FRIEDRICH-ALEXANDER-UNIVERSITÄT ERLANGEN-NÜRNBERG
INSTITUT FÜR INFORMATIK (MATHEMATISCHE MASCHINEN UND DATENVERARBEITUNG)

Lehrstuhl für Informatik 10 (Systemsimulation)

Is 1.7×10^{10} Unknowns the Largest Finite Element System that Can be Solved Today?

B. Bergen, F. Hülsemann and and U. Rüde

Lehrstuhlbericht 05-3

Abstract

The hierarchical hybrid Grids (HHG) framework attempts to remove limitations on the size of problem that can be solved using a finite element discretization of a partial differential equation (PDE) by using a process of regular refinement, of an unstructured input grid, to generate a nested hierarchy of *patch-wise structured* grids that is suitable for use with geometric multigrid. The regularity of the resulting grids may be exploited in such a way that it is no longer necessary to explicitly assemble the global discretization matrix. In particular, given an appropriate input grid, the discretization matrix may be defined implicitly using stencils that are constant for each structured patch. This drastically reduces the amount of memory required for the discretization, thus allowing for a much larger problem to be solved.
Here we present a brief description of the HHG framework, detailing the principles that led to solving a finite element system with 1.7×10^{10} unknowns, on an SGI Altix supercomputer, using 1024 nodes, with an overall performance of 0.96 TFLOP/s, on a logically unstructured grid, using geometric multigrid as a solver.

1 Introduction

Due to its flexibility, and the ease with which it can be applied to unstructured grids, the finite element method (FEM) is commonly used for simulating partial differential equations (PDE's) on complex domains. In such cases, solving the discrete problem is equivalent to solving a linear system of n equations in n unknowns. A variety of solvers may be used to solve the resulting system. However, as the problem size grows, the complexity of the solver algorithm becomes a limiting factor. For example, on the fastest systems available today, operating at theoretical peak, solving a system with 10^{10} unknowns using an algorithm with $\mathcal{O}(n^3)$ complexity would take on the order of 100 million years[1].

Fortunately, for discretizations of elliptic PDE's, multigrid may be used as a solver. Multigrid algorithms combine the smoothing properties of inexpensive iterative methods with corrections computed on coarse representations of the computational grid to achieve asymptotically optimal convergence, i.e., computing the solution requires on the order of n operations, where n is the number of unknowns. Algebraic multigrid is a suitable solver for elliptic operators that are the result of the finite element method being applied to an unstructured grid. Therefore, algorithmic complexity is not the limiting factor on the size of a problem that may be solved in such cases. However, since these problems are discretized on unstructured grids, the couplings of the entries in the operator matrix are also unstructured and must be stored using some type of indirect indexing. A common data structure for storing these entries is the compressed row storage (CRS) format [2], which stores only the non-zero entries of the matrix. Although this type of storage scheme is efficient, in the sense that it does not store unnecessary values, the amount of memory needed to store the matrix entries becomes the limiting factor on the size of problem that can be solved.

The inherent indirection of such storage schemes is also an issue with respect to performance. This is primarily due to the fact that the indirect addressing of the matrix entries precludes certain aggressive compiler optimizations that can reduce latency. The ordering of the unknowns is also generally such that the cache subsystem, common to many modern architectures, can not be used effectively to reduce the memory bandwidth limitations of the solver algorithm.

For discretizations of problems defined on structured grids with constant coefficients, these limitations do not come into play. In such cases, explicit assembly of the discretization matrix is not necessary, so that the memory requirements of the discretization are insignificant. It is also possible to use geometric multigrid as a solver, which has less pre-processing overhead than algebraic multigrid, and can be implemented using stencil-based operations. These can often be implemented to be extremely efficient. The drawback of such techniques is that they have difficulty capturing the important features of complex domains. So, while these techniques are efficient and do not have the same memory limitations as unstructured grid representations, they fail to address the original goal of simulating complex systems.

[1] To solve a problem with 10^{10} unknowns using an $\mathcal{O}(n^3)$ algorithm we would need to perform on the order of 10^{30} operations. Assuming that we could perform 10^{14} operations per second (100 TFLOP/s), we would need approximately $317,097,919$ years to compute the solution. Using an $\mathcal{O}(n^2)$ algorithm would still take approximately 11 days, which is a long time to wait.

The goal of the HHG framework is to mix the flexibility of the finite element method on unstructured grids with the efficiency and optimal complexity of geometric multigrid to enable efficient simulation of complex geometries on modern day supercomputers. The motivation for this approach comes from the observation that it is common practice to apply regular refinement to unstructured input grids in order to resolve certain fine scale features of the domain that are not sufficiently captured by the finite element partitioning. Adding several levels of refinement, generates a grid hierarchy, of logically unstructured grids, that is suitable for use with geometric multigrid, and has structured regions that may be discretized using stencils. An optimal situation for this approach is the case where the input grid has patch-wise constant material parameters. In such an instance, the discretization stencil for each patch is constant coefficient, regardless of the level of refinement, with the result that the memory required for the discretization is greatly reduced when compared to a discretization of the same grid using CRS data structures.

The HHG framework is part of the Gridlib project, a KONWIHR[2] funded project to develop tools for large-scale simulation and visualization[11, 3, 10]. The Gridlib project is a joint undertaking between several chairs at the Universität Erlangen–Nürnberg including: Informatik 10 (simulation – LSIM), Informatik 9 (computer graphics – LGDV) and the Chair for Fluid-Mechanics (LSTM).

