by now, it is required for the test program that all three of them are installed)

Identifier	Name	Vertices[a]	Edges[b]
`tpv28`	SCEC/USGS Spontaneous Rupture	366 276	1 456 718
`tpv33`	Code Verification Project	1 118 216	4 454 918
`loh1-small`	LOH.1 benchmark	386 518	1 524 392
`loh1-large`		7 252 482	28 752 118
`sumatra-221m`	2004 Sumatra Earthquake	220 993 734	881 935 700
`landers-7m`	1992 Landers Earthquake	7 509 016	29 937 384
`landers-98m`		98 138 869	389 672 068
`landers-191m`		191 098 540	761 146 364

[a] Dual graph vertices
[b] Dual graph edges

Table 6.2: Earthquake meshes used for testing

6.1.3 Test Datasets

Table 6.2 shows a list of the used test datasets. The first few meshes (`tpv28`, `tpv33`, `loh1-small`, `loh1-large`) come from benchmark and verification sets for earthquake simulation software. The second kind of meshes (1992 Landers Earthquake with 7, 98, and 191 million cells; and the 2004 Sumatra Earthquake with 221 million cells) model real earthquakes in different resolutions.

6.1.4 Partitioner Configurations

Table 6.3 shows a list of partitioner configurations we used.

From ParMETIS, we try out the standard `ParMETIS_V3_PartKWay()` method, as well as `ParMETIS_V3_PartGeomKWay()`. For comparison, we also include a method which uses the old partitioning code (modified as little as possible). This legacy method is not able to take edge weights into account, as it uses `ParMETIS_V3_PartMeshKWay()` internally.

As PT-SCOTCH comes with a variety of configuration options, we try out some of the default strategies. For all the possible optimization directions speed, balance and quality, we chose to test all of them and also to combine speed with balance and quality with balance. (combining speed with quality would not make sense here) By looking into the source code [42], we could also see that the quality focus has no effect (it simply does not change the strategy), hence we do not have to test it.

For ParHIP, we chose to try out the three different given settings for meshes: eco, fast and ultrafast.

Identifier	Partitioner	Remarks
`legacy`	ParMETIS	Old SeisSol partitioner code, for comparison
`parmetis`	ParMETIS	Uses `ParMETIS_V3_PartKWay()`
`parmetis-geo`	ParMETIS	Uses `ParMETIS_V3_PartGeomKWay()`
`ptscotch`	PT-SCOTCH	Default strategy
`ptscotch-b`	PT-SCOTCH	Strategy with focus on balance
`ptscotch-s`	PT-SCOTCH	Strategy with focus on speed
`ptscotch-sb`	PT-SCOTCH	Strategy with focus on speed and balance
`parhip-eco`	ParHIP	ParHIP eco mesh mode
`parhip-fast`	ParHIP	ParHIP-fast mesh mode
`parhip-ultrafast`	ParHIP	ParHIP ultrafast mesh mode

Table 6.3: Partitioner configurations

6.1.5 Test Settings

We will now describe the default test settings which we use in the following sections. All tests are done on the CoolMUC3 machine. For default settings, we use:

- The mesh `loh1-large`
- 256 partition parts
- Seed value: 1
- Maximum allowed imbalance: 1%
- LTS rate $r = 2$, as always in SeisSol (this is only relevant for the LTS weights)
- MPI processes: 64, distributed on 16 nodes (i.e. 4 MPI processes per node)
- Vertex weights set as described in section 2.4; likewise edge weights are set uniformly to 1

These default settings are set to be a bit harder than in a realistic case, because the cells per partition ratio is rather low with about 28300 cells per partition part in the perfectly balanced case. The reason we did not use 64 MPI processes per node is that it caused out of memory problems very quickly. This way, we can also simulate the performance when the MPI processes of the partitioners communicate mainly over the interconnect and not inner-node, making the scenario more realistic. We also do not use one MPI process per node, as this would be impractical on the cluster we used (due to high waiting times for resources). In comparison, in [10], they use one MPI process per node and the cells per partition ratio is about 72000 for 3072 nodes and 221 million cells, or about 100000 for 512 nodes and 51 million cells. We only present a single run for each test, because of time constraints.

We evaluate the partitioners by three criteria, namely edge cut, resulting imbalance and partitioning time. Edge cut and resulting imbalance can be directly calculated from the partitioning output. We do not use any library to perform these calculations. The time measurement does not include the mesh to dual graph conversion (strictly speaking, excluding `legacy`), cf. subsection 6.1.1. Additionally, we will look at the additional amount of memory the libraries use. In the following, we are going to use the terms “mesh” and “dual graph of the mesh” interchangeably, as it is clear that we always mean the latter when partitioning. Likewise, vertices mean dual graph vertices in the following sections, and they are equivalent to cells here. The same holds for edges and faces.

6.2 General Evaluation

First of all, we examine how the partitioners work, when using the settings from 6.1.5 (with slight modifications in some cases).

6.2.1 Vertex and Edge Weights

First, we test the effect of vertex and edge weights on the dual graph. In this special case, we are going to use the mesh `tpv33` instead of `loh1-large`. For random vertex and random edge weights, we chose integer values between 1 and 1000, inclusive. We also tried to test larger meshes (like `loh1-large`) with random values. This required us to compile the partitioners with 64 bit support, otherwise they encountered integer overflows. The same holds, if we chose larger (than 1000) random values for the mesh `tpv33`. Figure 6.1 shows an overview over the results for different vertex weighting configurations, run on the mesh `tpv33`, with 256 parts maximum. We show three different vertex weightings: uniform (all set to 1), random (for stress testing), and real (i.e. the weights from section 2.4). As one can see, the edge cut does not differ very much for all different vertex weighting systems. `parhip-ultrafast` is seen to yield the worst edge cut in all cases, while both PT-SCOTCH and ParMETIS (and to some extent also `parhip-eco`) almost tie for the best results, with PT-SCOTCH slightly outperforming ParMETIS. All variants of PT-SCOTCH show approximately equally good results. Concerning the imbalance, all but ParMETIS obey it strictly, while ParMETIS slightly violates it. Furthermore, `ptscotch-b` almost completely balances the graph. In contrast to ParMETIS, `ptscotch-b` seems to have most trouble with uniform vertex weights, while ParMETIS struggles with random most. When it comes to the time spent for partitioning, ParMETIS is the quickest, with PT-SCOTCH not far behind. `ptscotch-s` and `ptscotch-sb` in this case as double as fast as their respective counterparts. In the case of larger meshes like `loh1-large`, this relation is not that strong any more. Only ParHIP needs much more time, with `parhip-ultrafast` and `parhip-fast` still staying under 100 seconds, `parhip-eco` needs a bit more than 7 minutes for random vertex weights. Moreover, `parhip-eco` needs for this particular weighting about twice as long as for both uniform vertex weighting and the variant that is currently used in

SeisSol.

In the tests with random vertex and random edge weights on `tpv33` (we have no graph available for that), PT-SCOTCH became extremely slow. We aborted the run with `ptscotch` after approximately 15 hours, with no result. Though again, it should be noted that we did use the single-threaded variant of PT-SCOTCH. With additional multi-threading, this time might be reduced.

6.2.2 Memory consumption

Next, we look at the memory consumption of the partitioners of one run with default settings per rank. This time, we use the default settings from subsection 6.1.5 (i.e. the mesh `loh1-large`). Table 6.4 shows the results. Note that the table covers the memory used for the whole interface, including a potential conversion of data to the right bitlength. If we estimate by that we have around 113320 cells per rank (which equals a possibly balanced distribution of all 7 million vertices of the `loh1-large` mesh) and that each cell is connected with each other cell, then we will use approximately 5.4 MiB of additional memory ($113320 \cdot 8$ to get the length of an array storing a `uint64_t` for each cell; then we need this times 4 for the adjacency list plus 1 for the adjacency index list, plus another 1 for the vertex weights) which is negligible, compared to the other consumed memory. As one can observe, ParMETIS requires the least memory, especially when using the geometric initial partitioning. PT-SCOTCH (independently from its strategy) requires a bit more memory, and shows that there is barely any difference between the different varieties. ParHIP on the other hand requires a lot more memory per rank, yet it has the lowest variance of memory consumption over all ranks. The high memory requirements might be explained with the additional data structures that KaHIP uses to store information about vertices and edges, or they might be due to an error in our interface.

