Semester Thesis

Mesh Partitioning with Quadtrees and ParMETIS

Roman Hellmüller

February 1, 2012

Supervision:

Achim Gsell (PSI)
Dr. Andreas Adelmann (PSI)

Responsible professor:

Prof. Dr. Peter Arbenz (ETHZ)

ABSTRACT

The aim of this semester thesis is to investigate a prepartitioning algorithm using quadtrees to partition large meshes with ParMETIS. A geometrical prepartitioner using quadtrees is compared to a direct parallel multilevel graph partitioning scheme. This study shows that prepartitioning can remove several bottlenecks in handling meshes.

Contents

1. Introduction

In the past few years high performance computers (HPC) reached petascale peak performance. Due to the power wall [1] development of processors changed direction. Instead of building processors with higher frequencies the trend leads to multiple cores per processor, where each core is slower but consumes less power. This HPC trend leads to more processors with less memory per core. Increasing parallelism due to memory requirements has influence on simulations running on such systems.
This results in increasing importance of scalability of simulations. An upcoming problem is reading and distributing data. It takes an increasing amount of time and decreases the efficiency of a simulation. Due to scattered data the problem generally increases with a larger amount of cores.
The problem of reading and distributing meshes as well as data can be introduced by taking a Finite Element code as an example. Due to growing hardware resources it is possible to run computations with finer meshes or larger problems. However meshes and data need to be split into more parts. In general optimal alignment is not possible for an unspecified or changing number of processors. Due to more parts, meshes and data are more scattered. This makes the problem even harder to solve. Before computations start mesh and data have to be loaded and distributed to all processors.
In earlier approaches one file per processor was used to avoid dealing with scattered meshes and data. Nowadays with tens of thousands of processors this is not applicable anymore, since it is impossible to handle so many files.
A possible approach is to fix the partition beforehand and align mesh and data accordingly. However this leads to severe restrictions concerning portability and usability. Therefore variable partition sizes are desirable and considered as a requirement.
Similarly one could store different partitions without a specific alignment of meshes and data. This would require to randomly read tiny parts of a huge file in parallel according to a precomputed partition. Such a scattered read decreases speed considerably and creates a large bottleneck.
Another approach is to read mesh and data sequentially with a subset of processors. Afterwards calculating the partition and distribute them over the network causing a lot of communication. However calculating a k-way partition of a mesh is in general an exponentially scaling problem [2]. Therefore time for partitioning grows rapidly with increasing size of meshes.

Today's state of the art code for mesh partitioning is ParMETIS [3]. ParMETIS offers several heuristic approaches for creating partitions. Graphs can be partitioned with geometrical methods (i.e. recursive coordinate bisection [4]), spectral methods (i.e. spectral bisection [5]) or with multilevel approaches. Multilevel approaches coarsen the graph in several steps until it is simple and affordable to partition. During coarsening the partition is refined on every level till a partition of the original mesh is available. However partitioning with more then a few hundred cores using multilevel schemes cause a bottleneck that needs to be addressed [6]. This work will be about another approach to face the partitioning problem of meshes for large parallel computation.
In this work we create chunks out of meshes and data. Thereby grouping elements together and creating a structure with larger parts. Those chunks fit several purposes. Firstly it is more efficient to read chunks of sizes in the order of file system blocks or above instead of single elements. Secondly the number of pieces to partition is reduced considerably, because only chunks need to be partitioned and not single elements.
Due to limited scale of this work the following restrictions where made. First in order to decrease complexity of implementation only two dimensional triangle meshes are considered. For this work creating chunks is achieved by using a quadtree. The reduction to two dimensions decreases the amount of work considerably since only quadtrees instead of octrees need to be implemented. The restrictions to triangular meshes does not influence results since no mesh shape specific knowledge is integrated. Second the alignment of meshes according to chunks is to extensive to be part of this work.
A quadtree does geometrically divide areas to obtain a number of regions called leafs. Each leaf contains a contiguous part of the mesh. All leafs together contain all elements i.e. the mesh. The leafs are partitioned and distributed among all processors and hence already creating a reasonable partitioning. Depending on the amount of elements per leaf, the size of the partitioning problem can be reduced considerably. It is possible to reduce the size of the problem by a factor of several orders of magnitude for this exponentially scaling partitioning problem. To further decrease needed communication, a load balancing step is necessary. Since most elements will not be relocated, it is an efficient method to get a good distribution.
Additionally a speed gain can be achieved by storing mesh and data in chunks according to the leafs. Mesh and data can be read in in chunks which decreases I/O bottlenecks and already produces a reasonable initial distribution of mesh elements and data. This storage would still be independent of the number of partitions during runtime. Because partitioning of the quadtree takes place after specifying available number of cores. Further optimization can be achieved with aligning chunks in the file such that they can be read continuously. However the implementation of this block structure is beyond the range of this work.