In the following sections, we present a brief description of the HHG grid framework and the component algorithms needed to implement a geometric multigrid solver on the HHG data structures. Next, we discuss some of the limitations of this approach and describe ways in which certain restrictions could be eased to allow the framework to be applied to a larger class of problems. We then present performance results for an implementation of the HHG data structures for homogeneous, tetrahedral input grids that is able to solve Poisson type problems with 10^{10} unknowns. We conclude with some remarks about possible future work and the challenges that must be overcome to solve even larger problems.

2 Elements of Multigrid

Multigrid uses a recursive combination of local error reduction, or *smoothing*, and global coarse grid correction to achieve asymptotically optimal complexity on elliptic problems. This method is motivated by two observations. The first is that standard iterative methods are effective at reducing high-frequency, or oscillatory error while leaving low-frequency error essentially unchanged. Such iterative solvers are generally referred to as smoothers because their application produces geometrically smooth error. The second observation is that the resulting smooth error may be accurately represented on a coarser grid. Restriction of the residual error to a coarse grid creates an error equation that, in the context of the coarse grid, again has high-frequency error components. Smoothing is effective at reducing these error components.

These observations lead to a process of smoothing and coarse grid correction that is applied recursively on each subsequent coarse grid until a level is reached where the resulting error equation is small enough to solve directly. The grid hierarchy is then recursed upward, adding the corresponding correction at each level until the finest level is reached. It is common to apply smoothing operations during this upward traversal as well, as the process of coarse grid correction can excite some of the high-frequency modes. Because of this, multigrid algorithms are often referred to by cycle type and the number of pre and post-smoothing operations that are applied. For example, one cycle of a multigrid algorithm that recurses sequentially through a grid hierarchy applying two smoothing iterations at each level on the way down, and two smoothing iterations at each level on the way back up, is generally called a $V(2,2)$ cycle. The V denotes the type of cycle, which in this case looks like the letter V. For a more in-depth coverage of multigrid see [5, 17].

In order to implement a geometric multigrid solver, one needs the following component algorithms: A smoother algorithm, an algorithm to compute the residual error, an algorithm to transfer the error from a fine grid to a coarse grid, an algorithm to transfer the correction from a coarse grid to a fine grid, and a coarse grid solver. The algorithms used to transfer quantities between the various grid representations are generally referred to as transfer operators. The transfer operation from coarser grids to finer grids is usually called *interpolation*, while the transfer operation from finer grids to coarser grids is usually called *restriction*. The term *geometric* in "geometric

[2]Kompetenznetzwerk für Technisch-Wissenschaftliches Hoch- und Höchstleistungsrechnen in Bayern

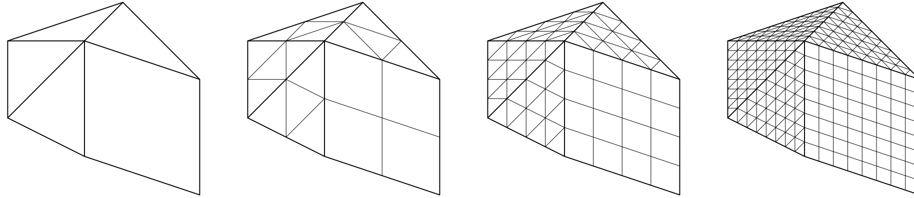

Figure 1: Regular refinement example for a two-dimensional input grid. Beginning with the input grid on the left, each successive level of refinement creates a new grid that has a larger number of interior points with structured couplings.

multigrid" refers to the case where the geometry of the problem domain may be used in defining the transfer operators.

In the next section, we introduce the HHG grid framework, and discuss the principles needed to adapt the geometric multigrid component algorithms so that they may be applied to hierarchies of logically unstructured grids.

3 Hierarchical Hybrid Grids

As stated in the introduction, the basic idea behind HHG is to take grid hierarchies that have been generated by regular refinement and decompose them into structured regions. Treated individually, each region is a structured grid in its own right. Because of the regularity of the original hierarchy, transfer operations between grids are defined geometrically allowing a straightforward implementation of geometric multigrid. This is essentially a variant of a block-structured approach with the advantage that the setup and resolution of the block interfaces is handled automatically. In this section, we outline the principles of the HHG data structures.

3.1 Regular Refinement and Grid Decomposition

We begin with a purely unstructured input grid that has been created by meshing a simply-connected, open, bounded domain Ω. This grid is assumed to be fairly coarse, in the sense that it resolves only the large scale structure of the domain. For example, the input grid should resolve the geometry and the material parameters of the domain, but not necessarily all of the fine scale features. It is also assumed that the desired resolution of the solution is much higher than that of the input grid, so that some type of refinement is necessary to resolve fine scale features not handled by the grid partitioning.

To this grid, we add regular refinement by inserting a new vertex at the midpoint of each edge and then adding new edges and faces in such a way that each element in the input grid is divided into eight sub-elements of equal volume[3]. This step is applied successively to each new grid, until the desired level of resolution is obtained, generating a nested hierarchy of grids that is suitable for use with geometric multigrid algorithms. Each grid in the hierarchy is still *logically unstructured*, but has *patch-wise structured* regions. Figure 1 shows an example of regular refinement as applied to a two-dimensional input grid. We are currently only concerned with three-dimensional input grids, however this example is easier to visualize. We use the notation Ω^l for $l = 0, 1, \ldots, n_R - 1$, to denote the lth grid in the hierarchy, where Ω^0 denotes the input grid, and n_R is the total number of refinement levels including the input grid.

