High Performance Multigrid on Current Large Scale Parallel Computers

Tobias Gradl, Ulrich Rüde

Lehrstuhl für Systemsimulation
Friedrich-Alexander-Universität Erlangen-Nürnberg
Cauerstr. 6
D-91058 Erlangen
tobias.gradl@informatik.uni-erlangen.de
ulrich.ruede@informatik.uni-erlangen.de

Abstract: Making multigrid algorithms run efficiently on large parallel computers is a challenge. Without clever data structures the communication overhead will lead to an unacceptable performance drop when using thousands of processors. We show that with a good implementation it is possible to solve a linear system with 10^{11} unknowns in about 1.5 minutes on almost 10,000 processors. The data structures also allow for efficient adaptive mesh refinement, opening a wide range of applications to our solver.

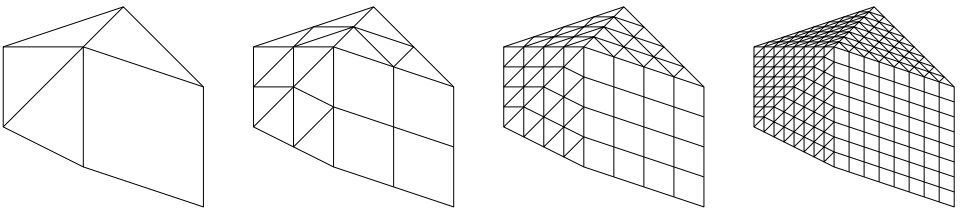
1 Introduction

In the most recent TOP-500 list, published in November 2007, HLRB II at the Leibniz Computing Center of the Bavarian Academy of Sciences is ranked at position 15 for solving a linear system with 1.58 million unknowns at a rate of 56.5 Teraflops in the Linpack benchmark. However, this impressive result is of little direct value for scientific applications. There are few real life problems that could profit from the solution of a general dense system of equations of such a size. The test problem reported in this article is a finite element discretization on tetrahedral 3D finite elements for a linear, scalar, elliptic partial differential equation (PDE) in 3D, as it could be used as a building block in numerous more advanced applications. We have selected this problem, since it has a wide range of applications, and also, because it is an excellent test example for any high performance computer architecture. Our tests on HLRB II show that this computer is well suited and yields high performance also for this type of application.

HLRB II, an SGI-Altix system, went into operation in September 2006 with 4096 processors and an aggregate main memory of 17.5 Terabytes ("phase 1"). In April 2007, the system was upgraded to 9728 cores and 39 Terabytes of main memory ("phase 2"). In particular in terms of available main memory, it is currently one of the largest computers in the world. Though HLRB II is a general purpose supercomputer, it is especially well suited for finite element problems, since it has a large main memory and a high bandwidth.

With our article we would like to demonstrate the extraordinary power of today's comput-

Figure 1: Regular refinement example for a two-dimensional input grid. Beginning with the input grid on the left, each successive level of refinement creates a new grid that has a larger number of interior points with structured couplings.



ers for solving finite element problems, but also which algorithmic choices and implementation techniques are necessary to exploit these systems to their full potential.

2 Hierarchical Hybrid Grids

In this article we focus on multigrid algorithms [BHM00, TOS01], since these provide mathematically the most efficient solvers for systems originating from elliptic PDEs. Since multigrid algorithms rely on using a hierarchy of coarser grids, clever data structures must be used and the parallel implementation must be designed carefully so that the communication overhead remains minimal. This is not easy, but our results below will demonstrate excellent performance on solving linear systems with up to 3×10^{11} unknowns and for up to almost 10,000 processors.

HHG ("Hierarchical Hybrid Grids") [BGHR06, BHR05] is a framework for the multigrid solution for finite element (FE) problems. FE methods are often preferred for solving elliptic PDEs, since they permit flexible, unstructured meshes. Among the multigrid methods, algebraic multigrid [Mei06] also supports unstructured grids automatically. Geometric multigrid, in contrast, relies on a given hierarchy of nested grids. On the other hand, geometric multigrid achieves a significantly higher performance in terms of unknowns computed per second than algebraic multigrid.

HHG is designed to close this gap between FE flexibility and geometric multigrid performance by using a compromise between structured and unstructured grids: a coarse input FE mesh is organized into the grid primitives vertices, edges, faces, and volumes that are then refined in a structured way, as indicated in fig 1. This approach preserves the flexibility of unstructured meshes, while the regular internal structure allows for an efficient implementation on current computer architectures, especially on parallel computers.

