Open Benchmarks for Load Balancing Heuristics in Parallel Adaptive Finite Element Computations*

Oliver Marquardt  
Paderborn Center for Parallel Computing (PC²)  
Universität Paderborn  
Fürstenallee 11, D-33102 Paderborn  
marquard@uni-paderborn.de

Stefan Schamberger  
Fakultät für Elektrotechnik, Informatik und Mathematik  
Universität Paderborn  
Fürstenallee 11, D-33102 Paderborn  
schaum@uni-paderborn.de

ABSTRACT Load balancing plays an important role in large parallel numerical simulations. To address this problem, some general purpose libraries as well as a number of more specific approaches have been developed, and results on their performance and solution quality have been published. However, the evaluations in most of these publications are based on an application and problem instance combination that is very hard to come by if not impossible to obtain. This is a dilemma, because it means that results of newly developed approaches cannot be compared with published results what makes the latter less useful than they could be.

The solution is of course to create a benchmark set that is open to the public. In this paper we describe a way to build a load balancing benchmark for general purpose libraries operating on graphs, and enumerate the required information, such that also existing applications might be modified to output the necessary data. Furthermore, we present a way to generate simple FEM related instances for testing purposes, which might form the basis of a future benchmark set. Applying the proposed methods, we run experiments with the two parallel state-of-the-art libraries Metis and Jostle and a prototype of the PartyDB library, and exemplarily compare their solutions according to several metrics and norms.

I. INTRODUCTION

Finite Element Methods (FEM) are used extensively by engineers to analyse a variety of physical processes which can be expressed via Partial Differential Equations (PDE). The domain on which the PDEs have to be solved is discretized into a mesh, and the PDEs are transformed into a set of linear equations defined on the mesh’s elements (see e.g. [1]). These can then be solved by iterative methods such as Conjugate Gradient (CG) and Multigrid. Due to the very large amount of elements needed to obtain an accurate approximation of the original problem, this method has become a classical application for parallel computers. The parallelization of numerical simulation algorithms usually follows the Single-Program Multiple-Data (SPMD) paradigm: Each processor executes the same code on a different part of the data. This means that the mesh has to be split into $P$ sub-domains and each sub-domain is then assigned to one of the $P$ processors. To minimize the overall computation time, all processors should thereby roughly contain the same amount of elements. Since iterative solution algorithms perform mainly local operations, i.e. data dependencies are defined by the mesh, the parallel algorithm mainly requires communication at the partition boundaries. Hence, these should be as small as possible. The described problem can be expressed as a graph partitioning problem and existing state-of-the-art libraries to solve it are listed later on.

Depending on the application, some areas of the simulation space require a higher resolution and therefore more elements. Since the location of these areas is not known beforehand or can even vary over time, the mesh is refined and coarsened during the computation. However, this can cause an imbalance between the processors’ load and therefore delay the simulation. To avoid this, the distribution of elements needs to be rebalanced. The application is interrupted and the at this program state static repartitioning problem is solved. To keep the interruption as short as possible, it is necessary to find a new balanced partitioning with small boundaries quickly, with the additional objective not to cause too many elements to change their processor. Migrating elements can be an extremely costly operation since a lot of data has to be sent over communication links and reinserted into complex data structures. Existing approaches addressing the repartitioning problem are referred to in the next section.

Most libraries that solve the partitioning and repartitioning problem involve some kind of heuristic. This is unavoidable due to the complexity of these problems, the large problem sizes and the given time constraints. Even if approximation algorithms are applied for some calculations, the overall computation can still be seen as a heuristic, because the influence between the different components has not been theoretically investigated yet. This means that empirical tests are necessary and are currently the only way to compare the different approaches and implementations.

In the case of graph partitioning, a number of test instances have been made available that can be regarded as benchmarks. One collection can be found e.g. on [2]. These graphs express data dependencies in different kinds of problems like 2- and 3-dimensional FEM simulations, sparse matrix computations or even data mining and acoustics. They therefore reflect a variety of properties and are well suited as test sets.

*This work has been supported by the German Science Foundation (DFG) project SFB-376.
However, for the load balancing problem to our knowledge no equivalent problem instances exist. This might be due to the number of involved components that are needed to run numerical simulations and that are often very problem specific. First of all, some kind of model is needed. We are sure that there exists many detailed and exact geometric representations of technical structures, but this data is usually not freely available. Next, a discretization of the simulation area is required which is usually computed by some meshing software. Of course, an important component is the simulation code itself which is often not available to the public either. It includes the formulation of the problem, the PDEs and a suitable parallel solver for them. Then, an error estimator tells the simulation which parts of the mesh have to be refined, and an adaptation or re-meshing algorithm is needed to perform the necessary modifications. Finally, the load balancing algorithm can be invoked. If one of the described components is missing, it is not possible to rerun the simulation to test load balancing strategies, identify their problems and make improvements nor to perform comparisons with previously published results.