Next, we perform a grid decomposition on the entire hierarchy with respect to the input grid. This decomposition divides the the hierarchy into its basic parts or *grid primitives*. We restrict the class of admissible input grids to those that may be decomposed into the following primitive types:

- *Elements* - open, bounded, polygonal *volumes* in the form of tetrahedra, paralleloids, or paralleloid prisms.
- *Faces* - open, bounded, polygonal *surfaces* in the form of triangles, or parallelograms.

[3]This assertion implies certain restrictions on the types of input grids under consideration.

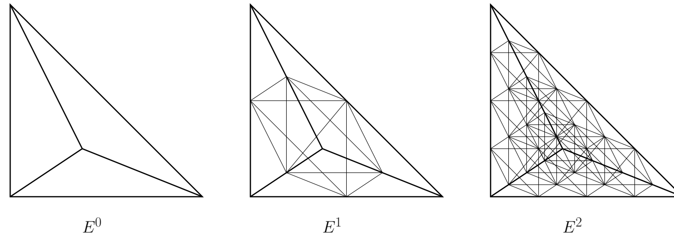

Figure 2: Regular refinement of a tetrahedral input element. From left to right, the input element E^0 is refined once to obtain E^1 and again to obtain E^2. Notice that E^2 is the first level of refinement where the element has an interior unknown.

- *Edges* - open, bounded, linear *line segments*.
- *Vertices* - single *points*.

Because the original domain is simply-connected, each grid primitive will have at least one other grid primitive that is adjacent to it.

As a consequence of the grid decomposition, we now have two distinct ways in which to view a particular grid level that has been generated by regular refinement. Each level may still be viewed as a composite entity that has been refined as a whole in the context of the nested hierarchy, i.e., $\Omega^0 \subset \Omega^1 \subset \cdots \subset \Omega^{n_R - 1}$. However, we may also view each grid level as the union of the *input grid primitives* with refinement applied on a per-primitive basis. As an example of a refined element primitive, consider Figure 2, which shows the first two levels of regular refinement as applied to a tetrahedral input element. The refinement rule used for tetrahedra is due to [4].

The grid decomposition is an important aspect of the HHG framework because it allows us to treat each grid primitive separately. For example, due to the grid decomposition and regular refinement, at a sufficient level of refinement[4], each element primitive has a block-structured interior with a constant material parameter, for input grids with patch-wise constant material parameters. Using the appropriate data structures, introduced in the next section, we can exploit these properties. In particular, instead of explicitly assembling a global stiffness matrix for the finite element discretization, we can define it implicitly using stencils, such that, for a given level of refinement, the stencil for each element primitive is *constant for all unknowns that are interior to it.* The same approach may be used for the face primitives, and to some extent, for the edges. Vertex primitives are inherently unstructured, since they incorporate the unstructured nature of the input grid, and therefore must still make use of indirect indexing to store their coefficient weights.

The fact that the global stiffness matrix can be defined using stencils is perhaps the single most important attribute of HHG with respect to performing extremely large simulations, because it *shifts the bulk of the memory usage needed to solve a problem from the discretization to the unknowns.* As a specific example, consider a typical tetrahedral input grid with 50,000 input elements, 100,000 input faces, 50,000 input edges, and 9000 vertices. Applying seven levels of refinement generates a fine grid with approximately 1.7×10^{10} unknowns. Assuming that each unknown has fifteen neighbors (this will at least be true for the interiors of the elements and faces, which, at seven levels of refinement, account for 99.9% of the total unknowns), we would require approximately 3.7 Terabytes of RAM to store the global stiffness matrix using a standard CRS format, compared to approximately 30 Megabytes of RAM for HHG.

3.2 HHG and Multigrid

As an implementational consideration, in order to treat the refined grid primitives independently, each primitive must be allocated as a separate memory block that includes an additional memory layer or *halo* to hold dependency information from adjacent primitives. During the course of solving the discrete problem, these dependencies must be updated. For example, this is necessary during a Gauß–Seidel smoothing iteration to get the current approximations of neighboring unknowns that

[4]Tetrahedral elements must be regularly refined twice before they have interior points.

are not local to the primitive being updated. This implies that some *local communication* is necessary between adjacent primitives, and requires that some changes be made to the implementation of the component algorithms needed to perform geometric multigrid.

To illustrate these changes, consider Algorithm 1. Instead of looping over each unknown in the computational grid, as would be usual if we were to treat the refined grid as a global unstructured grid, we loop over each primitive of a particular type, performing a component algorithm and updating the dependencies as needed. This is a decomposition of the algorithm that is analogous to the grid decomposition. The resulting algorithm performs the same set of operations as if the

Algorithm 1 Algorithm Decomposition for HHG

1: **for** each vertex **do**
2: apply operation to vertex
3: **end for**
4: **for** each edge **do**
5: copy from vertex *interiors*
6: apply operation to edge
7: copy to vertex *halos*
8: **end for**
9: **for** each face **do**
10: copy from vertex and edge *interiors*
11: apply operation to face
12: copy to vertex and edge *halos*
13: **end for**
14: **for** each element **do**
15: copy from vertex, edge, and face *interiors*
16: apply operation to element
17: copy to vertex, edge, and face *halos*
18: **end for**

grid were treated globally, albeit, with a different ordering of the unknowns. For the structured regions of the refined grid, the multigrid component algorithms are the same as those used for a purely structured grid. The unstructured regions are treated in a similar manner to completely unstructured grids, in the sense that they must use indirect indexing to access neighboring points. However, these regions also benefit from the refinement because the geometry of the problem can be used to define their transfer operations. Using these principles, we can define the multigrid component algorithms for each type of primitive in the grid decomposition.