The grid decomposition into the primitives allows each group of primitive to be treated separately during the discretization and solver phases of the simulation, so that the structure of the grid can be exploited. For example, instead of explicitly assembling a global stiffness matrix for the finite element discretization element by element, we can define it implicitly using stencils. If the material parameters are constant within an element, the stencil for each element primitive is constant for all unknowns interior to it for a given

level of refinement. Then, of course, only one stencil has to be stored in memory for each level of that element, which is the main reason for HHG's memory efficiency and high execution speed.

3 Parallelization

To exploit high end computers, the programs must be parallelized using message passing. For an overview of parallel multigrid algorithms see [HKMR06] The HHG framework is an ideal starting point for this, since the mesh partitioning can be essentially accomplished on the level of the coarse input grid, that is, with a grid size that can still be handled efficiently by standard mesh partitioning software like Metis[1]. In order to avoid excessive latency, the algorithmic details and the communication must be designed carefully. The multigrid solver uses a Gauß-Seidel smoother that traverses the grid points in the order of the primitives of the coarse input mesh: first, all vertices are smoothed, then all edges, and so on. During the update of any such group, no parallel communication is performed. Instead, data needed in the same iteration by neighbors of higher dimension is sent after the update of a group in one large message per communication partner; data needed by neighbors of lower dimension in the next iteration can even be gathered from all groups and sent altogether at the end of the iteration (see fig 2).

This procedure minimizes the number of messages that have to be sent, and thus greatly reduces communication latency. At the same time, it guarantees an important prerequisite of the Gauß-Seidel algorithm: because primitives within a group are never connected to each other directly, but only to primitives of other groups, all neighbors' most recent values are already available when a grid point is updated. For example, faces are only connected to other faces via vertices, edges or volumes, no communication is necessary during the smoothing of the faces. This strategy only goes wrong near the corners of triangles, where edges directly depend on each other (see fig 3). Here the values from the previous iteration are used, giving the smoother Jacobi characteristics at the affected points. Numerically, this leads to only a slight deterioration of the convergence rates, but the gain in execution time more than outweighs this effect.

4 World record in linear system solving

In our largest computation to date, we have used 9170 cores of HLRB II and HHG to solve a finite element problem with 307 billion unknowns in 93 seconds run time. We believe that this is the largest finite element system that has been solved to date. Additionally, we point out that the absolute times to solution are still fast enough to leave room for using this solver as a building block in e. g. a time stepping scheme.

The results in Table 1 show the results of a scaling experiment from 4 to 9170 compute

[1] `http://glaros.dtc.umn.edu/gkhome/views/metis`

Figure 2: HHG grouping of communication

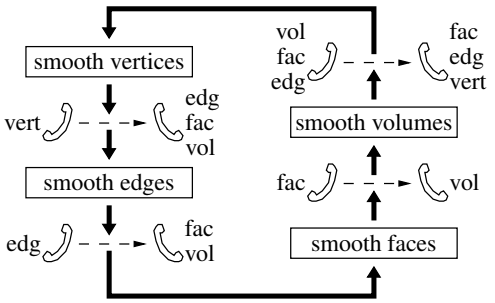
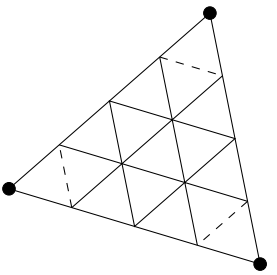


Figure 3: HHG communication and ignored dependencies



cores. The amount of memory per core is kept constant and the problem size is chosen to fill as much of the available memory as possible, which is commonly referred to as *weak scaling experiment*. If the program were perfectly scalable, the time per V cycle would stay constant throughout the table, because the ratio of problem size (i. e. workload) versus number of cores (i. e. compute power) stays constant. Near perfect scaling is seen as measure of the quality of an algorithm and its implementation. For HLRB II in installation phase 2, the computation time increases by only 20% when scaling from 4 to 9170 cores. This is still not perfect but in our view acceptable, especially when compared to other algorithms and especially in terms of the absolute compute time. Note that perfect scalability is the more difficult to achieve the faster a code is. The shorter the time spent in actual calculations is, the larger is the fraction of the time spent in communication, and the more pronounced are the communication overheads introduced by the scaling.