Due to these reasons, there is the need for load balancing benchmarks. These should be freely available and cover the whole range of geometric models, the different existing meshing software and mathematical simulation code. To create a large variety of problem instances that reflect the many different properties of the possible simulations, the help of the community is needed. Only then it is possible to obtain problem instances that are widely accepted as benchmarks. In this paper we demonstrate that it does not take too much effort to collect data for a repartitioning benchmark and exemplarily describe the necessary modifications to an existing FEM framework. To obtain further examples and illustrate how the proposed method works, we present a basic but extendible mesh sequence generator. Based on its output, we perform experiments with some well known load balancing libraries and present the results. Since the overall run-time of applications depends on a variety of factors, we also discuss several useful metrics to compare the partition quality.

The remaining part of this paper is organized as follows. The next section proposes a load balancing benchmark generation scheme. The created instances contain all the necessary data such that any subsequent experiments can be performed independently of the originating application. We also present a basic generator that we developed to be able to provide first examples. We then demonstrate how comparisons between different libraries look alike and present one of the numerical results according to various metrics in detail.

II. BENCHMARKS

This section describes the necessary data for the benchmark scheme we propose in this paper. We first specify the minimal required information and then show how visual outputs can be generated by providing a little extra data. Furthermore, we present a basic though expandable benchmark generator.

A. Graph Sequences and Transitions

While a graph partitioning benchmark only consists out of a single graph, the evaluation of load balancing heuristics requires a sequence of them. This sequence reflects the changes of the mesh caused by the refinement and coarsening procedure and each graph, also called frame, reflects the static mesh at that point when the load balancing algorithm is started. Between two balancing steps, the mesh is adapted to suit the new conditions. If this is performed in parallel, new elements usually belong to the partition they have been created on. To be able to provide this information independently of a given partitioning, each of the inserted elements must be assigned to one element of the preceding graph. This allows to place the new objects onto the same processor as their predecessors and therefore to model the parallel behavior, even if the actual partitioning is not known. Since elements can also be removed and the order of existing elements can change, it is necessary to assign a predecessor to every element of the new mesh. We refer to this data as transition information.

Figure 1 gives an example. On the left, two consecutive frames of the mesh are shown. Their dual graph, which forms the input for the load balancers, is displayed next to them. Note, that the graphs’ vertices are numbered continuously from 0 to |V| and that a vertex’s number can be different in the following frame, although it itself has not been altered.

This approach has several advantages. First of all, one need not to fulfill all the requirements that are necessary to run a simulation as described before. Second, the evaluations can be performed faster since no mathematical computations or meshing operations have to be performed. Furthermore, tests can be made with a larger variety of options that might have not been possible before. For example, the large amount of numerical data in a simulation prevents the use of common hardware that now is sufficient.

Summarized, the minimal required information for a benchmark consists of a number of graphs and the interleaving transition information. Optionally, it is possible to provide some more information to draw the graph or even the originating mesh. Visual evaluation of the results sometimes reveals important deficiencies and possible improvements and should not be underestimated. The easiest way is to provide vertex coordinates for the graph which is a requirement when testing geometric approaches.

More element information can
be included to enhance the visual presentation as described exemplarily in the next subsection.

**B. A Basic Generator**

During the development process of a load balancing heuristic, many single features and their combinations have to be tested. Some of them only consider special cases that rarely occur in real simulations but that are nevertheless important to be handled correctly to ensure the stability of the partitioning and load balancing library. To be able to generate the required constellations, we created a basic tool that outputs artificial mesh sequences. Furthermore, it has the advantage of being independent of the availability of models and mathematical code.

The applied mesh generation process is based on a known refinement scheme. Starting with a 2-dimensional square divided into two triangles, the latter can be refined by always inserting a new vertex at a triangle's base (which is the longest edge). Since this scheme is regular, we obtain meshes with angles of 45 and 90 degree only. Before a triangle is split, we check if this leads to a T-intersection. In this case, the affected larger triangle is split first, which of course is a recursive process. Merging is also possible if none of the anticipating triangles is required anymore to avoid T-intersections. Note, that the whole refinement tree is stored at all time and that the triangles of the mesh are the leaves of this tree.