To complete our development of the HHG multigrid algorithm, we still need a coarse grid solver. Since the original input grid is purely unstructured, one possibility is to use algebraic multigrid. It is also feasible to use a standard direct solver such as LU decomposition. However, using algebraic multigrid offers a more efficient solver algorithm and reinforces the idea behind an important property of the HHG data structures: The combination of exploiting structured refinement where possible while still benefiting from the flexibility of an unstructured grid partitioning.

3.3 HHG Implementation and Multigrid Convergence

Our current implementation of the HHG data structures is in the form of a C++ library. C++ was chosen for the flexibility it offers in designing more complex data structures. However, this flexibility often incurs a penalty in performance. This is due to certain language features, e.g., *aliasing*, that make it difficult or impossible for the compiler to perform code optimizations that exploit instruction-level parallelism. Therefore, where possible, the computational kernels have been implemented in Fortran 77. The mixture of C++ and Fortran 77 is analogous to the mixture of unstructured and structured grids in the original HHG concept, so that the current implementation reflects the same principles of flexibility and efficiency found there.

Grid management and partitioning are handled by the Unstructured Grid Geometry Library (UGLi). UGLi is an internally developed C++ library that has interfaces to the METIS[12, 14] and ParMETIS[13] partitioning libraries. Currently, we employ an MPI only communication model even on architectures like the SGI® Altix™3700 that have *hybrid* memory characteristics. Adding OpenMP style parallelism is an interesting possibility for future work. However, it is unclear

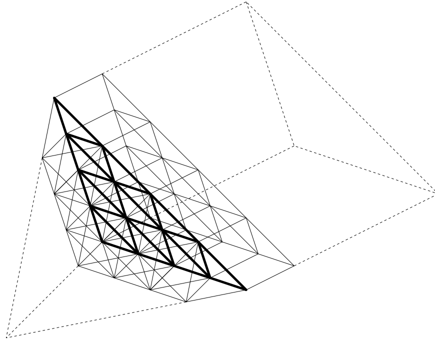

Figure 3: Refined triangular face with geometric couplings. The interior structure of the face does not depend on the type of elements adjacent to it.

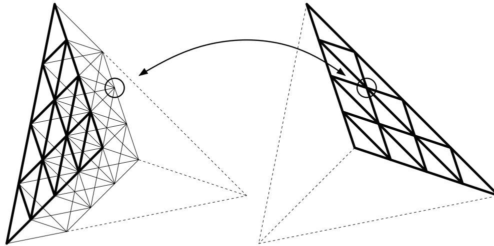

Figure 4: Example of an ignored dependency between two triangular faces.

whether or not we could realize a performance gain by doing so [15, 16]. The current implementation handles only homogeneous tetrahedral input grids. However, the basic data structures have been implemented for hexahedral and prismatic elements so that heterogeneous input grids will be handled in the future.

The current HHG implementation offers two choices for the coarse grid solver. These are: The BoomerAMG algebraic multigrid solver [9], through an interface to the Hypre library [8], and LU decomposition, through an interface to the SuperLU library[6, 7].

In order to simplify the overall implementation, and to avoid excessive latency in parallel, we impose a *one-way communication model* for updating the local dependencies. By "one-way", we mean that the copying of dependency information only occurs between grid primitives of different dimensions, e.g. edges do not communicate directly with other edges. Also, the copying is always handled by the primitive of higher dimension. The second restriction allows us to create a reasonable design for handling heterogeneous input grids. Consider the refined face primitive in Figure 3 that is shown with its geometric couplings. Regardless of what type of elements adjoin at a face, the face will always have the same interior structure (denoted by bold outline). In addition, each element type knows what type of face it has on each of its exterior surfaces. Therefore, an element knows the internal structure of its faces. This does not hold the other way around. As shown in Figure 3, the structure of the geometric couplings of the face depends on the interior structure of the elements adjacent to it. Since we are dealing with unstructured input grids, there is no way to know beforehand what type of elements will adjoin at a particular face. In principle, we could implement a separate data structure for each possible combination of elements adjoining at each type of face. However, this would complicate our implementation and add to the amount of code that must be maintained. Instead, we let the primitives of higher dimension handle copying to and from their adjacencies.

As a consequence of the first restriction of one-way communication, i.e., that there is no dependency information traded between primitives of the same dimension, there are certain dependencies that are partially ignored during computation. As an example, consider Figure 4, which shows a partial representation of the geometric coupling between two triangular faces. Recall that in our modified update in Algorithm 1, the faces are updated sequentially. Suppose that the face on the

	$V(2,2)$		$V(3,3)$	
unknowns	Deps	No Deps	Deps	No Deps
125,055	0.0923	0.0993	0.0676	0.0742
1,024,255	0.0909	0.0937	0.0667	0.0689
8,290,815	0.0867	0.0878	0.0641	0.0647

Table 1: Average serial convergence rates for both serial implementations. This table shows that certain dependencies, which are ignored during smoothing, do not adversely effect multigrid convergence. The rates shown are the average to the convergence rates of the first six V-cycles.

right has been updated during an iteration of a Gauß–Seidel smoothing algorithm. Unless the faces communicate directly with each other, the face on the left will have a *stale* value of the circled point in its halo when it performs its update. Because we ignore such dependencies, the resulting smoothing algorithm has some *Jacobi-like* points, i.e., points where the most recent approximation of those points is not used in updating other points.