Another advantage is that generating a quadtree does not necessarily add additional work. Lots of simulations need to locate elements that contain certain points. For this task a quadtree is often used. In such a case the time for creating a quadtree is saved since it is already available. For this reason a quadtree approach is chosen instead of recursive coordinate bisection [4].
In the following chapter the quadtree approach is described in detail, the obtained results are analyzed and compared to state of the art methods. Finally a conclusion and an outlook is presented. In the appendix explanations to basic concepts such as meshes, graphs, trees and partitioning in general can be found.

2. Methods

In this chapter specific information to all necessary techniques, algorithms and libraries is given. Followed by a description of our quadtree approach for prepartitoning meshes.

2.1. Meshes

For our studies we use triangular meshes. However support for quadrilateral cells could be added easily. The restriction to two dimensional meshes results merely from avoiding increased implementation complexity of an octree compared to a quadtree.

2.2. Partitioning

For the partitioning of the meshes we decided to use the state of the art software ParMETIS. It provides geometrical and multilevel graph partitioning algorithms. Parallel multilevel partitioning schemes show a significantly better performance concerning time and quality then geometrical approaches [6]. Therefore those are chosen as reference solution. However serial approaches perform better they are mostly infeasible due to memory requirements.

2.3. Prepartitioning with quadtrees

The goal of this work is to give a proof of concept for using a coarse partitioning with quadtrees with an additional load balancing step instead of parallel multilevel partitioning for meshes.
Our approach is designed the following way. First the mesh is read in and every element of the mesh gets represented as a point in space. The points are stored in a quadtree afterwards, and used for coarse partitioning. Then the dual graph (see appendix B) of the created quadtree and all points per leaf are stored. For every mesh the generation of the quadtree needs to be done only once and can be reused afterwards.

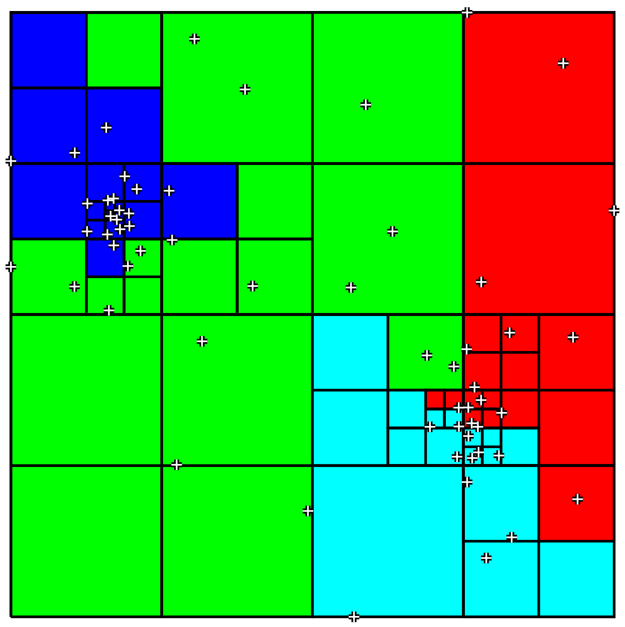

Figure 2.1.: A quadtree built from a small triangle mesh. + represent midpoints of elements. There are maximally two elements in a leaf. Additionally color indicates a possible partition of the quadtree into four parts.

At the beginning of the simulation the dual graph of the quadtree is read in. The partitioning of the dual graph is calculated as visualized in Fig 2.1. We assume that the work for every element is equal. Therefore the weights per leaf/vertex are set according to the number of elements per leaf. Each processor reads its share of leafs. If data is aligned according to the leafs, reading scattered data is avoided. Reading leafs as partitioned leads to a contiguous initial distribution of the mesh. For a minimal edge cut and optimal distribution an additional load balancing step is required. We will investigate run time and quality of the load balancing step compared to a full parallel partitioning.