To determine which triangle to split, we insert so called attractors. These are currently implemented as points, but other geometric shapes are possible. All triangles compute the ratio of their area and the distance to the closest attractor. These values are stored in a priority queue, such that always that triangle with the largest ratio can be determined and be selected for refinement next. A second queue holds triangles for merging. Furthermore, we use a geometry to cut areas out of the mesh. If a triangle is fully covered by the geometry, it is marked inactive and will no longer be part of the mesh. Currently, the geometry is limited to circle sets, but extensions are also possible here.

The means described so far allow us to generate a static mesh. To create a sequence of meshes with small changes between them, traces are assigned to both, attractors and the geometry. A trace describes how an object changes its position over time. Furthermore, the attractors can vary their weight and the global refinement threshold can be adjusted via a parameter database. After the adjustments have taken place, all values are updated, and triangles that are now too large are split and too small ones are merged.

It is obvious that the generated meshes are unsuitable for mathematical computations. A closer look at the border of the circles in figure 2 reveals that the circle surface is not approximated smoothly at all what for example will cause a fluid dynamics simulation to fail. However, since the graph partitioning and load balancing algorithms only consider the dual graph of the mesh they are not affected. Our experiments with meshes generated this way show, that the created instances reflect the properties and behavior of real problem instances well enough to lead to meaningful results.

To record the transition information described in the previous section we proceed as follows. Before each time step, all triangles of the mesh are assigned a serial number. The dual graph is then saved on disk with the vertices stored in their assigned serial order. If a triangle is split, both children of this triangle inherit its number. When merging two triangles, the number of the left triangle is assigned. After the mesh adaption, the old serial numbers now contain the information about the predecessor of each vertex in the previous graph. This is exactly the data that is required as transition information. Before it can be saved on disk, the new serial numbers need to be assigned that match the adapted graph to ensure the same vertex order in the transition file and the new graph.

Due to geometry movements, it can happen that triangles marked inactive become active again, e.g. after a circle has changed its position. These leaves do not have a valid transition number. To handle this case, all triangles with valid numbers propagate their data in the tree towards the root with the left child of an inner node having priority. In a second step, triangles without valid numbers are assigned the same number as their parent. This ensures that the uncovered triangles will be placed in the same partition as one of their neighbors.

To display our mesh, we created additional files that contain the three triangle coordinates according to the vertices of each dual graph. With this additional geometric information we are able to generate the postscript images that are e.g. presented in figure 2 in section III.

**C. Modification of a Real World Mesh Application**

To demonstrate the easiness of modifying a real world application to output graph transitions, we have extended the open FEM framework package padfem\(^2\) [3]. This framework solves partial differential equations on tetrahedral meshes and deterministically generates refined meshes through its parallel adaptation module. Thus, it can be applied to produce graph sequences of 3-dimensional meshes. For the graph partitioning process, every tetrahedron of the mesh is associated with a vertex of the volume graph which forms the input data for the partitioning and re-partitioning algorithms.

In padfem\(^2\), the geometric representation and information about regular refined tetrahedra and their closure during the adaptation process is stored in a distributed mesh database. Every time a tetrahedron is modified by a refinement rule, the database stores the unique identifiers of the old tetrahedron and the newly created ones into a lookup table using the identifiers of the new tetrahedra as lookup keys.

The refinement process consists out of two important operations [4]. The first one is the regular or so-called red refinement of a tetrahedron into eight sub-tetrahedra, with its associated green closure refinement that creates less than eight sub-tetrahedra from neighboring instances to avoid hanging nodes. Obviously, the predecessor of all new sub-tetrahedra is the origin tetrahedron. Hence, we simply obtain a 1 : \(m\) identifier mapping. The second operation is the reverse step, namely the merging or so-called coarsening of several
tetrahedra into a single one. In this case, we have a $m:1$ identifier mapping. Unfortunately, we cannot easily determine a dedicated predecessor tetrahedron, because in contrast to the 2-dimensional case described before there is no left or right object, but many different constellations exist. However, since we do not require a bijective mapping of tetrahedra identifiers to transition identifiers, it is sufficient to assign the smallest identifier of all replaced tetrahedra as the predecessor of the combined one. During the adaption process, we record the tree-like history of the whole coarsening process to know which tetrahedra can form a coarse one. $padfem^2$ updates this history data automatically and also stores the transitions between the old and new states.