In total, all nodes together have a memory of 1536 GB, and 24 GB per node (remember: one node has 96 96 GB RAM; we have 16 nodes and 4 processes per node).

6.2.3 Connectivity

As we also tracked which partition part is adjacent to which other parts, we can present in a plot on how many adjacent parts a partition part has. For each partitioner, we did one run with default settings and visualized the results in figure 6.2. For all runs, we used the default settings from subsection 6.1.5. Surprisingly, it does barely matter which partitioner nor which configuration we choose, we end up with at maximum about 46 neighbors which is about 18% of all possible neighbors. The median seems to center around 8 or 10 which is approximately 3.1% or 3.9%, respectively.

Tests with larger meshes (for whose we do not present graphs here) have shown that generally, the number of adjacent parts decreases, but the median roughly stays the same.

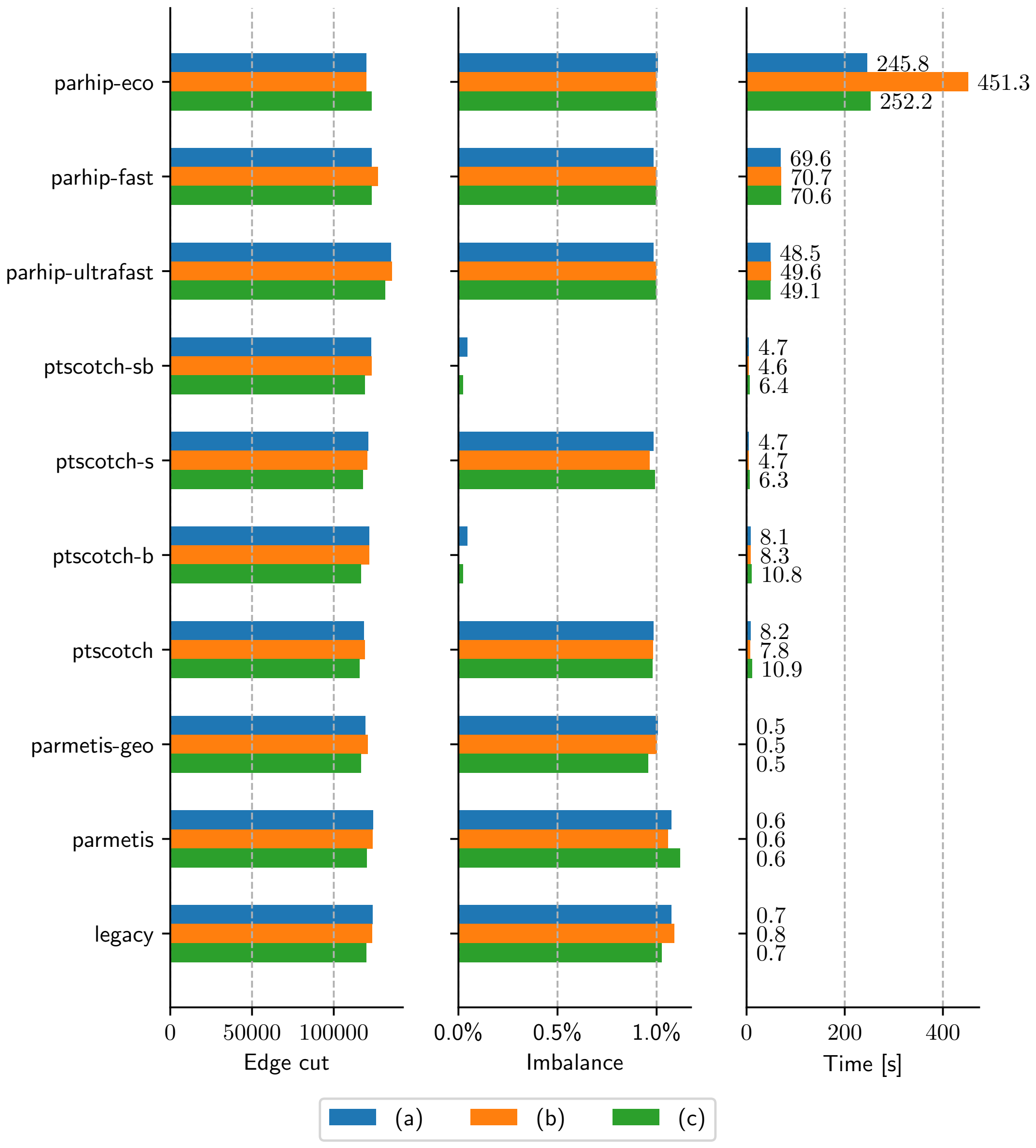

Figure 6.1: An overview over edge cut, imbalance and partitioning time for different vertex weights. The tests have been run on the mesh `tpv33`, partitioning into 256 parts; we did exactly one run for each partitioner and vertex weighting model. Dataset (a) means uniform weights of 1 for all vertices. For dataset (b), all vertices have random values, equally distributed between 1 and 1000, both inclusive. Dataset (c) uses the vertex weights from section 2.4, as it is currently done in SeisSol. The edge weights in all cases are uniformly set to 1.

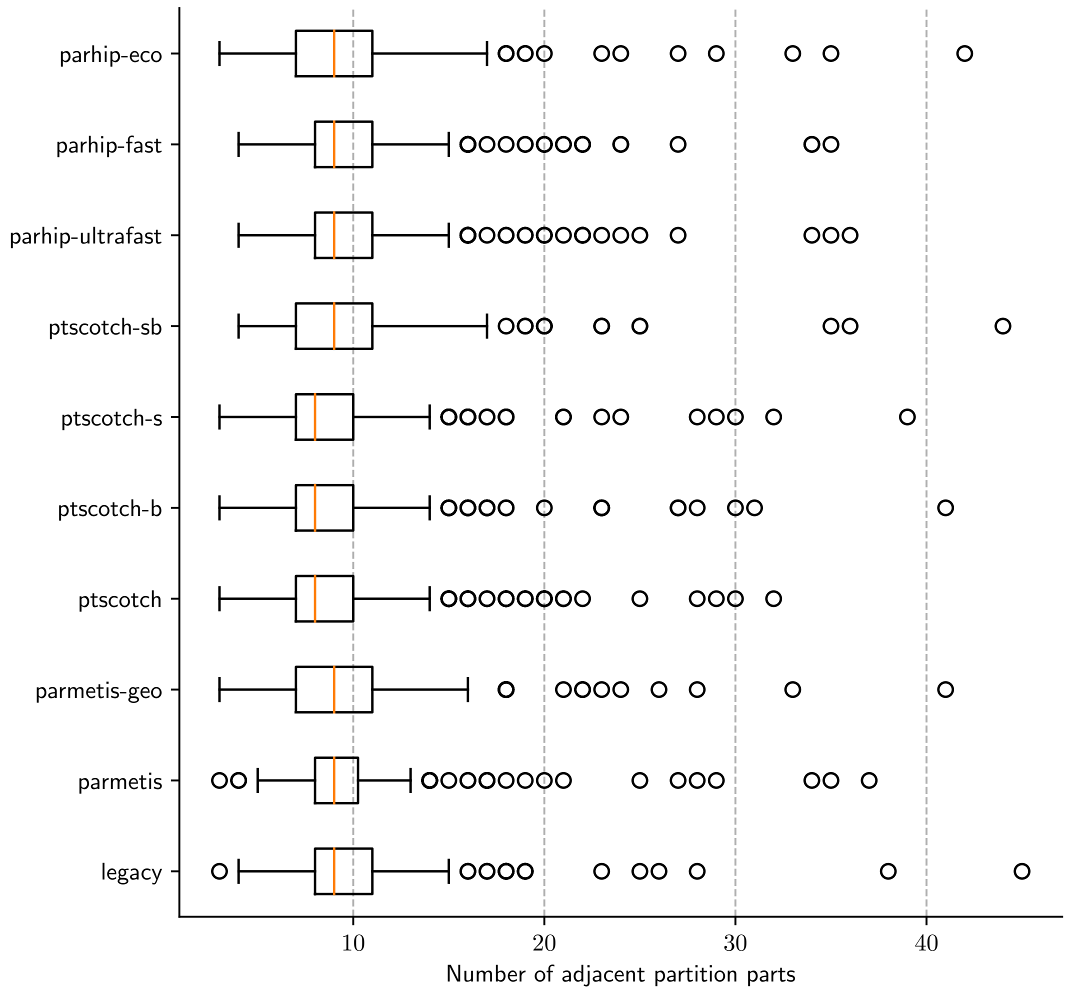

Figure 6.2: The number neighbors for each partition part, for one partitioning run on the `loh1-large` mesh into 256 parts.