To find out what affect the ignored dependencies have on multigrid convergence, two versions of the serial code were implemented: One that honors these dependencies, and one that does not. Table 1 shows convergence rates for each case run with a $V(2,2)$ cycle and a $V(3,3)$ cycle on a Poisson problem. All convergence rates shown are averages over the first six V-cycles. These tests indicate that the ignored dependencies do not have a significant effect on serial convergence. Parallel results for the same problem show that the ignored dependencies do not cause a problem there either. On average, each $V(3,3)$ cycle in parallel reduces the residual error by a factor of approximately 15 for a convergence rate of 0.067 (again averaged over the first six V-cycles).

3.4 Limitations

The current implementation of HHG has several limitations. Some of these are due to restrictions that we placed on the types of input grids that are admissible, while others have to do with the decisions that we made in order to make implementing the data structures more feasible. There are certain limitations that are inherent to HHG and cannot be removed. We now discuss some of the different limitations and how certain restrictions might be lessened to make the HHG principles applicable to a larger class of problems.

The first limitation we discuss has to do with the restrictions that the input grid have patch-wise constant material parameters and that the elements in the input grid be such that they can be subdivided into child elements of equal volume. Both of these restrictions are necessary to ensure that the resulting grid hierarchy can be discretized using constant-coefficient stencils. As stated previously, this is an important factor in allowing us to solve extremely large problems, because it reduces the amount of memory needed to represent the global discretization matrix. Completely easing these restrictions to include grids with continuously variable coefficients and arbitrary hexahedral and prismatic elements would mean that the HHG data structures would require substantially more memory, although still less than a standard CRS format[5]. A possible compromise would be to allow certain elements in the input grid to have continuously variable coefficients while keeping the rest constant. This way, if there were regions in the domain that did not really require variable coefficients, we could potentially save on the memory needed for the discretization. In the case of time-evolutionary problems, variable coefficients might be used to follow a shock while the rest of the domain stays relatively unchanged. This is a similar idea to adaptive refinement techniques, i.e., only do work were it is really necessary, except, in this case, we only use memory where it is really necessary.

Another limitation is due to a restriction that we have not yet mentioned. This is the restriction that the finite element discretization use only piece-wise linear basis functions. The decision to restrict the type of admissible basis functions has to do with an implementational difficulty concerning the support of higher order basis functions. Namely, because of the grid decomposition, if we were to use higher order basis functions, their support would span several adjacent primitives. This

[5]The HHG data structures would save the memory needed to store the column information for each matrix entry, which for a 64-bit operating system, would require as much memory as for the entries themselves.

refinement level	2	3	4	5	6	7
% structured	69	91	98	99	99*	99*
memory savings (MB)	39	439	3,804	31,041	249,577	1,999,150

Table 2: Ratio of structured to unstructured points with possible memory savings for a typical input grid. The asterisks denote that after six levels of refinement the fine grid is effectively completely structured.

would complicate the implementation of both local and parallel communication between primitives. Although this is not an inherent limitation of the HHG data structures, there is currently no plan to change or ease this restriction.

The final limitation that we discuss involves regular refinement. This is a fundamental principle of the HHG approach and cannot be removed. At issue, is the fact that at least some regular refinement must be applied in order to use the HHG data structures. In practice, it is necessary to add several levels of refinement to obtain the memory and performance benefits of the method. Table 2 shows the ratio of structured to unstructured points and the possible memory savings for a typical input grid with respect to the refinement level. It is clear that the method offers a substantial memory savings after only a few levels of refinement. However, if the problem being solved cannot benefit from the added refinement, or if the input grid already has very high resolution, then the HHG approach may not be appropriate.

This is not to say that the method is overly hindered by this requirement. For example, the HHG approach offers a straightforward method of adding dynamic adaptivity. This can be accomplished by going back to the original input grid and refining a single patch or group of patches to create a modified input grid that has finer resolution of some problem area that has been identified during processing. This introduces some dangling nodes that must be resolved by forming new connections. However, when regular refinement is then applied to the new input grid, the problem areas will have a much higher resolution, and computation can be continued using the original lower resolution approximations to initialize the new degrees of freedom.

4 Performance Results

The goals of our performance tests were to determine: First, what kind of performance could be achieved, second, whether the performance would be scalable both in serial and in parallel, and finally, how large a problem could be solved. Our primary performance results were performed on a cluster of four SGI® Altix™3700 supercluster nodes connected by a NUMAlink™4 fast interconnect located at SGI's factory in Chippewa Valley, Wisconsin. Each supercluster on this machine is made up of 512 1.6 GHz Itanium® 2 processors, each with 3 MB level 3 cache, and 1 GB main memory. In general, the Itanium® 2 is able to perform 4 floating point operations per cycle, so that each processor has a theoretical peak performance of 6.4 GFLOP/s. In total, this machine has a peak theoretical performance of 13.1 TFLOP/s with 2 TB main memory.