Phase 1 of HLRB II used single-core processors, providing every core with its own memory and network interface. The dual-core configuration of phase 2 provides less bandwidth per core, since two cores must now share an interface. Additionally, a part of the installation is now configured as so-called "high density partitions" where two dual-core processors share one interface, which means there is even less bandwidth available per core. Benchmark results including these high density partitions are marked with an asterisk in table 1. HHG is highly sensitive to the available memory bandwidth. The timings for 64, 504, and 2040 cores show that the dual-core processors of phase 2 account for approximately 39% deterioration in runtime compared to phase 1; compare this to the 20% of efficiency lost through scaling over the whole computer. The same effect is observed when switching between the regular and the high density partitions of phase 2. While one V cycle takes only 6.33 s on 6120 cores of the regular partitions, on the high density partitions the runtime is already 7.06 s on just 128 cores but then increases only slightly further to 7.75 s for our largest runs.

Table 1: Scaleup results for HHG. With a convergence rate of 0.3, 12 V cycles are necessary to reduce the starting residual by a factor of 10^{-6}. The entries marked with * correspond to runs on (or including) high density partitions with reduced memory bandwidth per core.

# Processors	# Unknowns	Time per V cycle (s)		Time to solution (s)	
		Phase 1	Phase 2	Phase 1	Phase 2
4	134.2	3.16	6.38 *	37.9	76.6 *
8	268.4	3.27	6.67 *	39.3	80.0 *
16	536.9	3.35	6.75 *	40.3	81.0 *
32	1 073.7	3.38	6.80 *	40.6	81.6 *
64	2 147.5	3.53	4.93	42.3	59.2
128	4 295.0	3.60	7.06 *	43.2	84.7 *
252	8 455.7	3.87	7.39 *	46.4	88.7 *
504	16 911.4	3.96	5.44	47.6	65.3
2040	68 451.0	4.92	5.60	59.0	67.2
3825	128 345.7	6.90		82.8	
4080	136 902.1		5.68		68.2
6120	205 353.1		6.33		76.0
8152	273 535.7		7.43 *		89.2 *
9170	307 694.1		7.75 *		93.0 *

5 Parallel Adaptive Grid Refinement

The paradigm of splitting the grid into its primitives (vertices, edges, faces, and volumes) and the technique of regular refinement also prove valuable when implementing adaptivity. Our remarks on this topic divide the refinement methods into two groups, those which create conforming grids, and those which do not. For an exact definition of the term *conforming grid* see [Rüd93b]; in short it means that all grid nodes lie only at the ends of edges and at the boundaries of faces. Techniques like red-green refinement [BSW83] create conforming grids and are widely used, because they are numerically unproblematic. The other group of techniques creates non-conforming grids with hanging nodes which are often considered numerically unpleasant. Yet the implementation of such a technique is especially straightforward in HHG, that is why we show how to treat the hanging nodes correctly so they do not pose a problem.

For more details we refer to an introductory article about multigrid on adaptively refined grids, with many links to related work, by Bastian and Wieners [BW06]. A paper by Lang and Wittum describes the building blocks of a parallel adaptive multigrid solver in detail [LW05].

Figure 4: Grid originating from two triangles, the lower one refined once, the upper one refined twice. The hanging nodes are encircled.

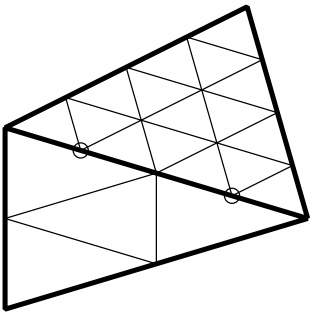
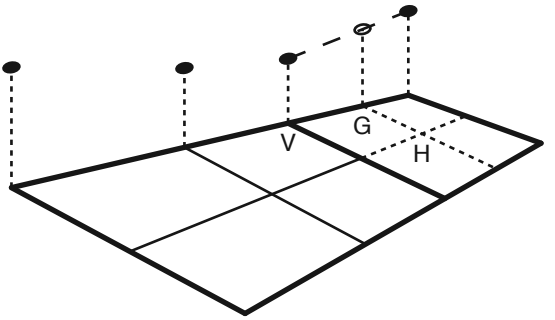


Figure 5: Two quadrilaterals, the left one with one level of refinement. For evaluating the stencil at vertex V, the values at the grid points G and H have to be interpolated.



5.1 Refinement with Hanging Nodes

In HHG, adaptive refinement can simply be achieved by allowing the coarse grid elements to be refined to different levels, with the effect of having a non-conforming grid with hanging nodes at the interfaces between elements. Figure 4 shows a non-conforming grid consisting of two triangles refined to different levels.