Provided with such a lookup table, we can easily generate graph sequences with their transition information. After the adaptation phase the mesh is dumped to a file. Additionally, the contents of the lookup table is traversed for every tetrahedron of the mesh to check if it has been modified. If so, we write the origin tetrahedron identifier to the transition file, otherwise the actual tetrahedron identifier is written. If the ordering of the tetrahedra identifiers is not a continuous sequence from 0 to $|V|$, we have to renumber the tetrahedra.

### III. Experiments

This section describes our experiments, defines and discusses the metrics and norms involved in the evaluation and contains the results that we have obtained running the libraries Metis, Jostle and Party/DB on one of our benchmark instances.

All considered benchmarks contain 101 frames, each consisting of around 15000 triangles. Though the instances are quite small, important observations can already be made while we are still able to include an example in this paper. Due to the space limitations, the results of the remaining 2-dimensional experiments as well as the complete data for the benchmarks has to be omitted here but can be found at [5]. Furthermore, we decided to only include the results of 12 partitionings since we have seen that the observations are similar for a different number of parts. Also our first tests with 3-dimensional meshes created with the modified version of the $padfem^2$ environment show comparable results. Unfortunately, we did not have exclusive access to the machines during our experiments and therefore omit the run-time results in this paper. Although we know that different vertex orderings can influence the results [6], we only perform one computation per benchmark due to time restrictions.

The parallel libraries of Metis (version 3.1) [7], [8] and Jostle (version 3.0) [9] offer a large number of options. For the presented evaluation, we chose the recommended values from their manual, respectively, and left the remaining parameters at their default. This means that Metis operates with an $itr$ value of 1000.0 and Jostle uses the options $threshold = 20$, $matching = local$, $imbalance = 3$. Note, that Jostle seems to ignore the imbalance setting and always computes totally balanced partitions, except for the initial solution where the sequential versions of the libraries are applied. In contrast to Metis and Jostle which are both based on the multilevel paradigm [10], the Party/DB library is a prototypic implementation of the partitioning algorithm presented in [11]. It replaces the main operations of the bubble framework [12] through a special diffusion algorithm as described in detail in [13].

To evaluate the quality of a partitioning, a number of metrics are possible. The traditional one is the edge-cut, that is the number of edges between different partitions, but it is known that this usually does not model the real costs [14]. Depending on the application, some of the metrics might be more important than others, and more information is provided if we list them separately. Furthermore, the quality of a partitioning depends on its balance. A less balanced solution does not necessarily cause problems during the computation, but of course allows other metrics to decrease further and makes comparisons less meaningful.

First of all, the run-time of the libraries is important. This of course depends on the underlying hardware. Besides the classical edge cut, we also measure the number of boundary vertices and the communication volume (send and receive), assuming that vertices represent information and edges the communication pattern. These metrics have been also partly been used in e.g. [15], [16]. To rate the shape of a partition, its aspect ratio has shown to be a good indicator [12]. Note, that the shape of a partition plays an important role for the convergence rate of some mathematical approaches like the Domain Decomposition [17], [18]. However, this metric requires geometric information that might not be available. Hence, we compute the partition diameter instead, which also displays if a partition is connected or not. Furthermore, after each balancing step, some vertices have to be migrated to another partition. We account for this as outgoing and incoming migration volume.

Summarized, for each partition $p$ the metrics can be described as follows: external edges: Number of edges that are incident to exactly one vertex of partition $p$. boundary vertices: Number of vertices of partition $p$ that are adjacent to at least one vertex from a different partition. send volume: The amount of outgoing information is the sum of the adjacent partitions different to $p$ that each vertex residing inside partition $p$ has. receive volume: The amount of incoming information is the number of vertices of partitions different to $p$ adjacent to at least one vertex of partition $p$. diameter: The longest shortest path between two vertices of the same partition. infinity, if the partition is not connected. outgoing migration: Number of vertices that have to be migrated to a different partition. incoming migration: Number of vertices that have to be migrated from a different partition.

For the listed metrics we consider three different norms. Given the values $x_1, \ldots, x_p$, these are defined as follows:

- $l_1(X) := |x_1| + \cdots + |x_p|$, $l_2(X) := (x_1^2 + \cdots + x_p^2)^{\frac{1}{2}}$, $l_\infty(X) := \max(|x_1|, \ldots, |x_p|)$. The $l_1$-norm (summation norm) is a global norm. The global edge cut belongs into this category (it equals half the external edges in this norm). In contrast to the $l_1$-norm, the $l_\infty$-norm (maximum norm) is a local norm only considering the worst value. This norm is favorable if synchronized processes are involved. The $l_2$-norm (Euclidean
norm) lays in between the \( l_1 \) and the \( l_\infty \)-norm and reflects the global situation as well as local peaks.