A similar SGI® Altix™3700 supercluster was used for tests that did not require the full resources of the cluster in Wisconsin. This is a test machine for the new National Super Computer located at the Leibniz-Rechenzentrum (LRZ) in München Germany with 128 1.6 GHz Itanium® 2 processors, each with 6 MB level 3 cache and 4 GB main memory. The initial installation of this computer will be completed during Q1 2006 and will have a peak theoretical performance of 33 TFLOP/s. It will be upgraded by mid 2007 to nearly 70 TFLOP/s peak.

Several different tools were used to measure performance. These included: SGI® Histx, PAPI, instrumentation in the code, and theoretical operation counts. In particular, we were very careful to verify timings and operation counts using multiple tools. SGI® Histx[6] is a suite of tools designed to complement pfmon, a part of the perfmon project[7]. The *histx* tool (included as part of SGI® Histx) performs statistical counts of various events by evaluating the program pointer at a user definable cycle frequency, e.g., if the cycle frequency is set to $1,000,000$, histx records the position of the program pointer every $1,000,000$ cycles and creates a statistical profile. The lipfpm (also included

[6]Information about the Histx suite can be found at: http://www.sgi.com/products/software/histx.

[7]Information about the perfmon project can be found at: http://www.hpl.hp.com/research/linux/perfmon.

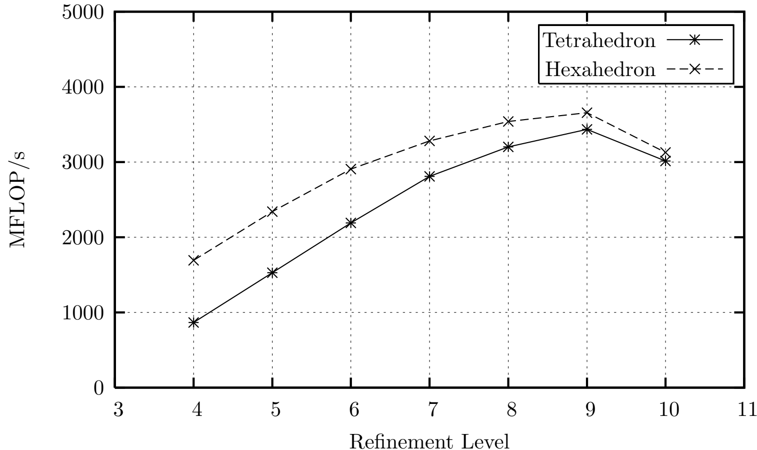

Figure 5: Serial performance of row red-black Gauß–Seidel implementation applied to a single input element on the SGI® Altix™3700 supercluster at LRZ.

as part of SGI® Histx) and PAPI[8] tools read performance registers directly and are generally more accurate. PAPI has the advantage that it can be used to instrument specific code sections so that profiling results can be used to isolate the performance of a particular loop or function call.

4.1 Serial Performance

As stated, our first goal was to determine the raw performance that could be expected from our implementation. Using histx, we determined that, on average, approximately 88% of the execution cycles of the multigrid algorithm are spent performing smoothing operations. Based on this measurement, it seems safe to assume that the performance of our multigrid code depends strongly on the performance of the smoother.

Figure 5 shows the serial performance in MFLOP/s of performing a row red-black Gauß–Seidel[9] smoothing iteration on a single tetrahedral element (solid line). The interior points of each tetrahedral input element are updated by applying a fifteen-point stencil. The figure shows that the overall performance of the smoother depends on the level of refinement. There are several possible reasons for this. The most likely reason is the change in the ratio of loop overhead to increasing inner-most loop length, i.e., the loop overhead should be constant while the inner-most loop length will become longer with each level of refinement. The inner-most loop length of an iterative method is an important factor on the Itanium® 2. Because this processor uses explicit software pipelining, it depends heavily on aggressive scheduling to reach its full capabilities. Longer inner-most loop lengths facilitate the processors scheduling ability, and thus, help performance.

One drawback of the tetrahedral data structure is that, regardless of the level of refinement, the inner-most loop length always decreases as the algorithm moves through the interior unknowns. To try to isolate this effect, we have applied the same fifteen-point stencil to a hexahedral input element (shown in Figure 5 as a dashed line). The hexahedral data structure has the same inner-most loop length throughout its entire structured interior, and therefore, does not suffer from the same problem. The results show that the same stencil applied to the hexahedral memory block does, in fact, achieve better performance. Since the memory layout of the data structure is the only difference between the two results, this shows that the decreasing inner-most loop length, inherent to the tetrahedral data structures, is responsible for some loss of performance at lower refinement levels. However, the tetrahedral data structures still achieve an extremely high percentage of peak performance. For example, at seven levels of refinement the tetrahedral data structure attains 44% of peak.

[8]Information about PAPI can be found at: http://icl.cs.utk.edu/papi.

[9]Row red-black Gauß–Seidel employs a red-black ordering per row of unknowns in a structured grid. This is useful for avoiding pipeline stalls that hurt performance on certain architectures.