3. Results

All the results where obtained on the Felsim cluster at PSI. It consists of 16 nodes with each dual-socket quad core Intel Xeon E5450 3.0 GHz CPUs and 16 GB Ram. Furthermore the network uses an InfiniBand interconnect.

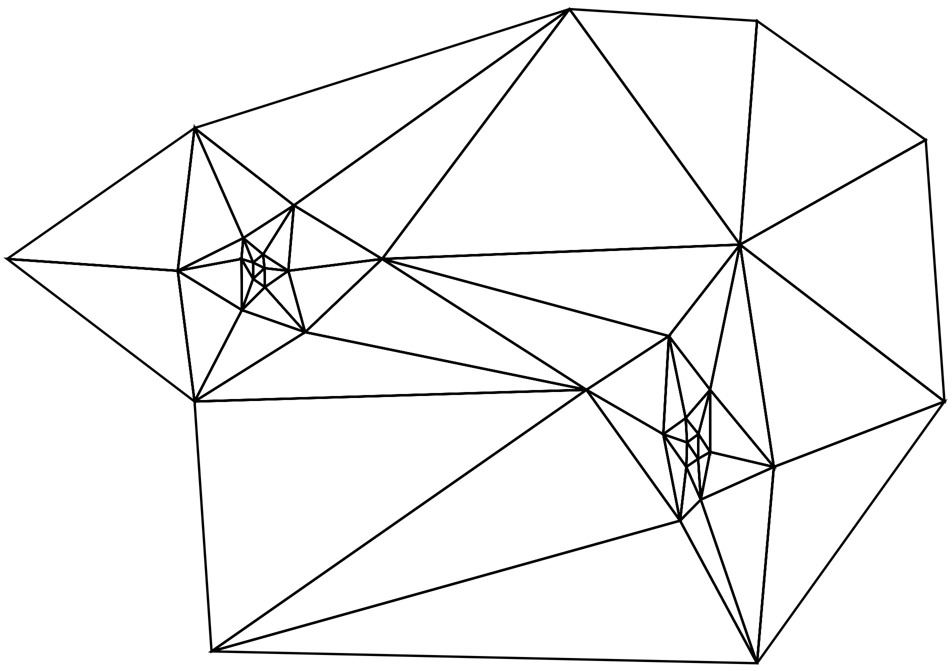

Figure 3.1.: Basic unstructured triangle mesh consisting of 61 elements.

For obtaining the following results, meshes are used that are refined from the basis mesh in Fig 3.1. With a uniform refinement that creates four new elements out of each element by splitting edges in half and connecting the resulting midpoints, finer meshes are generated. The exact number of elements per mesh can be found in Tab 3.1 below. For generating the quadtree a variable amount of maximal number of elements per leaf was used to have a nearly constant amount of leafs for all meshes.

mesh name	number of elements
mesh 1	999’424
mesh 2	3’997’696
mesh 3	15’990’784

Table 3.1.: Size of meshes

3.1. Edge cut

First we compare the edge cut of our approach with existing solutions of Metis [7] and ParMETIS. This analysis allows us to compare the quality of the partitioning since the edge cut is closely related to the amount of interprocessor communication needed during simulation.

