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

Lehrstuhl für Informatik 10 (Systemsimulation)

Hierarchical hybrid grids:
A framework for efficient multigrid on high performance architectures

Ben Bergen, Frank Hülsemann

Lehrstuhlbericht 03-5

Abstract

For many scientific and engineering applications it is often desirable to use unstructured grids to represent complex geometries. Unfortunately the data structures required to represent discretizations on such grids typically result in extremely inefficient performance on current high-performance architectures. Here we introduce a grid framework using patch-wise, regular refinement that retains the flexibility of unstructured grids while achieving performance comparable to that seen with purely structured grids. This approach leads to a grid hierarchy suitable for use with geometric multigrid methods thus combining asymptotically optimal algorithms with extremely efficient data structures to obtain a powerful technique for large scale simulations.

1 Introduction

We begin in section 2 by discussing the underlying causes of the poor performance experienced when using purely unstructured grids. This will be followed in section 3 by a description of the Hierarchical Hybrid Grids(HHG) framework which offers one possible solution to this problem. Also in this section we will consider a caveat of the communication model being used and show that it does not have an adverse effect on the convergence rates obtained from a geometric multigrid algorithm. Finally, we will present performance results obtained on a Hitachi SR8000-F1 super computer and on a Intel Pentium 4 personal computer for both a purely unstructured implementation and the HHG implementation.

2 Problems With Unstructured Grids

In this section we will look at three causes of poor performance on unstructured grids. In all three cases the underlying culprit is the type of data structure required to represent an unstructured grid, so it is worth our time to talk about the general characteristics of such data structures before we consider why they cause performance problems.

For our purposes when we talk about an unstructured grid we are really referring to a element mesh that might be used to represent a problem domain on which we would like to solve some equation or system of equations using a finite element or finite volume discretization. Such discretizations lead to generally sparse matrices that represent couplings between unknowns in the grid. Because the resulting matrix is sparse it makes sense to only store the non-zero values[1], however, because it is unstructured we must introduce some type of indirect indexing so that we can locate the non-zero entries on a row when we are applying an update. As it turns out this last statement contains the causes of all three of the problems that we will consider: First, indirect indexing means that we do not know which value we will need next; second, it is unlikely that related unknowns will be in close proximity to one another; and finally, the dereferencing needed to locate the positions of the non-zero values will require many integer operations. These causes correspond respectively to the problems discussed in the following subsections.

2.1 Prefetch Optimizations

This first problem we will address is caused by the extra level of indirection introduced by the use of indirect indexing. Because the current memory bandwidth available is not fast enough to keep up with the demands of modern CPUs[6], the ability to prefetch data before it is needed is of critical importance in producing efficient codes. Whether or not data can be prefetched depends largely on the predictability of data access. If the compiler is unable to determine a pattern of data access then it cannot schedule prefetch instructions which results in a substantial loss in performance. For unstructured grids it is not possible for the compiler to determine the data access patterns due to the indirect indexing of the data.

2.2 Cache Effects

The undesirable cache effects that one encounters with purely unstructured grids are largely due to a lack of *spatial locality*[5] of the underlying data structures used to represent the grid. The term *spatial locality* is used to indicate that if a cache line is retrieved because the processor has requested a particular byte of data[1] then the rest of the data in the cache line are likely to