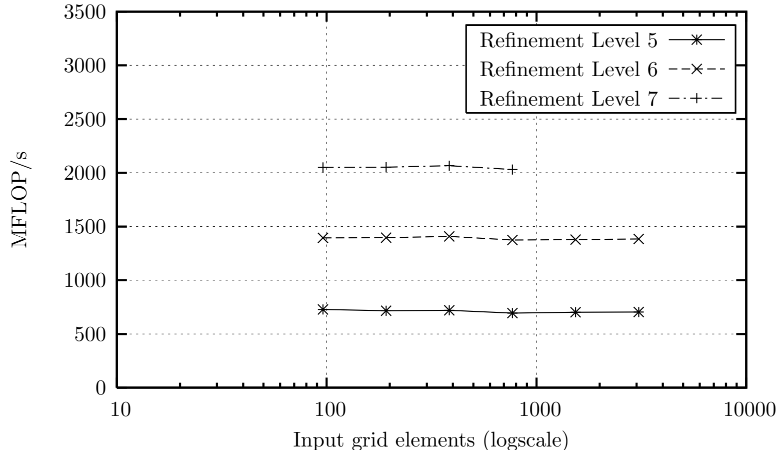

Figure 6: Serial scalability test performed on a single Itanium® 2 processor on the SGI® Altix™3700 supercluster at LRZ.

Again considering Figure 5, we see that at ten levels of refinement the performance suddenly drops. Although not presented here, our results indicate that this is due to the fact that at ten levels of refinement the problem no longer fits into the level 3 cache. In general, our goal is to be able to apply between six and eight levels of refinement to an unstructured input grid, so that ten levels is somewhat excessive. The result is included here only to give a complete overview of the performance characteristics of the smoothing algorithm.

4.2 Serial Scalability

Because of the local communication needed to update the dependencies between the various primitives, it is important to show that the code scales well as the number of elements in the input grid grows, as this will introduce more local copying, and could have an adverse affect on performance. Figure 6 shows smoothing results for the cases where five, six, and seven levels of refinement have been applied to input grids with increasing numbers of elements (shown as a logscale on the horizontal axis of the plot). These results show that the local copying has little or no effect for the levels of refinement shown.

4.3 Parallel Results

Figure 7 shows a parallel scalability study of both our smoothing and V-cycle implementations. These results were generated on the cluster of 4 SGI® Altix™3700 supercluster nodes at SGI's factory in Chippewa Valley, Wisconsin. Each test used in the study was run with seven levels of refinement to solve a Poisson problem. Table 3 shows grid details for each test, including: The number of unknowns, the aggregate performance in GFLOP/s for both the smoothing and V-cycle implementations, and the time to solution. Although this machine has a total of 2048 processors, our largest test case used only 1024 processors. This was due to the limited amount of memory available per processor (1 GB) and the need to find a suitable combination of unstructured input elements and regular refinement that would fit within the memory constraints of the machine.

The parallel results show that the HHG implementation has good *weak scalability* to at least 512 processors, i.e., from 2 to 512 processors, the grid size was doubled each time the number of processors was doubled. The last interval in the plot, both for smoothing and V-cycles, show a *strong scalability* result (denoted by boxed symbols), i.e., the problem size was held constant and the number of processors was doubled so that this result shows *speedup*. The combination of performance in GFLOP/s and time to solution show that the implementation is very efficient. Given a similar machine with 4 GB RAM per CPU, we predict that we would be able to solve a problem with 6.8×10^{10} unknowns at approximately 3.5 TFLOP/s.

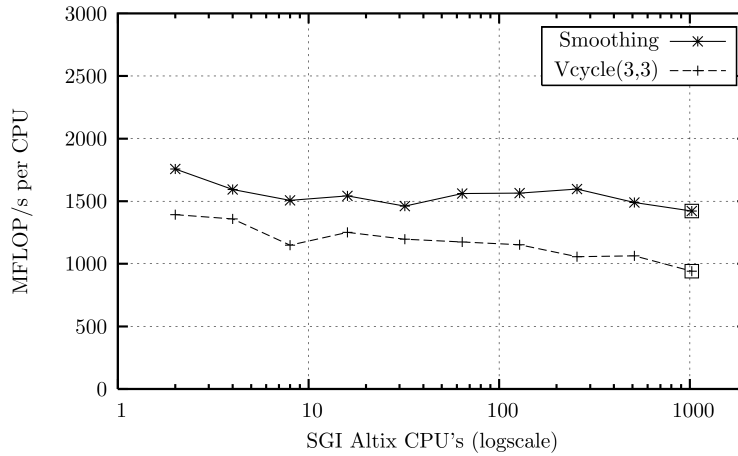

Figure 7: Parallel performance test performed on a cluster of 4 SGI® Altix™3700 supercluster nodes at SGI's factory in Chippewa Valley, Wisconsin. Note that from 512 to 1024 processors, the number of unknowns was held constant so that this interval shows *strong scalability.* From 2 to 512 processors, the grid size was doubled at each level along with the number of processes so that, for these intervals, the results show *weak scalability.* Table 3 shows the corresponding time to solution for each test.

#Procs	#Dofs$\times 10^6$	#Elements$\times 10^6$	#Input Elements	GFLOP/s	Time (seconds)
2	66.72	402.65	192	3.5/2.8	57
4	133.43	805.31	384	6.4/5.4	59
8	267.65	1,648.36	786	12.1/9.2	69
16	535.30	3,221.23	1536	24.7/20.0	64
32	1,070.61	6,442.45	3072	46.8/38.3	67
64	2,144.35	12,884.90	6144	100/75	68
128	4,288.71	25,769.80	12288	200/147	69
256	8,577.41	51,539.61	24576	409/270	76
512	17,167.40	103,079.22	49152	762/545	75
1024	17,167.40	103,079.22	49152	1,456/964	43

Table 3: Details of scalability test from Figure 7. Note that the aggregate performance in GFLOP/s of the smoothing and V-cycle implementations are shown as *smoothing/V-cycle*. The last column on the right shows the time needed to perform 10 V(3,3) cycles. Also note that the problem size does not change for the last two rows.