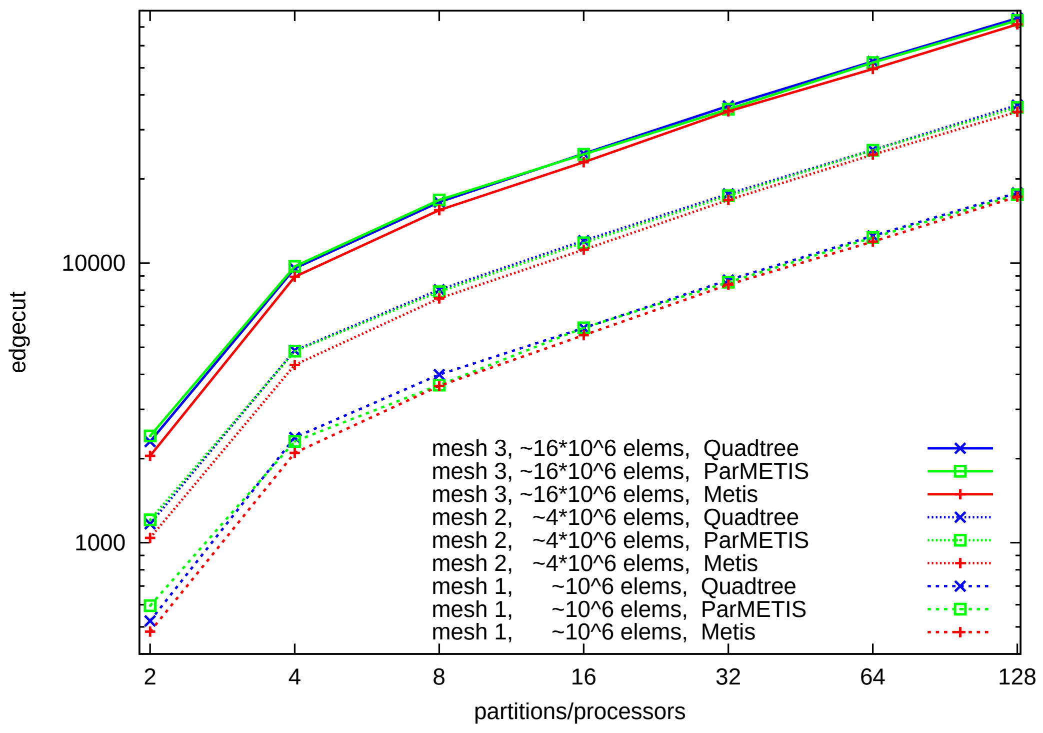

Figure 3.2.: Edge cut for distribution to different number of partitions. Compared are the quality of the partitioning with the serial Metis, our quadtree based approach and ParMETIS. For both parallel partitioning algorithms the number of partitions also corresponds to number of used processors.

Figure 3.2 shows that Metis produces the best results concerning edge cut, which corresponds to the amount of interprocessor communication. Partitioning with our quadtree based approach shows the same exponential behavior as Metis and ParMETIS. This shows that our approach generates reasonable solutions. However the quadtree method as well as multilevel partitioning of ParMETIS do not reach the quality of Metis. More importantly the differences stays equal with increased number of partitions. So relative differences are decreasing which leads to increasingly better results with growing number of partition for the parallel schemes.
For a clearer comparison between quadtree and ParMETIS based partitioning, same results can be displayed in a relative plot.

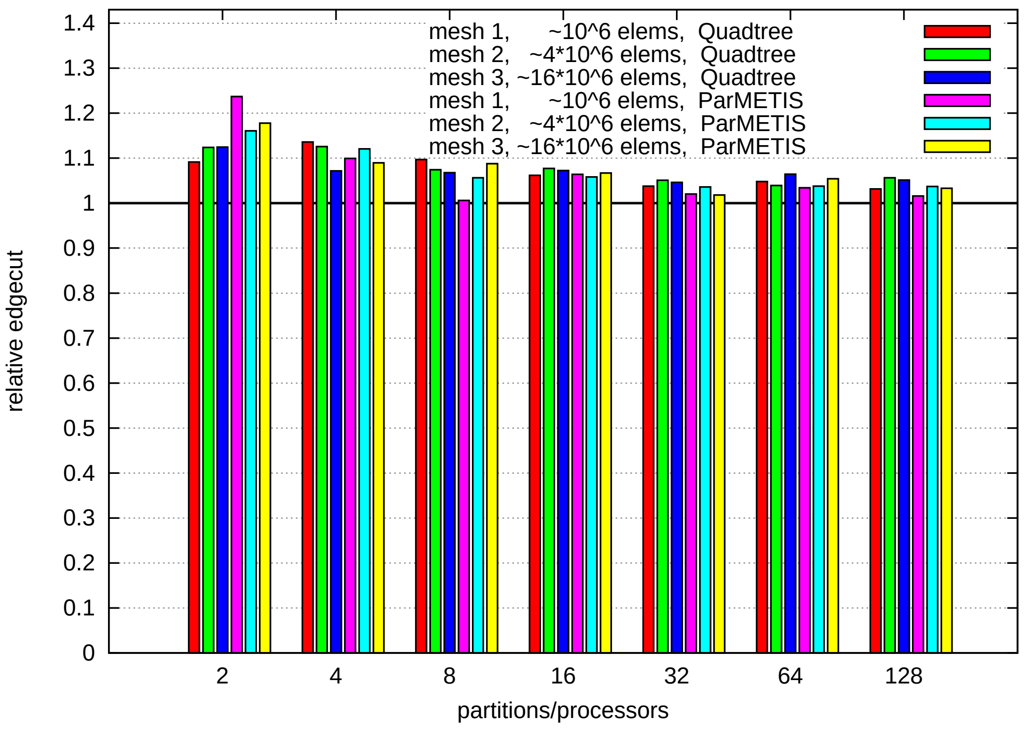

Figure 3.3.: Comparison of calculated edge cut relatively to Metis. A value below one would indicate a smaller edge cut than calculated with Metis.