[1]The specific byte requested by the CPU is called the **critical word**

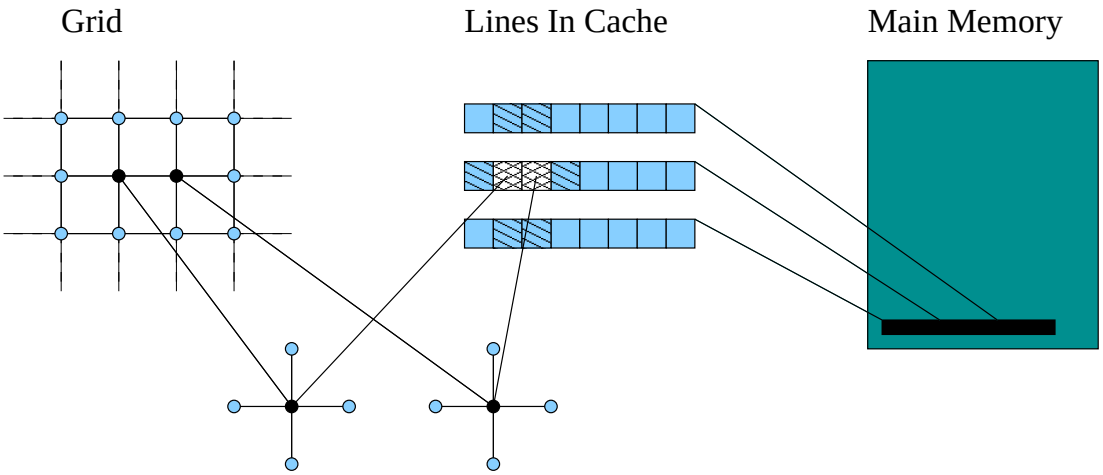

Figure 1: To understand what is meant by spatial locality consider the following scenario: Assume that no data is currently cached and the CPU is to update the left-most black data unknown by taking a weighted average of its nearest neighbors. It begins this process by loading the data unknown above the one being updated. This results in a cache miss which causes the top line in the cache lines diagram to be filled. Because the data is stored in a contiguous array the neighboring value of the right-most data unknown is also now in cache. The CPU now needs the data unknown to the left of the one being updated. This also results in a cache miss which causes the middle line in the cache lines diagram to be filled. Again, because of the contiguous storage scheme, the three data unknowns to the right of the one that caused the cache miss are also now in cache. The same step is repeated for the unknown below the one being updated with the result that when the right-most black data unknown is updated there are no cache misses because all of the required unknowns are already in the cache due to their spatial locality in memory to the unknowns of the previous step. Note that this example ignores the possibility of a cache conflict occurring.

be needed as well. To see how this works in a structured grid consider figure 1. Here we see that not only is the next unknown to be updated likely to be fetched in the case of a cache miss, but also the neighbors of the next unknown, since they are in close proximity to the neighbors of the current unknown in memory. In an unstructured grid it is generally not possible to come up with an ordering of the unknowns that will result in such favorable memory access patterns which consequently leads to poorer performance.

2.3 Instruction Mix

The final problem that we will consider has to do with the types of instructions that must be executed to access the non-zero entries of a matrix that represents an unstructured grid. Again indirect indexing is the underlying cause of the problem, however, in this case, we are not worried about the irregularity of the data access. Instead the concern here is the mix of operations that must be performed in updating an unknown. Most modern CPUs now have multiple *Arithmetic Logic Units* which means that under favorable conditions they are able to perform several operations at each clock cycle. Depending on the hardware capabilities of the processor and the mixture of tasks to be performed, i.e., integer versus floating point operations, many processors can achieve on-chip parallelism that can vastly increase performance. To see how indirect indexing might cause a problem in this context consider the schematic processor execution core in figure 2 which is typical of current (super-scalar) architectures and is able to perform up to four floating point operations per clock cycle. It should be clear that such processors might be at a disadvantage if one is trying to work with purely unstructured grids. For example, imagine that we would like to update an unknown by taking a weighted average of its neighbors. If the unknown in question has four neighbors then we would need five integer operations and seven floating point operations which would take no fewer than six clock cycles to complete[2]. On a structured grid the same computation would take only three clock cycles. In this case the problem is not only that there is more work to be done than if we were working

[2]This does not include load and store operations and also assumes that the weights are constant.