	Peak [kB]	Average [kB]	Std. dev. [kB]
`legacy`	67 308	58 595	6064
`parmetis`	62 216	53 031	6033
`parmetis-geo`	56 888	42 936	10 406
`ptscotch`	115 656	99 023	11 980
`ptscotch-b`	114 960	99 515	11 545
`ptscotch-s`	112 032	97 606	11 495
`ptscotch-sb`	113 052	97 926	11 712
`parhip-ultrafast`	3 549 292	3 538 462	5939
`parhip-fast`	3 598 904	3 586 470	5500
`parhip-eco`	3 583 580	3 571 252	5494

Table 6.4: The memory consumption of the partitioners per rank, using one run on the mesh `loh1-large`, and partitioning into 256 parts. The first column shows the peak rank consumption, the second one the average over all ranks and the third one the standard deviation of these values. Note that all memory for our interface to access the library (i.e. type conversion of arrays etc.) is included in the difference.

6.3 Input and Parameter Variation

Next, we vary the parameters specified in subsection 6.1.5 (excluding vertex weighting, as we have already done that in subsection 6.2.1) as well as the input meshes. From now on, we will not test `ptscotch-s` and `ptscotch-sb` any more. They have shown to be slightly faster than `ptscotch`, and `ptscotch-b`, but in our case (with uniform edge weights), it makes barely a difference. Apart from that, `legacy` is not going to be reviewed any further. It performs almost identically to `parmetis`, and it is a bit slower, as it starts with the original mesh instead of the dual graph.

6.3.1 Maximum Allowed Imbalance

First of all, we vary the maximum allowed imbalance. For this, we will test the values 1%, 3%, and 5%. We also tried to set the maximum allowed imbalance to 0.1% or below. In this case, as it can be seen in table 6.5, the partitioners begin to behave strangely. In the end, only PT-SCOTCH stays below the 0.1% value, while ParMETIS and ParHIP exceed it. For ParHIP, this might be attributed to the implementation of our interface, as there, imbalance values of $< 1\%$ are rounded down to 0%. It is also done like this in the implementation of the ParHIP command line program, and the interface to KaFFPa only accepts integer values for the imbalance. In fact, internally, the imbalance is stored in two ways – one of the is to store the percentage value as an unsigned integer [41].

A plot which shows imbalance values greater or equal to 1% is presented with figure 6.3. As one can see, the only partitioner which is really affected by an increasing allowed imbalance is PT-SCOTCH when told to balance the graph at much as possible. We

	Imbalance [%]	Exceeds 0.1%
`legacy`	0.567	Yes
`parmetis`	1.070	Yes
`parmetis-geo`	0.418	Yes
`ptscotch`	0.098	No
`ptscotch-b`	0.002	No
`parhip-ultrafast`	3.557	Yes
`parhip-fast`	3.433	Yes
`parhip-eco`	1.980	Yes

Table 6.5: The effects of using maximum allowed imbalance values which are lower than 1%, using the standard configuration, i.e. one run on the mesh `loh1-large`, 256 parts. For this test, we used a maximum allowed imbalance of 0.1%.

guess that this is because the balancing might be applied afterwards, meaning that with a higher allowed imbalance, the library has to correct the partition more than with lower imbalances. All partitioners obeyed the maximum imbalance for all values $\geq 1\%$, and time-wise the runtimes slightly increased with a higher allowed imbalance for ParHIP and slightly decreased for the other partitioners. But both do not show interesting enough information for it being worth a plot.

6.3.2 Partition Parts

Next, we vary the partition parts. For this, we test power of 16, i.e. $16^1 = 16$, $16^2 = 256$, $16^3 = 4096$, and $16^4 = 65536$. The latter ones may not be very realistic (except if parallelizing solely for MPI, compare table). Table 6.6 shows the ratio of cells per partition part. Yet, this experiment gives a picture on how well the partitioners react to these settings. The effects on the partitioning duration and resulting imbalance can be seen in 6.4. As it can be seen, in the extreme case of 65536 parts, PT-SCOTCH actually becomes the fastest partitioner. ParMETIS on the other hand, seems to grow super-exponentially in time with the number of partition parts. ParHIP consumes more and more time with more partition parts. `parhip-ultrafast` and `parhip-fast` get closer together (in the logarithmic scale) with 65536 parts, and end up with needing over 1000 seconds to complete. `parhip-eco` leaps over this ledge already with 256 parts, and surpasses 10000 seconds for the 65536-mark. Concerning the imbalance, it can be seen in 6.4 that ParHIP in eco mode proves to adhere to the imbalance constraint best, as the only partitioner to do so. All other partitioners violate the constraint. The earliest to do so is ParMETIS, next are ParHIP in fast and ultrafast mode. PT-SCOTCH stays under the maximum imbalance until we try to partition into 65536 parts, but then also fails slightly. As it can be seen, the PT-SCOTCH variant that has to balance the partitions as far as possible gets less efficient as the desired number of partitions grows, until 65536, where it is completely futile and has no effect any more.

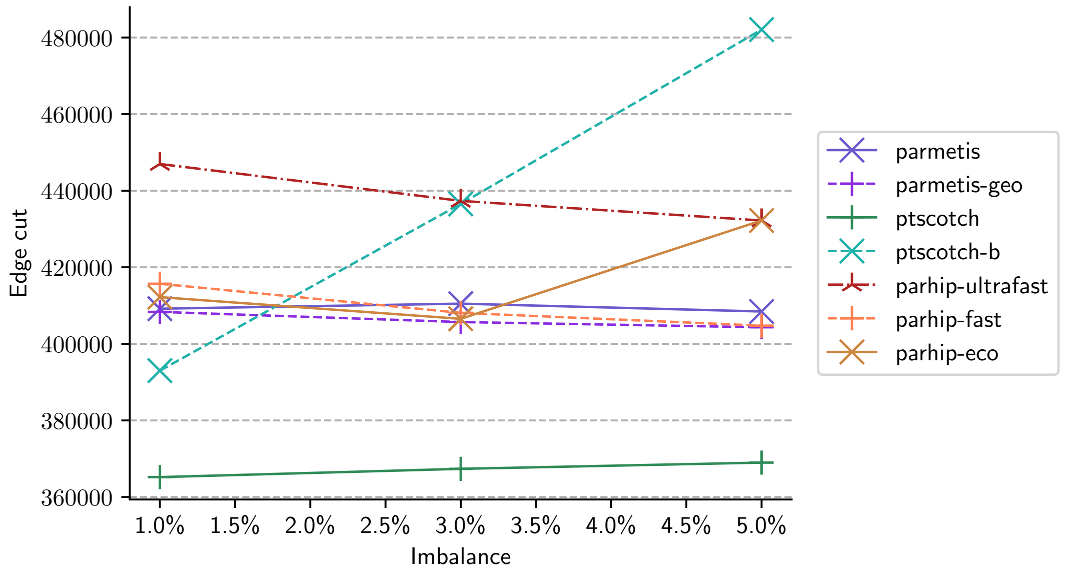

Figure 6.3: A comparison of the maximum allowed imbalance values on the resulting edge cut. Results of one run on the mesh `loh1-large`, 256 parts.

Number of parts	Cells per part
16	453000
256	28300
4096	1770
65536	111

Table 6.6: Comparing the cells per part rate between the different number of partition parts, rounded to 3 numbers. We assume that no imbalance is allowed (i.e. $\epsilon = 0$) and use the mesh `loh1-large` with 7252482 cells.