Firstly one can see that no solution reaches an edge cut lower than Metis. However the tendency that with a larger number of partitions relative edge cuts gets smaller is visible. This corresponds to the already observed constant difference in Fig. 3.2. Secondly it is possible to compare ParMETIS with the quadtree based approach. ParMETIS seems to perform up to ten percent better than our quadtree based approach concerning edge cut. However the difference between Metis and ParMETIS is still more dominant. Further tests with smaller meshes have shown, that the quadtree approach as well as ParMETIS can give better results than Metis. This effect was already seen in previous studies [6]

3.2. Timing

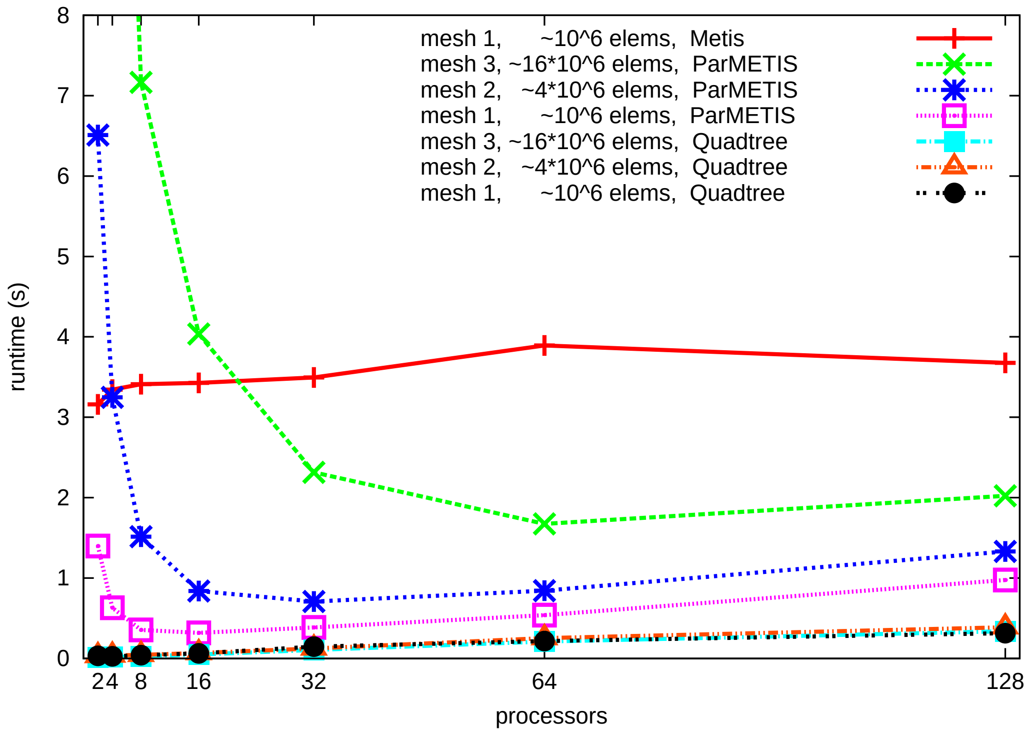

Figure 3.4.: Timing on different number of processors. The partitioning with ParMETIS is compared to the load balancing step with the quadtree approach. To allow a comparison also the timing for mesh 1 with Metis is included.

The ParMETIS timing results shows for a small number of processors a rapid decrease in time as expected from scaling effects. With increasing number of processors scaling effects disappear and runtime starts to increase slightly again. Concerning efficiency it is obvious that more processors may not be beneficial because time increases while adding more computing resources. This behavior is expected since communication overhead increases and finally dominates runtime. On the other hand with large meshes corresponding to bigger memory requirements increasing number of cores is the only option to be able to run simulations. Our results show that multilevel partitioning should may be done on less processors then actual computing. It is clear from Fig. 3.2 that mesh size influences the optimal amount of processors for partitioning considerably and can therefore only be determined per problem.