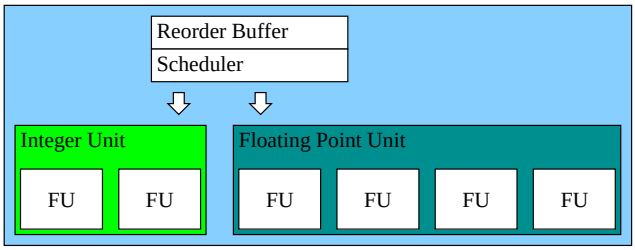

Figure 2: Processor execution core with two integer units and four floating point units. The efficiency of an algorithm often depends on having a favorable mix of instructions. The indirect indexing used in representing unstructured grids requires many integer operations and may cause an imbalance that will have a negative effect on performance.

with the structured grid, there is also the matter of scheduling to consider. Because the CPU only has two integer units the floating point operations cannot begin until the memory locations of at least two of the neighboring points have been resolved. Therefore, using an unstructured grid results in a bad instruction mix for this CPU. If the CPU had an equal number of integer and floating point units we would not see as much discrepancy in the number of clock cycles needed to complete the task so this problem is clearly architecture dependent.

3 Hierarchical Hybrid Grids

So far we have seen that the data structures needed to represent unstructured grids can cause some performance problems. Still, there are many applications for which we would like to use unstructured grids because of the flexibility they provide. In the next few subsections we present a possible solution to these problems that uses patch-wise regular refinement of a purely unstructured input grid to achieve nearly the same performance as that seen on structured grids(some early steps in this direction appear in [4]). We begin our discussion with an outline of the basic principles of HHG.

3.1 Basic Principles

The idea behind HHG is as follows: We begin with a purely unstructured input grid. This grid is assumed to be fairly coarse and is only meant to resolve the geometry of the problem being solved. This means, for example, that the input grid will resolve different material parameters but will not attempt to resolve problems such as shocks or singularities. 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 both to ensure the proper resolution and to handle mathematical problems like those already mentioned. We then apply regular refinement to each patch of the input grid. Doing this successively, generates a nested grid hierarchy that is suitable for use with geometric multigrid algorithms. For an example of what such a grid hierarchy looks like consider figure 3.

Each grid in the new hierarchy is still logically unstructured, however, by using the right data structures we can now exploit the regularity of the patches. What we would ultimately like is for the neighbors of the interior points to occur at known, regular offsets from those points. This can now be accomplished by allocating separate blocks of memory for each individual patch. Then, patch-wise, we will have memory access patterns similar to a structured grid thus allowing stencil based operations for such things as smoothing, residual calculation, and transfer operations, and thereby avoiding the performance penalties associated with indirect indexing. This is essentially a variant of using block structured grids with the advantage that the structure and the resolution of the block connections are generated automatically. We will see in the next two subsections how the blocks are joined.

3.2 Logical Objects

Exploiting the patch-wise regularity of our new grids is such a nice idea that it makes sense to ask if there is some more structure in this approach that may be exploited. The answer is yes, so long as we decompose the grid into the correct logical objects. Figure 4 shows that there

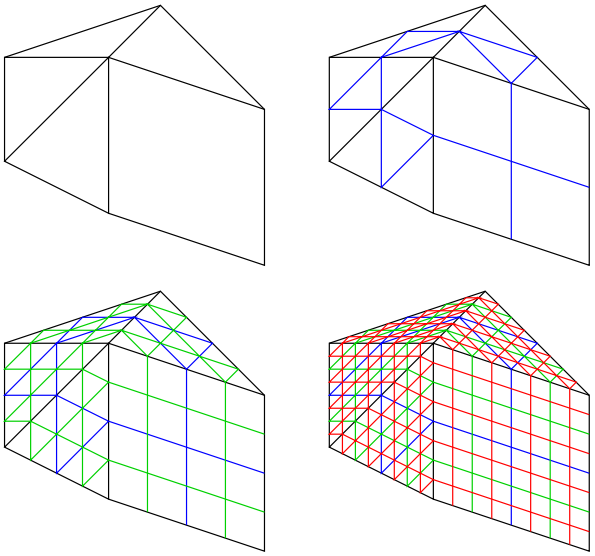

Figure 3: Beginning with the hybrid input grid on the top left, regular refinement is applied once to obtain the grid on the top right, and again to obtain the lower left grid, and again to finally obtain the lower right grid. After two levels of refinement(lower left grid) we can see that each patch has a structured interior. This structure may now be exploited to achieve higher performance.

is also a regular geometric structure along the edge unknowns. In order to use this regularity we again need to use the right data structures, i.e., each edge must be treated as a separate object in memory so that its neighbors occur at regular offsets. The relationship between the geometric structure of an edge and its memory representation is shown in figure 5 for an edge between two triangles.