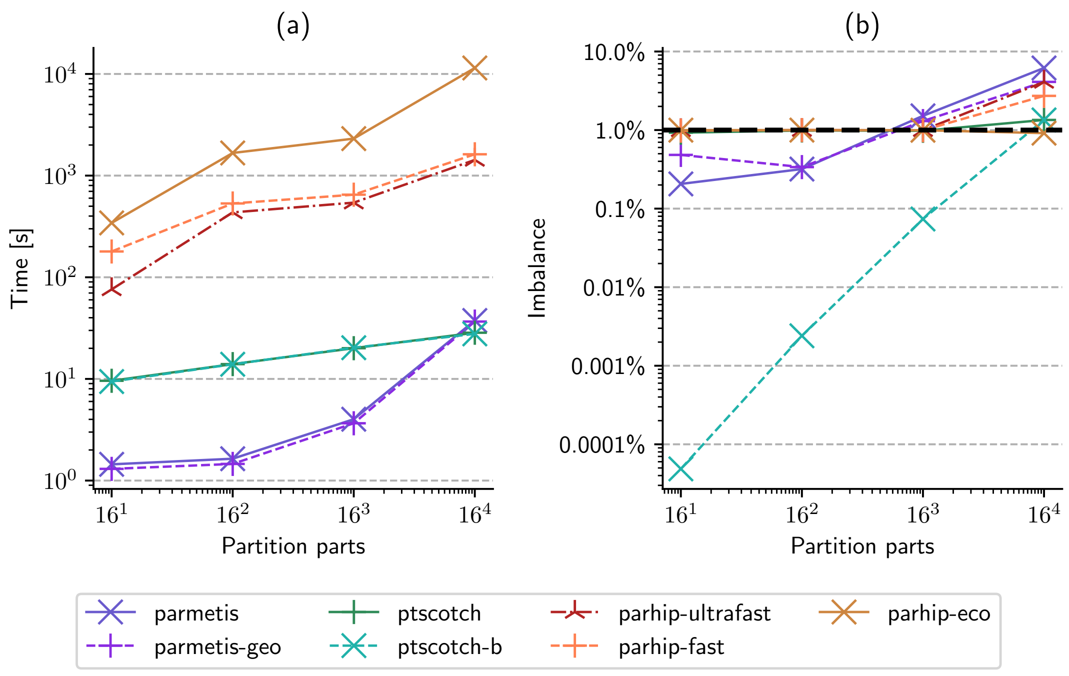

Figure 6.4: A comparison of both time and imbalance to the number of parts we partition into. Plot (a) shows for 16, 256, 4096, and 65536 parts the resulting time the partitioners need. Plot (b) shows the resulting imbalance of the partitions. The bold black line here signifies the maximum allowed imbalance of 1%, as we defined it. We partition the mesh `loh1-large` with roughly 7 million parts, all data points correspond to exactly one run of a partitioner. Note the log-log scaling for *both* graphs.

6.3.3 Seed Variance

In the next test, we ran all partitioners with different seed values, while settings everything else to default values. In total, we have results for 38 different seeds in the case of `parmetis`, `parmetis-geo` and `ptscotch`, 37 different seeds tested for `ptscotch-b`, `parhip-ultrafast`, `parhip-fast`, and 36 different seeds tested for `parhip-eco`. The difference in the number of tested seeds resulted from job timeouts as well as one out of memory error. In figure 6.5, we present a boxplot of both imbalance (right) and edge cut (left) variation.

In general, we can say that concerning edge cut, ParHIP roughly yields the same value for all seeds. This matches nicely with the algorithms used in KaFFPaE and KaFFPa which should accomplish exactly that. PT-SCOTCH allows a bit more variation in the edge cut, though the balance constraint seems to slightly increase the range of possible values. In contrast to these two libraries, the edge cut when using `parmetis-geo` can improve by a value of up to 30000 using another seed (in our test scenario; and this is an extreme case) which is about 7% less. For `parmetis`, similar results can be seen.

Concerning the allowed imbalance, both PT-SCOTCH (if not focusing on balance) and ParHIP keep almost exactly at the maximum allowed imbalance of 1%. The balance-focused PT-SCOTCH strategies instead stay always close to 0% imbalance. Only ParMETIS stays between 0% and 1%, again with a greater variance.

6.3.4 MPI Processes

The last test changes the number of MPI processes used for testing. We use 64 MPI processes for testing, with 4 processes per node, at maximum. The default behavior of slurm in case of having more than one MPI process (or task) per node is to fill a node first with MPI processes, and then continue with the next one [63]. Hence, we only use one node, until we use more than 4 MPI processes (i.e. no interconnect influence here). In fact, the number of nodes we use is the number of MPI processes divided by 4 in this case (more than 4 MPI processes). Figure 6.6 presents the number of MPI processes plotted against the time for the partitioning process. Concerning ParHIP, the time `parhip-eco` needs approximately halves from 1 MPI process, compared to 8. After that, nothing more happens here in our case. `parhip-fast` and `parhip-ultrafast` also improve a bit, but do so less than `parhip-eco`, and also stop with improving at about 8 MPI processes. The number 8 could be related to the fact that this is the smallest number of processes where we use more than one node for partitioning. PT-SCOTCH almost scales exponentially, but the effect of doubling the MPI processes decreases, the more there are already. ParMETIS seems to scale a bit super-exponentially, at least in the beginning. Later, it resorts to exponential scaling.

6.3.5 Different Meshes

Last but not least, we run our partitioners on different meshes, keeping every other setting the same. Figure 6.7 shows the effect on smaller meshes than `loh1-large`. As

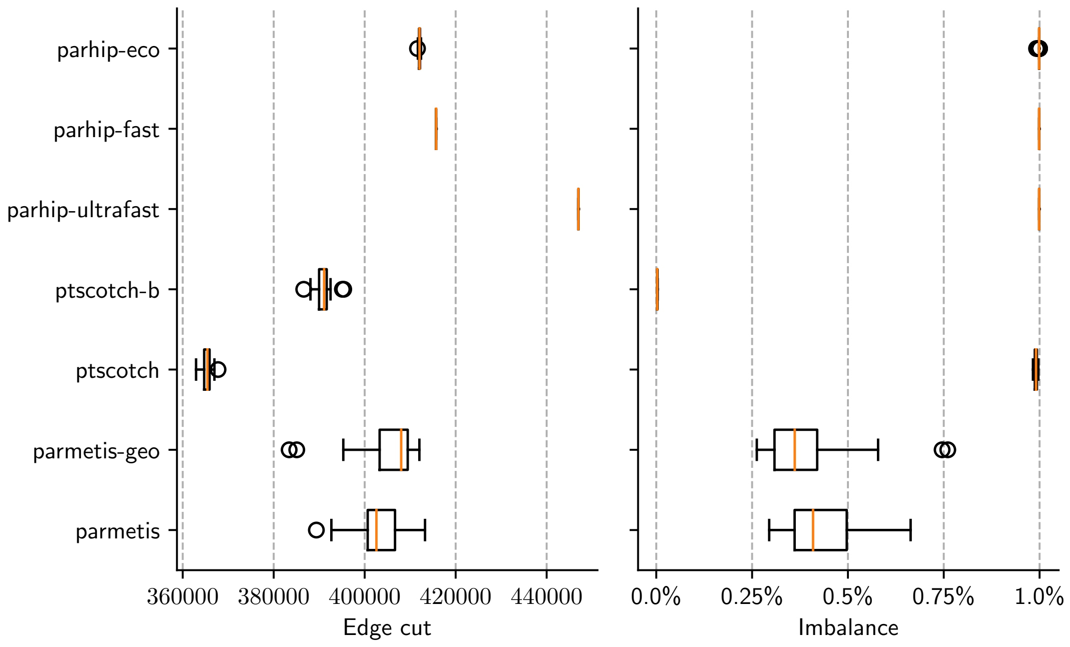

Figure 6.5: A boxplot which shows the influence of the random seed value on the partitioning quality. On the left, we have a plot which shows the variation of the edge cut, applied to the dual graph of the `loh1-large` mesh. On the right, we have the imbalance (allowed: $\leq 1\%$), tested on the same graph. We always partition into 256 parts. The data base is 38 tested seeds for `parmetis`, `parmetis-geo`, and `ptscotch`; 37 tested seeds for `ptscotch-b`, `parhip-ultrafast`, `parhip-fast`, and 36 tested seeds for `parhip-eco`.