Second, one can see clearly that load balancing, required by the quadtree approach, uses much less time. Nevertheless the amount of time rises with an increasing number of processors. However for all the different mesh sizes the timing is equal. This suggests that it is mostly influenced by overhead from global communication and does not reflect increasing amount of work due to larger meshes. Therefore the optimal amount of processors for our approach depends mainly on memory requirement.

4. Conclusions and outlook

The aim of this work was to investigate a potential benefit of a coarse grained prepartitioning of large meshes compared to a direct multilevel partitioning using ParMETIS [6]. As a coarse grained prepartitioner a quadtree algorithm was introduced.
In Ch. 3 we show that it is possible to maintain the quality of partitions and keeping interprocessor communication as low as possible with our quadtree approach. Additionally we could show that concerning runtime it is extremely beneficial to use prepartitioning. Time requirement for load balancing also proofed to be independent of the mesh size. This is especially remarkable due to the fact that the amount of leafs where nearly constant. A constant amount of leafs leads to larger leafs with finer meshes. Prepartitioning with larger leafs is expected to produce a worse prepartitioning. However this did not effect the edge cut after the load balance.
Consequently recommendation is clearly towards using a coarse grained prepartitioner. Before any implementation into the H5hut library [8] large scale tests could clarify how large the possible gain in performance would be. For the implementation the quadtree approach needs to be extended to octrees in 3D. No complications are expected from this extension. Furthermore changing file layout such that data is aligned according to leafs would probably create a huge gain in I/O time. Scattered I/O could be avoided. However further studies are necessary to asses potential advantages.

Appendix

A. Meshes

A mesh consists of individual cells that have a specific shape. Also mixed shapes of cells in a mesh are possible and can provide advantages for certain applications.

B. Graph/ dual graph

A graph is used to represent objects that are connected to each other. Every object is represented by a vertex and connections by edges. A dual graph is a related construct. However every area enclosed by the graph is represented by a vertex in the dual graph and neighboring areas/vertices are connected with edges as seen in Fig 4.1

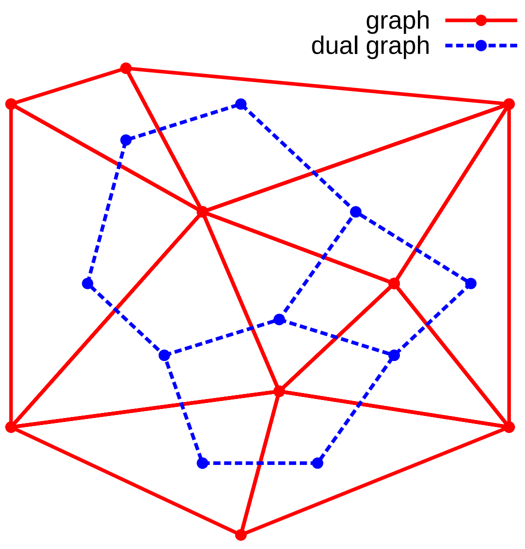

Figure 4.1.: A dual graph can be constructed out of a graph. From our point of view the graph corresponds to a mesh.

Graphs were used since quite a long time and are intensively studied in discrete mathematics. Also the problem of graph partitioning is well known and solution techniques have been developed. For this reason nowadays mesh partitioning is mostly based on graphs. Probably also due to the simplicity to obtain a graph from a mesh. Normally the dual graph of a mesh is used for partitioning since every element of the mesh corresponds to a vertex in the graph. Therefore a partition of the vertices of a graph can easily be mapped to corresponding elements in the mesh.

C. Quadtree/ octree

A quadtree [9] is a data structure where a node can be split into four quadrants called children. Normally there is a two dimensional region (square, rectangle) associated with each node. Each children itself is split into quadrants if the number of data elements per node excite a given maximum. A node is considered to be a leaf if it is not split. All leafs together span the whole area as the initial node and contain all data elements. One application of quadtrees is to use them as an alternative representation for pictures instead of the widespread pixel approach. However it can also be used for geometrical partitioning of space. Additionally it is worth mentioning that this concept can be extended to more dimensions. Instead of quadrants, octants are considered in a three dimensional case leading to octrees. Apart form the additional space dimension there is no conceptual difference.