In HHG, a grid primitive is always refined to the finest level of all adjacent primitives of higher dimension. This ensures that the primitives of highest dimension (faces in 2D, volumes in 3D), comprising the largest part of the unknowns, are all surrounded by primitives with at least the same refinement level. Thus, they do not need any special care and can be treated without performance penalties.

An interface primitive sitting between two primitives with different levels of refinement (vertex V in fig 5) sets up its stencils just as if all adjacent primitives were refined to the finest level, with the effect that some grid points referred to by the stencils do not exist (points G and H in fig 5). These points are interpolated from points on the finest available level on their primitives. The interpolation rules—crucial for the numerical stability of the algorithm—can be derived easily by interpreting the problem from the viewpoint of *hierarchical bases* (see e. g. [Rüd93a]), with the hierarchical surplus defined to be zero on the non-existent levels.

5.2 Red-Green Refinement

The two basic rules used in red-green refinement are shown in fig 6 and explained in detail in [BSW83]. The *red* rule is identical to the one we use in our regular refinements (cf. fig 1): a triangle, for example, is refined into four new triangles by connecting its edge midpoints with new edges.. The resulting hanging nodes are taken care of by applying

Figure 6: The middle triangle is red-refined and induces green refinement in the other elements.

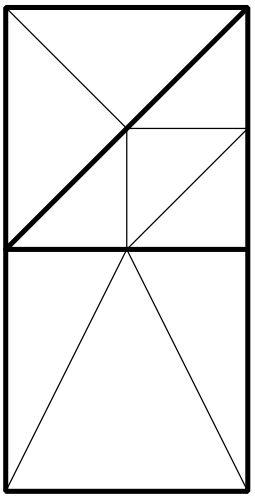
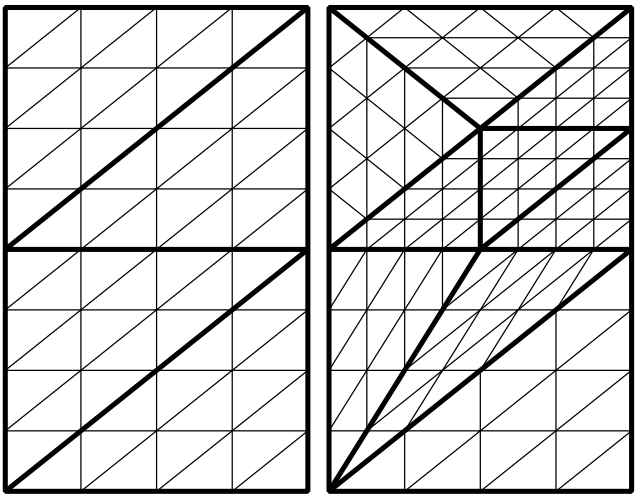


Figure 7: Red-green refinement of an already regularly refined mesh. (a) The initial mesh consists of four triangles, each refined regularly two times. (b) Red refinement in the upper right triangle induces green refinement in the upper and lower left triangles.



the *green* rule to each affected neighboring element: the element is split into two or more new elements by connecting the hanging node with one or more of the element's already existing corner nodes.

Red-green refinement can also be applied to already regularly refined elements as they occur in HHG, which is illustrated in fig 7. The refinement can—and should—be applied at the level of the coarse input grid. Then, thanks to the small number of elements in the input grid, its application is very cheap, and all the tools developed for these methods can be used. Furthermore, the regular internal grid structure of the input elements, responsible for HHG's high performance, is not harmed.

The upper right triangle in fig 7 with three interior points is split into four new triangles with three interior points each, doubling the grid resolution in this region. The structured interiors after refinement can be initialized in a natural and efficient way from the structured interiors before refinement. Some of the new grid points have the same location as old grid points, values at these points are simply inherited from the old grid. The values at the grid points in between are obtained by linear interpolation. The same applies to the neighboring triangles that have to be green-refined.

5.3 Combining Both Approaches

While each of the methods can alone be used to implement adaptive refinement, we pose that combining both results in additional advantages. Being able to do red-green refinement as well as structured refinement with varying levels, we can trade the advantages and disadvantages of both methods to obtain an optimal compromise between adaptivity and performance.

If there are not many geometrical features that have to be resolved in the domain, the initial HHG mesh can be very coarse. If it turns out during the solution process that the mesh has to be refined in some area, one or more of the very large coarse grid elements have to be refined regularly, leading to a fine mesh resolution also in areas where it is not needed. The solution to this problem is to red-green-refine the initial coarse mesh.