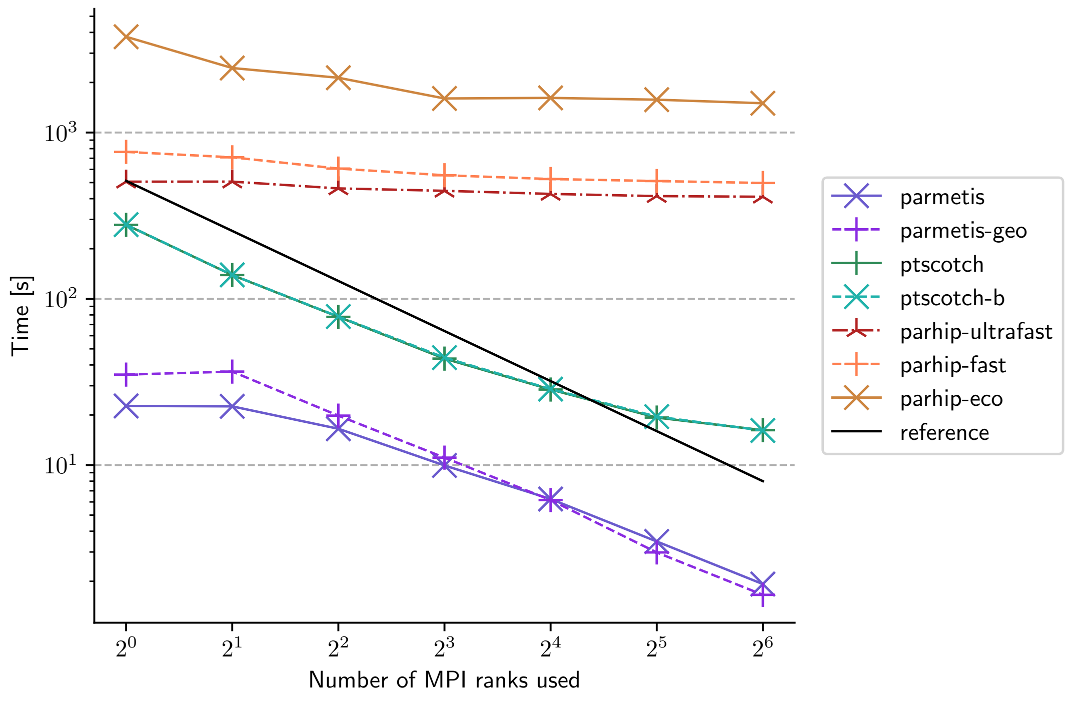

Figure 6.6: The effects of the number of MPI processes on the partitioning time. Tested with four MPI processes per CPU. The data results from one run partitioning the mesh `loh1-large` into 256 parts. The CPU model was Intel Xeon Phi 7210-F. Note the log-log scale. The black reference line shows the slope of a function which would half its time needed for each doubling of the MPI processes. Note that for up to 4 MPI processes, we use only one node. After that, the number of nodes we use doubles with the number of MPI processes.

it can be seen, the edge cut (relative to the best) is not very different for different partitioners, only `parhip-ultrafast` is slightly slower. When it comes to the maximum allowed imbalance, ParMETIS actually violates this constraint. This is again the case because the cell-to-part ratio is rather low here. As usual, `parhip-eco` needs the most time to compute a partition (always more than a minute), while `parhip-fast` and `parhip-ultrafast` stay mostly below one minute of partitioning time. For PT-SCOTCH and ParHIP, the needed time is negligible.

Next, we look at some larger meshes in figure 6.8. Here, every partitioner obeys imbalance constraint. Furthermore, PT-SCOTCH yields the best edge cut in all cases, except with `sumatra-221m`, where it failed because of a lack of memory (as a remainder: we have 1536 GB in total which makes 24 GB per rank). However, this might not pose a problem, as `sumatra-221m` is usually simulated in SeisSol with a much larger number of nodes anyway [10]. In general, the edge cut is again relatively seen not very different for all the larger meshes. ParHIP unfortunately failed to compute partitions for the `landers-98m`, `landers-191m`, and `sumatra-221m` mesh, due to memory problems, hence we can only present the results of `landers-7m` which is roughly the same size as the afore-used `loh1-large`. Again, `parhip-eco` takes by far the longest to partition the mesh with more than 25 minutes, and `parhip-fast` and `parhip-ultrafast` come next (with roughly $100s$ difference in execution time).

For all meshes (except `sumatra-221m`), ParMETIS is several times faster than PT-SCOTCH for the Landers meshes. Likewise, PT-SCOTCH was always several times faster than `parhip-ultrafast` alone.

6.4 Comparison with Claimed Results

After running all these tests, we will now come back and compare to the statements by the papers in section 4.3.

As claimed in [24], PT-SCOTCH was the slowest partitioner which also did not yield competitive edge cuts. We can confirm this for the case that the edges are not uniformly weighted. But in all other cases, PT-SCOTCH outperformed ParHIP and ParMETIS in terms of edge cut, and is still faster than ParHIP. The probably most unexplainable problem we encountered is the huge memory consumption of ParHIP which prevents it from partitioning meshes with 98 million vertices or more, though 1536 GB of memory are in total available. Compared to the test machine in [24], every MPI process has approximately 8 GB more RAM, if equally distributed (24 GB in our configuration, comparing it with the 512 GB-RAM test machine from the paper and 32 MPI processes yields 16 GB per rank). And against the claim that ParMETIS had problems with larger meshes, it is the only partitioning library which manages to partition the 221 million 2004 Sumatra Earthquake mesh. This way, it is the opposite situation compared to the paper [24], where ParHIP managed to partition bigger graphs than ParMETIS. The rest of the claims from [24], we could not reproduce. ParHIP returned in many cases worse edge cuts than ParMETIS did. We could only confirm that `parhip-eco` is the slower than ParMETIS. This may be attributed to a number of factors: first of all, it is not