D. Partitioning

Partitioning is the process of distributing a mesh to a given number of processors. A lot of partitioning schemes are available today. The simplest class are geometrical partitioners. Depending on the techniques it is split into two or several parts. This step is repeated as long as necessary. Another class involves multiple levels of meshes. As long as the mesh is too large it will be represented by a coarser level. The most coarse level is then split and during the uncoarsening phase the distribution is refined.
The quality of a partitioning algorithm is determined by three main factors. Firstly the boundaries between parts should be minimal. As partitioning is most widely used on graphs it is responsible for the term edge cut. The edge cut is the quantity of edges that have to be cut such that new unconnected graphs evolve out of the initial graph. This quantity is important since it corresponds to the amount of communication needed during computations. As communication can be a bottleneck for simulations and also decreases scalability, it is clear that a low edge cut is preferred.
Secondly an optimal load balance should be maintained. The distribution of work according to available resources is considered a good load balance. Since efficiency is important in high performance computing (HPC), it is clear that a balanced simulation is preferred. Furthermore, as soon as imbalance costs excite those for rebalancing is state of the art to do a load balancing.

Lastly available resources should be used as effectively as possible. On the other hand the amount of resources available sets limits on feasibility. As an example memory requirement mostly dictate that partitioning needs to be done in parallel. Also time becomes an issue if a serial approach would not finish within reasonable time.

E. H5hut

The H5Hut [8] project provides a high-performance I/O library for particle-based simulations. The goal of H5hut is to hide the complexity of parallel I/O without influencing performance. Additionally the library uses a single file approach, to alleviate post processing. Even though a 'one file per processor' approach is performing better during runtime it can lead to severe problems during post processing. Therefore a one file approach is chosen. For best possible compatibility the HDF5 [10] standard is used, so that visualization tools as well as other HDF5-based interfaces and tools can be used. The current development release allows to read meshes in parallel. Part of this thesis is to analyze potential benefits from using a prepartitioning approach instead of direct partitioning, and making a recommendation concerning implementing prepartitioning schemes.

F. Content of CD-ROM

A digital copy of this thesis, the source code and some additionals are put on CD-ROM which is put below.

Bibliography

[1] V. Agarwal, M.S. Hrishikesh, S.W. Keckler, and D. Burger. Clock rate versus ipc: the end of the road for conventional microarchitectures. In *Computer Architecture, 2000. Proceedings of the 27th International Symposium on*, pages 248 –259, june 2000.

[2] M. R. Garey and D. S. Johnson. *Computers and Intractability: A Guide to the Theory of NP-Completeness.* W. H. Freeman, San Francisco, CA, 1979.

[3] George Karypis and Vipin Kumar. ParMETIS: Parallel Unstructured Graph Partitioning and Sparse Matrix Ordering System, Version 4.0. `http://www.cs.umn.edu/~metis/parmetis`, 2011.

[4] Roy Williams. Performance of dynamic load balancing algorithms for unstructured mesh calculations. *Concurrency*, 3:457–481, 1991.

[5] Roy D. Williams. Performance of dynamic load balancing algorithms for unstructured mesh calculations. *Concurrency: Practice and Experience*, 3(5): 457–481, 1991. ISSN 1096-9128. doi: 10.1002/cpe.4330030502. URL `http://dx.doi.org/10.1002/cpe.4330030502`.

[6] George Karypis and Vipin Kumar. Parallel multilevel k-way partitioning scheme for irregular graphs, 1996.

[7] George Karypis and Vipin Kumar. MeTis: Unstructured Graph Partitioning and Sparse Matrix Ordering System, Version 5.0. `http://www.cs.umn.edu/~metis`, 2011.

[8] M. Howison, A. Adelmann, E.W. Bethel, A. Gsell, B. Oswald, and Prabhat. H5hut: A high-performance i/o library for particle-based simulations. In *Cluster Computing Workshops and Posters (CLUSTER WORKSHOPS), 2010 IEEE International Conference on*, pages 1 –8, sept. 2010. doi: 10.1109/CLUSTERWKSP.2010.5613098.

[9] R. A. Finkel and J. L. Bentley. Quad trees a data structure for retrieval on composite keys. *Acta Informatica*, 4:1–9, 1974. ISSN 0001-5903. URL `http://dx.doi.org/10.1007/BF00288933`. 10.1007/BF00288933.

[10] The HDF Group. Hierarchical data format version 5. `http://www.hdfgroup.org/HDF5`, 2000-2012.

Statement of authorship

I hereby certify that this semester thesis has been composed by myself, and describes my own work, unless otherwise acknowledged in the text. All references and verbatim extracts have been quoted, and all sources of information have been specifically acknowledged.

Signature: .... Date: ....