To provide a better impression of our evaluations, figure 2 shows some frames of the 'slowtric' benchmark. The mesh is generated according to three circles rotating around the center of the squared simulation area. Since the circle sizes differ but a similar attractor is placed in each of them, the area around the smaller circles is refined more deeply. Furthermore, the overall number of triangles increases slightly over time.

The solutions of Metis are shown in the middle left column. While the initial solution looks acceptable, there seem to be some problems in later balancing steps. A closer look to e.g. frame 50 reveals that one partition is degenerated into three parts, one of them only consisting out of a few vertices. Metis applies a local exchange heuristic that usually takes care of a few isolated vertices and assigns them to adjacent partitions. However, other frames show even single isolated triangles, and we guess that this problem is related to the parallelization of the exchange procedure. Furthermore, partitions sometimes are thin or contain long extensions into their neighbors, and both increases the boundary length and communication volumes as well as deteriorates the partition shapes. The distributions calculated with Jostle (middle right) are usually of a better shape, though partitions are occasionally disconnected. When looking at the partition movement, it is interesting too see that domains in deeply refined areas (e.g. around the smallest circle) try to follow these locations. This property is even more distinct in the solutions obtained with Party/DB (right). Furthermore, the latter library computes straighter boundaries and also guarantees connected domains. The diffusion based mechanisms in Party/DB contain a high degree of natural parallelism (one processor per vertex), but they perform very slowly if this parallelism is not exploited. Hence, this library is currently not applicable in practice.

The recorded metrics are displayed in figure 3. The left column contains the data according to the \( l_1 \)-norm while on the right side the \( l_\infty \)-norm has been applied. The \( l_2 \)-norm is omitted due to space limitations. The first row contains the balance, displayed as the maximal size of a partition. It reveals the imbalance setting of ParMetis to 3%, although sometimes up to 6% percent are reached. The parallel version of Jostle is invoked with the same imbalance allowance, but ignores this setting. Party/DB does not provide an imbalance parameter, but usually achieves partition sizes with less than 5% imbalance with some exceptions in other benchmarks.

The next four rows contain the edge-cut, the number of boundary vertices, the communication volume (send and receive volumes are added) and the partition diameter. One can see that the values of the solutions computed by Metis are higher than those for the partitionings obtained with Jostle or Party/DB. This general observation holds for all of our benchmark sets. Furthermore, the three metrics edge-cut, the number of boundary vertices, the communication volume seen to be related, which explains why cut minimizing heuristics are a good approximation to lower real communications costs. Looking at the diameter, we can see again that for some frames Metis computes disconnected domains. In other benchmarks, we can observe this even more frequently. The same holds for Jostle, though degenerated partitions occur less often than in case of Metis.

The last row displays the migration (outgoing and incoming elements are added). Here, Metis seems to follow a different strategy than the two other libraries, moving either very little or a huge amount of data while the migration volume of Jostle and Party/DB is more constant over the frames.

IV. CONCLUSION AND FUTURE WORK

In this paper we have shown that a load balancing benchmark can be build with relatively simple means. Just a graph sequence with the interleaving transition data is necessary.

The proposed benchmark scheme offers the opportunity to provide instances without releasing too much information to the public, what otherwise is often a problem. We now hope that more instances from different sources will be created and published. This would allow developers of load balancing libraries to test and improve their work more efficiently as well as provide an easier way to compare the results as it is currently possible.

Concerning the comparison of the three libraries Metis, Jostle and Party/DB, we conclude that Metis is very quick, but the quality of the solutions delivered by Jostle and Party/DB is much better in all considered metrics. Since the run-time of the Party/DB prototype is indiscussable high, it is currently not applicable in practice. However, if its high degree of parallelism can be exploited, it might become a considerable alternative to balance parallel FEM simulations.

REFERENCES

Fig. 2. Frames 0, 49, 50, 51 and 100 of the 'slowtric' benchmark. The left column shows the underlaying mesh while the other three columns display partitions into twelve domains computed with Metis (middle left), Jostle (middle right) and Party/DB (right), respectively.
Fig. 3. Results of all 101 frames of the ‘slowtric’ benchmark for Metis (blue triangles), Jostle (red squares) and Party/DB (green pentagons).