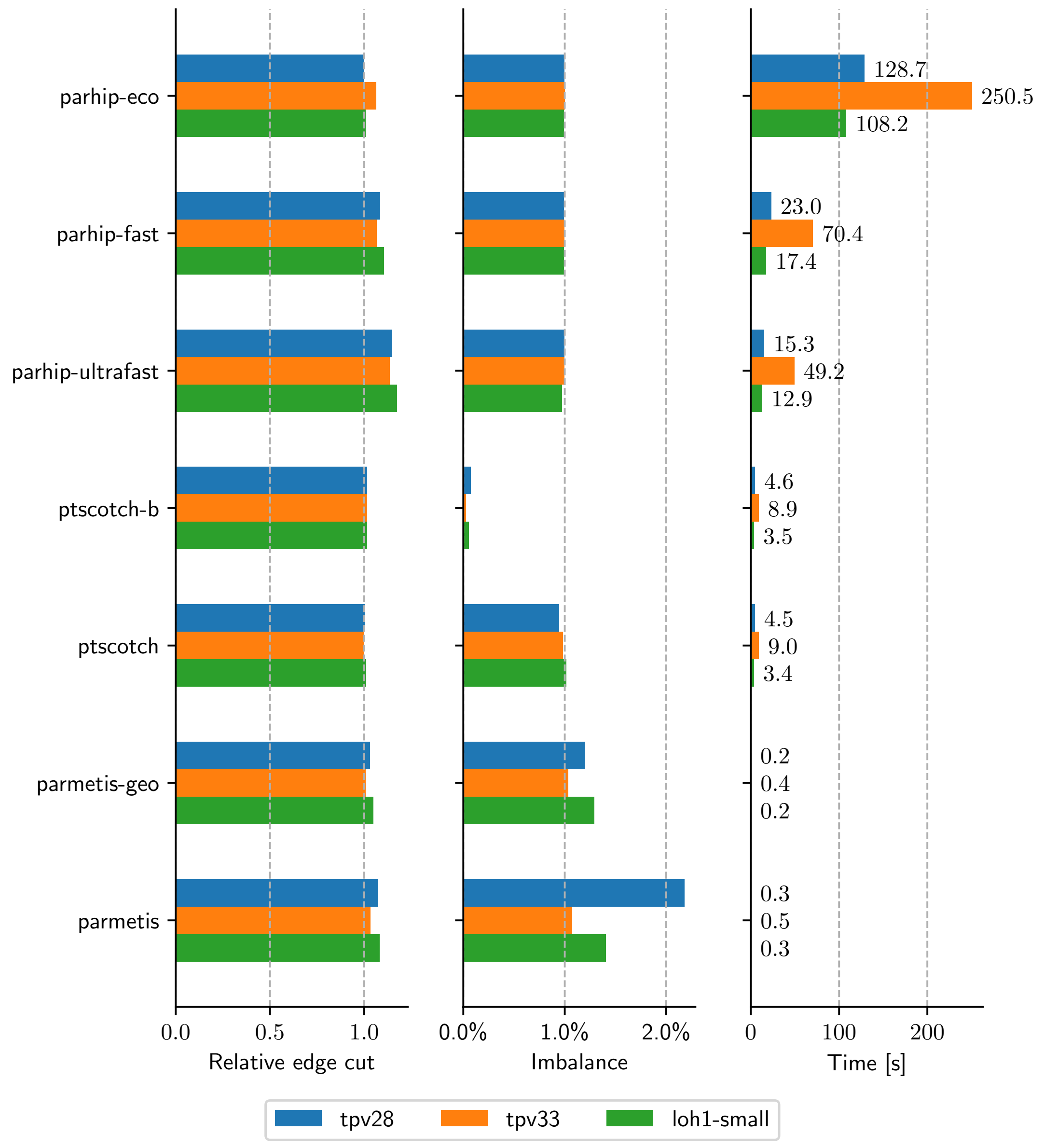

Figure 6.7: Partitioning results of one run on different small meshes (maximum approximately 1 million cells), always partitioning into 256 parts. For all meshes and partitioners, the resulting imbalance, time needed, and the edge cut (divided by the best edge cut for the respective mesh) are displayed.

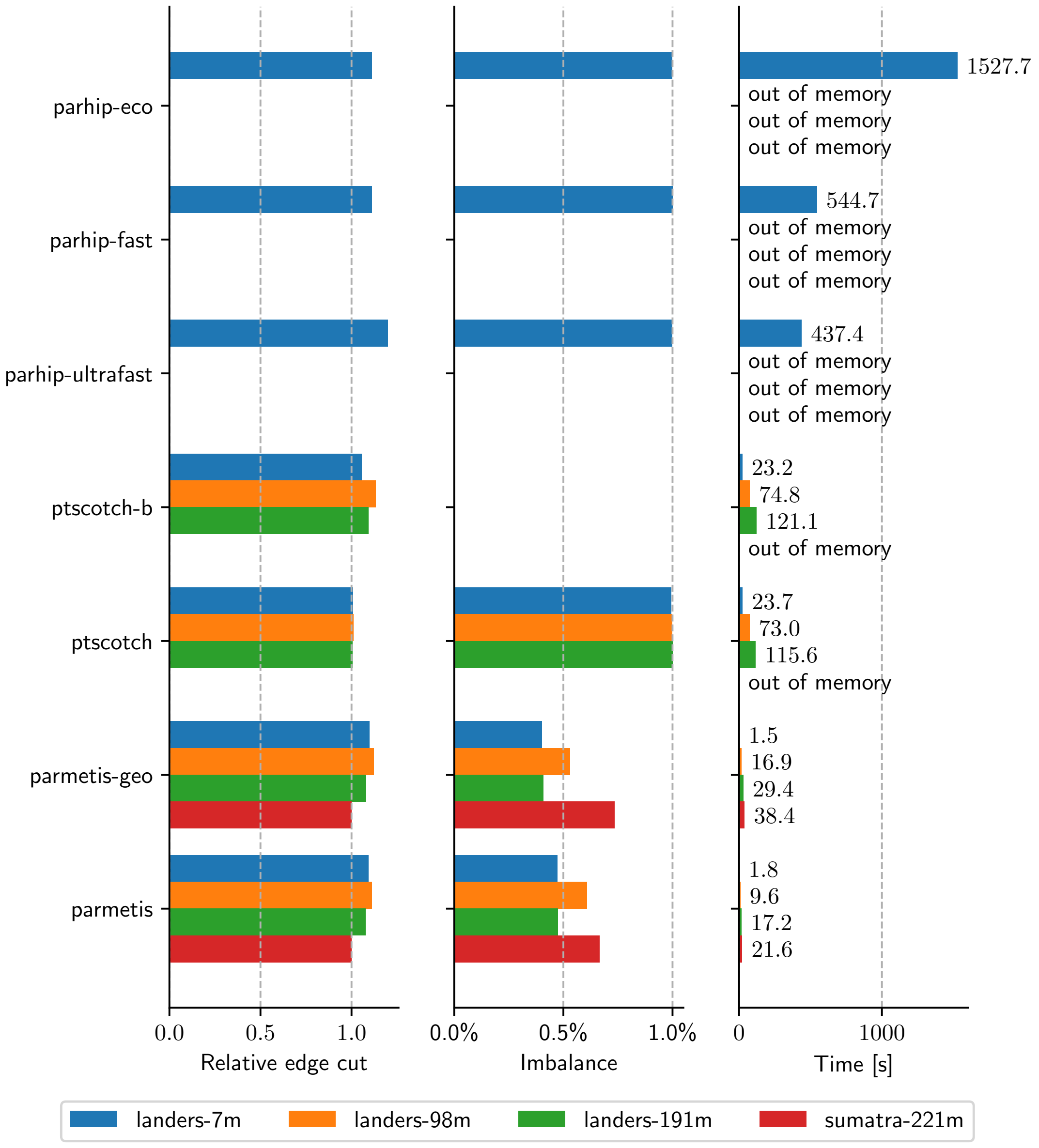

Figure 6.8: Partitioning results of one run on different resolutions of the `landers` mesh, partitioning into 256 parts. For all meshes and partitioners, the resulting imbalance, time needed, and the edge cut (divided by the best edge cut for the respective mesh) are displayed. Note that for `landers-98m`, `landers-191m`, and `sumatra-221m` ParHIP did not manage to partition these due to memory problems. PT-SCOTCH ran out of memory when partitioning `sumatra-221m`. ParMETIS had no problems in this regard.

excluded that our interface does not set some variables correctly. We tried to adhere to the code provided in the package as good as possible, but even then, small errors are possible. Secondly, we tested different graphs, and use an entirely different cluster and test setup.

Concerning [52], we can confirm that PT-SCOTCH returns a better edge cut than ParMETIS, when using a considerable amount of MPI processes (which means 64 in our case). Also, ParMETIS is seen to be always much faster than PT-SCOTCH during partitioning, conforming with the paper. Though it should be noted that PT-SCOTCH might perform faster, if multithreading with pthreads was enabled.

7 Summary and Outlook

All in all, it can be said that there is no partitioner which outperforms the other partitioners in all categories. ParMETIS is well-suited when it comes to a low run-time, and when obedience to the maximum allowed imbalance is not too important. In comparison, ParHIP still needs development for our cases. First of all, we needed to create a C++ interface by ourselves. In our tests, ParHIP had a high run-time and memory consumption, and its partition quality is improvable. Yet, it could again become a viable option, in case edge weights ever come into play. PT-SCOTCH worked extremely well for our meshes. Albeit a bit slower than ParMETIS, its configurability (e.g. for balance) and its partitioning quality surpassed both ParMETIS and ParHIP. As long as uniform edge weights are used, it performs best among all the partitioners we tested. Yet, if non-uniform edge weights come into play, PT-SCOTCH stays by far behind its competitors in run-time alone.

We also created an interface for different partitioners in PUML, and implemented the three partitioning libraries ParMETIS, PT-SCOTCH and ParHIP for it. While implementing the interface, we also replicated the mesh-to-dual-graph conversion which was present only in ParMETIS. For doing so, we abstracted an algorithm already used in SeisSol and implemented it as well for the PUML library. This abstraction can be used to simplify the existing code in SeisSol when working with PUML.

For improving the partitioning in the future, an idea would be to try to "chain" different partitioning algorithms; if that might yield some improvement, i.e. using partition refinement methods (as present in e.g. ParMETIS) on some partition computed by some other partitioner. Furthermore, an implementation of ZOLTAN might be thinkable to bring in some more partitioning algorithms to try out. Apart from that, it might be beneficial to look at other weighting models for vertices and maybe also edges for SeisSol. This could help to further improve the load balancing.