A disadvantage of red-green refinement is that it does not necessarily preserve the element type. A green refinement step turns a quadrilateral element into triangles (cf. fig 6). So, if purely quadrilateral/hexahedral meshes are desired, red-green refinement cannot be used.

One of the goals when setting up a simulation for HHG is to have as few coarse grid elements as possible, because then the structured areas are large and can be treated with high performance. Refinement with hanging nodes creates less coarse grid elements than red-green refinement and should thus be used whenever possible.

6 Conclusions and Outlook

The HHG framework and HLRB II have been used to solve a finite element problem of world-record size. HHG draws its power from using a multigrid solver that is especially designed and carefully optimized for current, massively parallel high performance architectures. The SGI Altix architecture is found to be well-suited for large scale iterative FE solvers. While the parallel scalability is already good, there is still room for improvement which we will exploit by further optimizing the communication patterns. The future will also bring comparative studies on other architectures, for example the IBM BlueGene. Adaptive refinement will enable us to conquer a wider range of applications than before.

Acknowledgments

We would like to point out that it was Benjamin K. Bergen who brought the idea of HHG to life within his Ph.D. thesis [Ber06]. The initial phase of the project was funded by the KONWIHR supercomputing research consortium[2]. The ongoing research on HHG is funded by the international doctorate program “Identification, Optimization and Control with Applications in Modern Technologies” within the Elite Network of Bavaria[3].

[2] `http://konwihr.in.tum.de/`

[3] `http://www2.am.uni-erlangen.de/elitenetzwerk-optimierung`

References

[Ber06] B. Bergen. *Hierarchical Hybrid Grids: Data Structures and Core Algorithms for Efficient Finite Element Simulations on Supercomputers*, volume 14 of *Advances in Simulation*. SCS Europe, July 2006.

[BGHR06] B. Bergen, T. Gradl, F. Hülsemann, and U. Rüde. A Massively Parallel Multigrid Method for Finite Elements. *Computing in Science & Engineering*, 8:56–62, November 2006.

[BHM00] W.L. Briggs, V.E. Henson, and S.F. McCormick. *A Multigrid Tutorial*. SIAM, 2. edition, 2000.

[BHR05] B. Bergen, F. Hülsemann, and U. Rüde. Is 1.7×10^{10} Unknowns the Largest Finite Element System that Can Be Solved Today? In *SC '05: Proceedings of the 2005 ACM/IEEE conference on Supercomputing*, page 5. IEEE Computer Society, 2005.

[BSW83] R.E. Bank, A.H. Sherman, and A. Weiser. Some refinement algorithms and data structures for regular local mesh refinement. In R. Stepleman et al., editors, *Scientific Computing, Applications of Mathematics and Computing to the Physical Sciences, Volume I*. IMACS, North-Holland, 1983.

[BW06] P. Bastian and C. Wieners. Multigrid Methods on Adaptively Refined Grids. *Computing in Science & Engineering*, 8:44–54, November 2006.

[HKMR06] F. Hülsemann, M. Kowarschik, M. Mohr, and U. Rüde. Parallel geometric Multigrid. In A.M. Bruaset and A. Tveito, editors, *Numerical Solution of Partial Differential Equations on Parallel Computers*, volume 51 of *Lecture Notes in Computational Science and Engineering*, pages 165–208. Springer, 2006.

[LW05] S. Lang and G. Wittum. Large-scale density-driven flow simulations using parallel unstructured Grid adaptation and local multigrid methods. *Concurrency and Computation: Practice and Experience*, 17:1415–1440, September 2005.

[Mei06] U. Meier Yang. Parallel Algebraic Multigrid Methods — High Performance Preconditioners. In A.M. Bruaset and A. Tveito, editors, *Numerical Solution of Partial Differential Equations on Parallel Computers*, volume 51 of *Lecture Notes in Computational Science and Engineering*, pages 209–236. Springer, 2006.

[Rüd93a] U. Rüde. Fully Adaptive Multigrid Methods. *SIAM Journal on Numerical Analysis*, 30(1):230–248, February 1993.

[Rüd93b] U. Rüde. *Mathematical and Computational Techniques for Multilevel Adaptive Methods*, volume 13 of *Frontiers in Applied Mathematics*. SIAM, 1993.

[TOS01] U. Trottenberg, C. Oosterlee, and A. Schüller. *Multigrid*. Academic Press, 2001.