5 Conclusions

We have shown that the HHG data structures make it possible to solve extremely large problems using finite element discretizations of elliptic PDE's on logically unstructured grids. In particular, we have shown results for solving a Poisson problem with 10^{10} degrees of freedom at one TFLOP/s on 1024 processors. Additionally, we have suggested how certain modifications can be made to the HHG data structures to expand the class of problems that can be handled using this approach. Some possibilities for future work include: Adding full support for hybrid input grids, adding dynamic adaptivity, and adding support for patch-wise continuously variable coefficients. In [1], the authors show results for solving a finite element problem with 10^8 degrees of freedom and claim that this is the largest published result for such a problem. We therefore ask the question: Can we now claim that 1.7×10^{10} is the largest finite element system that has been solved to date?

6 Acknowledgments

We would like to thank the following people for their roles in supporting this work: Gerhard Wellein and Georg Hager, Regionales Rechenzentrum Erlangen (RRZE), Erlangen, Germany; Rüdiger Wolff, SGI München, München, Germany; The people at Computer Services for Academic Research (CSAR), Manchester, England; and Ralf Ebner, Leibniz Rechenzentrum (LRZ), München, Germany.

References

[1] M. Adams, H. Bayraktar, T. Keaeny, and P. Papadopoulos. Ultrascalable implicit finite element analyses in solid mechanics with over a half a billion degrees of freedom. In *Proceedings of Supercomputing SC'2004*, Pittsburgh, Pennsylvania, November 2004.

[2] R. Barrett, M. Berry, T. F. Chan, J. Demmel, J. Donato, J. Dongarra, V. Eijkhout, R. Pozo, C. Romine, and H. Van der Vorst. *Templates for the Solution of Linear Systems: Building Blocks for Iterative Methods, 2nd Edition.* SIAM, Philadelphia, PA, 1994.

[3] B. Bergen and F. Hülsemann. Hierarchical hybrid grids: data structures and core algorithms for multigrid. *Numerical Linear Algebra with Applications*, 11(2–3):279–291, March–April 2004.

[4] J. Bey. Tetrahedral grid refinement. *Computing*, 55(4):355–378, 1995.

[5] William L. Briggs, Van Emden Henson, and S. F. McCormick. *A Multigrid Tutorial.* SIAM, Philadelphia, PA, 2000.

[6] J. Demmel, S. Eisenstat, J. Gilbert, X. Li, and J. Liu. A supernodal approach to sparse partial pivoting. *SIAM Journal on Matrix Analysis and Applications*, 20(3):720–755, 1999.

[7] J. Demmel and X. Li. Making sparse gaussian elimination scalable by static pivoting. In *Proceedings of Supercomputing SC'1998*, Orlando, Florida, November 1998.

[8] R. Falgout and U. Yang. hypre: A library of high performance preconditioners. In Peter M. A. Sloot, Chih Jeng Kenneth Tan, Jack Dongarra, and Alfons G. Hoekstra, editors, *International Conference on Computational Science (3)*, volume 2331 of *Lecture Notes in Computer Science*, pages 632–641. Springer, 2002.

[9] V. Henson and U. Yang. Boomeramg: a parallel algebraic multigrid solver and preconditioner. *Applied Numerical Mathematics*, 41:155–177, 2002.

[10] F. Hülsemann, B. Bergen, and U. Rüde. Hierarchical hybrid grids as basis for parallel numerical solution of PDE. In H. Kosch, L. Böszörményi, and H. Hellwagner, editors, *Euro-Par 2003 Parallel Processing*, volume 2790 of *Lecture Notes in Computer Science*, pages 840–843, Berlin, 2003. Springer.

[11] F. Hülsemann, S. Meinlschmidt, B. Bergen, G. Greiner, and U. Rüde. *gridlib* – a parallel, object-oriented framework for hierarchical-hybrid grid structures in technical simulation and scientific visualization. In S. Wagner, W. Hanke, A. Bode, and F. Durst, editors, *High Performance Computing in Science and Engineering Munich 2004*, pages 37–49. Springer, 2005. accepted for publication in the second joint HLRB and KONWIHR Result and Reviewing Workshop.

[12] George Karypis and Vipin Kumar. *MeTis: Unstrctured Graph Partitioning and Sparse Matrix Ordering System, Version 2.0*, 1995.

[13] George Karypis and Vipin Kumar. Parallel multilevel k-way partitioning schemes for irregular graphs. Technical Report 036, University of Minnesota, Minneapolis, MN 55454, May 1996.

[14] George Karypis and Vipin Kumar. Multilevel k-way partitioning scheme for irregular graphs. *Journal of Parallel and Distributed Computing*, 48(1):96–129, 1998.

[15] R. Rabenseifner. Comparison of parallel programming models on clusters of smp nodes. In *Proceedings of the 45nd Cray User Group Conference. (CUG)*, Columbus, Ohio, May 2003.

[16] R. Rabenseifner and G. Wellein. Comparison of parallel programming models on clusters of smp nodes. In *Proceedings of the International Conference on High Performance Scientific Computing*, pages 409–426, Hanoi, Vietnam, March 2003.

[17] U. Trottenberg, C. Oosterlee, and A. Schüller. *Multigrid*. Academic Press, San Diego, CA, 2001.