References

[1] George Karypis. *METIS – Serial Graph Partitioning and Fill-reducing Matrix Ordering.* URL: http://glaros.dtc.umn.edu/gkhome/metis/metis/overview (visited on 03/13/2019).

[2] George Karypis. *ParMETIS – Parallel Graph Partitioning and Fill-reducing Matrix Ordering.* URL: http://glaros.dtc.umn.edu/gkhome/metis/parmetis/overview (visited on 03/13/2019).

[3] François Pellegrini. *SCOTCH: Static Mapping, Graph, Mesh and Hypergraph Partitioning, and Parallel and Sequential Sparse Matrix Ordering Package.* URL: https://www.labri.fr/perso/pelegrin/scotch/ (visited on 03/13/2019).

[4] Sandia National Laboratories. *Zoltan: Parallel Partitioning, Load Balancing and Data-Management Services.* URL: http://www.cs.sandia.gov/Zoltan/Zoltan.html (visited on 03/13/2019).

[5] Umit Catalyurek. *PaToH v3.0.* URL: https://web.archive.org/web/20080709035220/http://bmi.osu.edu/~umit/software.htm (visited on 03/13/2019).

[6] Bruce A. Hendrickson and Robert Leland. *Chaco: Algorithms and Software for Partitioning Meshes.* URL: http://www.cs.sandia.gov/CRF/chac_p2.html (visited on 03/13/2019).

[7] Robert Preis. *PARTY Partitioning Library.* URL: https://web.archive.org/web/20160420163016/http://www2.cs.uni-paderborn.de/cs/ag-monien/PERSONAL/ROBSY/party.html (visited on 03/13/2019).

[8] Chris Walshaw. *JOSTLE – graph partitioning software.* URL: http://chriswalshaw.co.uk/jostle/ (visited on 03/13/2019).

[9] Christian Schulz. *KaHIP – Karlsruhe High Quality Partitioning.* URL: http://algo2.iti.kit.edu/kahip/ (visited on 03/13/2019).

[10] Carsten Uphoff et al. "Extreme Scale Multi-physics Simulations of the Tsunamigenic 2004 Sumatra Megathrust Earthquake". In: *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis.* SC '17. Denver, Colorado: ACM, 2017, 21:1–21:16. ISBN: 978-1-4503-5114-0. DOI: 10.1145/3126908.3126948. URL: http://doi.acm.org/10.1145/3126908.3126948.

[11] Alexander Breuer, Alexander Heinecke, and Michael Bader. "Petascale Local Time Stepping for the ADER-DG Finite Element Method". In: *2016 IEEE International Parallel and Distributed Processing Symposium (IPDPS)* (2016), pp. 854–863. URL: http://dx.doi.org/10.1109/IPDPS.2016.109.

[12] Sebastian Rettenberger et al. *PUML source code repository.* URL: `https://github.com/TUM-I5/PUML2` (visited on 03/13/2019).

[13] The SeisSol Team. *SeisSol Project Homepage.* URL: `http://www.seissol.org` (visited on 03/13/2019).

[14] The SeisSol Team. *SeisSol source code repository.* URL: `https://github.com/SeisSol/SeisSol` (visited on 03/13/2019).

[15] A. Heinecke et al. “Petascale High Order Dynamic Rupture Earthquake Simulations on Heterogeneous Supercomputers”. In: *SC ’14: Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis.* Nov. 2014, pp. 3–14. DOI: `10.1109/SC.2014.6`.

[16] Martin Käser and Michael Dumbser. “An arbitrary high-order discontinuous Galerkin method for elastic waves on unstructured meshes — I. The two-dimensional isotropic case with external source terms”. In: *Geophysical Journal International* 166.2 (Aug. 2006), pp. 855–877. ISSN: 0956-540X. DOI: `10.1111/j.1365-246X.2006.03051.x`. eprint: `http://oup.prod.sis.lan/gji/article-pdf/166/2/855/1514333/166-2-855.pdf`. URL: `https://dx.doi.org/10.1111/j.1365-246X.2006.03051.x`.

[17] Sebastian Rettenberger. “Scalable I/O on Modern Supercomputers for Simulations on Unstructured Meshes”. Dissertation. Department of Informatics, Technical University of Munich, 2018. ISBN: 9783843935869.

[18] Peter Wauligmann. “Parallel Construction of Unstructured Mesh Data Structures using OpenMP and MPI”. Bachelor’s thesis. Institut für Informatik, Technische Universität München, Mar. 2018.

[19] M. R. Garey, D. S. Johnson, and L. Stockmeyer. “Some Simplified NP-complete Problems”. In: *Proceedings of the Sixth Annual ACM Symposium on Theory of Computing.* STOC ’74. Seattle, Washington, USA: ACM, 1974, pp. 47–63. DOI: `10.1145/800119.803884`. URL: `http://doi.acm.org.eaccess.ub.tum.de/10.1145/800119.803884`.

[20] Konstantin Andreev and Harald Räcke. “Balanced Graph Partitioning”. In: *Theor. Comp. Sys.* 39.6 (Nov. 2006), pp. 929–939. ISSN: 1432-4350. DOI: `10.1007/s00224-006-1350-7`. URL: `http://dx.doi.org/10.1007/s00224-006-1350-7`.

[21] Andreas Emil Feldmann. “Fast Balanced Partitioning of Grid Graphs is Hard”. In: *CoRR* abs/1111.6745 (2011). arXiv: `1111.6745`. URL: `http://arxiv.org/abs/1111.6745`.

[22] Aydin Buluç et al. “Recent Advances in Graph Partitioning”. In: *CoRR* abs/1311.3144 (2013). arXiv: `1311.3144`. URL: `http://arxiv.org/abs/1311.3144`.

[23] George Karypis and Vipin Kumar. “Kumar, V.: Parallel Multilevel k-way Partitioning Scheme for Irregular Graphs. SIAM Review 41(2), 278-300”. In: *Siam Review - SIAM REV* 41 (June 1999), pp. 278–300. DOI: `10.1137/S0036144598334138`.

[24] Henning Meyerhenke, Peter Sanders, and Christian Schulz. "Parallel Graph Partitioning for Complex Networks". In: vol. 28. 9. 2017, pp. 2625–2638.

[25] Bruce Hendrickson and Robert Leland. "A Multilevel Algorithm for Partitioning Graphs". In: *Proceedings of the 1995 ACM/IEEE Conference on Supercomputing*. Supercomputing '95. San Diego, California, USA: ACM, 1995. ISBN: 0-89791-816-9. DOI: `10.1145/224170.224228`. URL: `http://doi.acm.org.eaccess.ub.tum.de/10.1145/224170.224228`.

[26] Chris Walshaw. "Multilevel Refinement for Combinatorial Optimisation Problems". In: *Annals of Operations Research* 131.1 (Oct. 2004), pp. 325–372. ISSN: 1572-9338. DOI: `10.1023/B:ANOR.0000039525.80601.15`. URL: `https://doi.org/10.1023/B:ANOR.0000039525.80601.15`.

[27] Peter Sanders and Christian Schulz. "Engineering Multilevel Graph Partitioning Algorithms". In: *Algorithms – ESA 2011*. Ed. by Camil Demetrescu and Magnús M. Halldórsson. Berlin, Heidelberg: Springer Berlin Heidelberg, 2011, pp. 469–480. ISBN: 978-3-642-23719-5.

[28] Usha Nandini Raghavan, Réka Albert, and Soundar Kumara. "Near linear time algorithm to detect community structures in large-scale networks". In: *Phys. Rev. E* 76 (3 Sept. 2007), p. 036106. DOI: `10.1103/PhysRevE.76.036106`. URL: `https://link.aps.org/doi/10.1103/PhysRevE.76.036106`.

[29] George Karypis and Vipin Kumar. "A Fast and High Quality Multilevel Scheme for Partitioning Irregular Graphs". In: *SIAM J. Sci. Comput.* 20.1 (Dec. 1998), pp. 359–392. ISSN: 1064-8275. DOI: `10.1137/S1064827595287997`. URL: `http://dx.doi.org/10.1137/S1064827595287997`.

[30] Christian Schulz. "High Quality Graph Partitioning". Dissertation. Karlsruhe Institute of Technology, 2013. ISBN: 978-3844264623.

[31] John R. Pilkington and Scott B. Baden. *Partitioning with Spacefilling Curves*. Tech. rep. Apr. 1994.

[32] B. W. Kernighan and S. Lin. "An efficient heuristic procedure for partitioning graphs". In: *The Bell System Technical Journal* 49.2 (Feb. 1970), pp. 291–307. ISSN: 0005-8580. DOI: `10.1002/j.1538-7305.1970.tb01770.x`.

[33] C. M. Fiduccia and R. M. Mattheyses. "A Linear-Time Heuristic for Improving Network Partitions". In: *19th Design Automation Conference*. June 1982, pp. 175–181. DOI: `10.1109/DAC.1982.1585498`.

[34] François Pellegrini. "Contributions au partitionnement de graphes parallèle multi-niveaux / Contributions to parallel multilevel graph partitioning". Habilitation. 2009.

[35] George Karypis et al. *METIS 5.1.0 source code archive*. URL: `http://glaros.dtc.umn.edu/gkhome/fetch/sw/metis/metis-5.1.0.tar.gz` (visited on 03/13/2019).

[36] Dominique LaSalle and George Karypis et al. *mtMETIS 0.6.0 source code archive.* URL: http://glaros.dtc.umn.edu/gkhome/fetch/sw/metis/mt-metis-0.6.0.tar.gz (visited on 03/13/2019).

[37] George Karypis and Kirk Schloegel. *ParMETIS – Parallel Graph Partitioning and Sparse Matrix Ordering Library.* Manual. University of Minnesota, Department of Computer Science and Engineering. URL: http://glaros.dtc.umn.edu/gkhome/fetch/sw/parmetis/manual.pdf (visited on 03/13/2019).

[38] Kirk Schloegel and George Karypis et al. *ParMETIS 4.0.3 source code archive.* URL: http://glaros.dtc.umn.edu/gkhome/fetch/sw/parmetis/parmetis-4.0.3.tar.gz (visited on 03/13/2019).

[39] *hMETIS – A Hypergraph Partitioning Package.* Manual. University of Minnesota, Department of Computer Science, Engineering, and Army HPC Research Center. URL: http://glaros.dtc.umn.edu/gkhome/fetch/sw/hmetis/manual.pdf (visited on 03/13/2019).

[40] George Karypis. *hMETIS – Hypergraph and Circuit Partitioning.* URL: http://glaros.dtc.umn.edu/gkhome/metis/hmetis/overview (visited on 03/13/2019).

[41] Christian Schulz et al. *KaHIP source code repository.* URL: https://github.com/schulzchristian/KaHIP (visited on 03/13/2019).

[42] François Pellegrini et al. *SCOTCH source code repository.* URL: https://gforge.inria.fr/frs/?group_id=248 (visited on 03/13/2019).

[43] Sandia National Laboratories. *ZOLTAN 3.83 source code archive.* URL: http://www.cs.sandia.gov/~kddevin/Zoltan_Distributions/zoltan_distrib_v3.83.tar.gz (visited on 03/13/2019).

[44] Sebastian Schlag. *KaHyPar - Karlsruhe Hypergraph Partitioning.* URL: http://kahypar.org/ (visited on 03/13/2019).

[45] Sebastian Schlag et al. *KaHyPar source code repository.* URL: https://github.com/SebastianSchlag/kahypar (visited on 03/13/2019).

[46] Aleksandar Trifunovic. *Parallel Hypergraph Partitioning Software.* URL: https://web.archive.org/web/20170420183852/http://www.doc.ic.ac.uk/~at701/parkway/ (visited on 03/13/2019).

[47] Trifunovic et al. Aleksandar. *Parkway 2.11 source code archive.* URL: https://web.archive.org/web/20170420183852/http://www.doc.ic.ac.uk/~at701/parkway/parkway-2.11-linux.tar.gz (visited on 03/13/2019).

[48] Bruce A. Hendrickson and Robert Leland et al. *Chaco 2.2 source code archive.* URL: https://www3.cs.stonybrook.edu/~algorith/implement/chaco/distrib/Chaco-2.2.tar.gz (visited on 03/13/2019).

[49] Peter Sanders and Christian Schulz. "Think Locally, Act Globally: Highly Balanced Graph Partitioning". In: *Proceedings of the 12th International Symposium on Experimental Algorithms (SEA'13).* Vol. 7933. LNCS. Springer, 2013, pp. 164–175.

[50] François Pellegrini et al. *PT-Scotch and libPTScotch 6.0 User's Guide*. Manual. Université Bordeaux 1, LaBRI and INRIA Bordeaux Sud-Ouest. URL: https://gforge.inria.fr/docman/view.php/248/8261/ptscotch_user6.0.pdf (visited on 03/13/2019).

[51] Manuel Holtgrewe, Peter Sanders, and Christian Schulz. "Engineering a scalable high quality graph partitioner". In: *Proceedings of the 24th International Parallal and Distributed Processing Symposium*. 2010, pp. 1–12.

[52] D. Lasalle and G. Karypis. "Multi-threaded Graph Partitioning". In: *2013 IEEE 27th International Symposium on Parallel and Distributed Processing*. May 2013, pp. 225–236. DOI: 10.1109/IPDPS.2013.50.

[53] Kirk Schloegel, George Karypis, and Vipin Kumar. "Parallel static and dynamic multi-constraint graph partitioning". In: *Concurrency and Computation: Practice and Experience* 14 (2002), pp. 219–240.

[54] Peter Sanders and Christian Schulz. *KaHIP v2.10 – Karlsruhe High Quality Partitioning*. Manual. Karlsruhe Institute of Technology and University of Vienna. URL: http://algo2.iti.kit.edu/schulz/software_releases/kahipv2.10.pdf (visited on 03/13/2019).

[55] Peter Sanders and Christian Schulz. "Distributed Evolutionary Graph Partitioning". In: *2012 Proceedings of the Fourteenth Workshop on Algorithm Engineering and Experiments (ALENEX)*, pp. 16–29. DOI: 10.1137/1.9781611972924.2. eprint: https://epubs.siam.org/doi/pdf/10.1137/1.9781611972924.2. URL: https://epubs.siam.org/doi/abs/10.1137/1.9781611972924.2.

[56] François Pellegrini et al. *Scotch and libScotch 6.0 User's Guide*. Manual. Université Bordeaux 1, LaBRI and INRIA Bordeaux Sud-Ouest. URL: https://gforge.inria.fr/docman/view.php/248/8260/scotch_user6.0.pdf (visited on 03/13/2019).

[57] Francois Pellegrini. "Distillating knowledge about SCOTCH". In: *Combinatorial Scientific Computing*. Ed. by Uwe Naumann et al. Dagstuhl Seminar Proceedings 09061. Dagstuhl, Germany: Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik, Germany, 2009. URL: http://drops.dagstuhl.de/opus/volltexte/2009/2091.

[58] Oren Ben-Kiki, Clark Evans, and Brian Ingerson. *YAML Ain't Markup Language (YAML) (tm) Version 1.2*. Tech. rep. YAML.org, Sept. 2009. URL: http://www.yaml.org/spec/1.2/spec.html.

[59] Carsten Uphoff. *easi Documentation*. URL: https://media.readthedocs.org/pdf/easyinit/latest/easyinit.pdf (visited on 03/13/2019).

[60] The Stackoverflow Community. *Trouble Understanding MPI_Type_create_struct*. URL: https://stackoverflow.com/questions/33618937/trouble-understanding-mpi-type-create-struct (visited on 03/13/2019).

[61] P. Hoffman and J. Snell. *JSON Merge Patch*. RFC 7396. RFC Editor, Oct. 2014.

[62] Leibniz-Rechenzentrum. *Many-Core Cluster CooLMUC-3 at LRZ.* URL: `https://www.lrz.de/services/compute/linux-cluster/coolmuc3/overview_en/` (visited on 03/13/2019).

[63] The Slurm Team and SchedMD. *Slurm Workload Manager – sbatch.* URL: `https://slurm.schedmd.com/sbatch.html` (visited on 03/15/2019).

List of Figures

List of Tables

List of Algorithms