Because they are inherently unstructured, it is not possible to treat the vertices of the patches in a structured manner, however, it does makes sense to treat them as separate memory objects since they are updated differently from the elements and edges. This implies that we should decompose the input grid into elements, edges, and vertices[3], and apply refinement to each object individually.

3.3 Communication Model

As a consequence of our grid decomposition scheme we must now introduce some communication between objects. This is necessary to ensure that every interior point in a structured region may be updated in the same manner and also to resolve the dependencies that exist between objects, e.g., some interior points of an element have neighboring points that are interior to an neighboring edge. We must therefore add *ghost* values around the edges of each logical object which correspond to the interior points of that objects neighbors. Interior and ghost points are shown in figure 6 for an edge and its neighboring elements.

In addition to communicating dependencies we must also rethink what an update of the unknowns on the grid should look like. In a normal treatment we would just loop over all unknowns in the grid applying an operation. However, this would simply be treating the refined grids as global unstructured grids and would then suffer from the same performance problems mentioned before. Exploiting the regularity of our new data structures is slightly more complicated although still straight forward.

The change is that we must now loop over each object type, i.e., elements, edges, and vertices. As an example consider performing a Gauss–Seidel step on one of the grids in out new hierarchy,

[3]In three dimensions we have elements, faces, edges, and vertices where both the vertices and edges exhibit the unstructured nature of the input grid. Therefore, the elements and faces may be regularly refined and the resulting structure exploited.

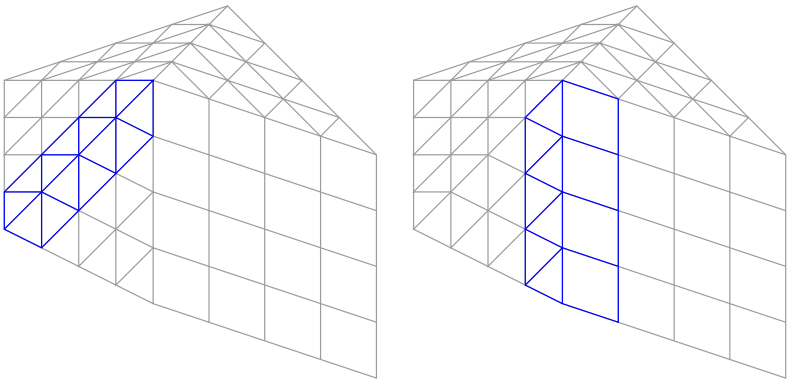

Figure 4: Edges also exhibit a regular structure which may be exploited. The grid on the left shows the structure of an edge between two triangular patches while the grid on the right shows the structure of an edge between a triangle and a quadrilateral. Although not pictured here, edges between two quadrilaterals also have a regular structure that may be exploited.

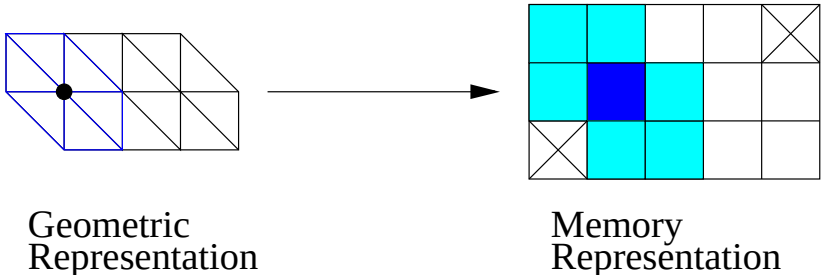

Figure 5: This edge may be expressed as a two-dimensional array in which case the light blue memory squares represent the neighboring points each of which has a regular offset from the dark blue interior point. The X's in the lower left and top right corners of the memory representation denote that these cells are not used. This edge has been regularly refined twice with the result that it has three interior points that may be operated on by the use of stencils.

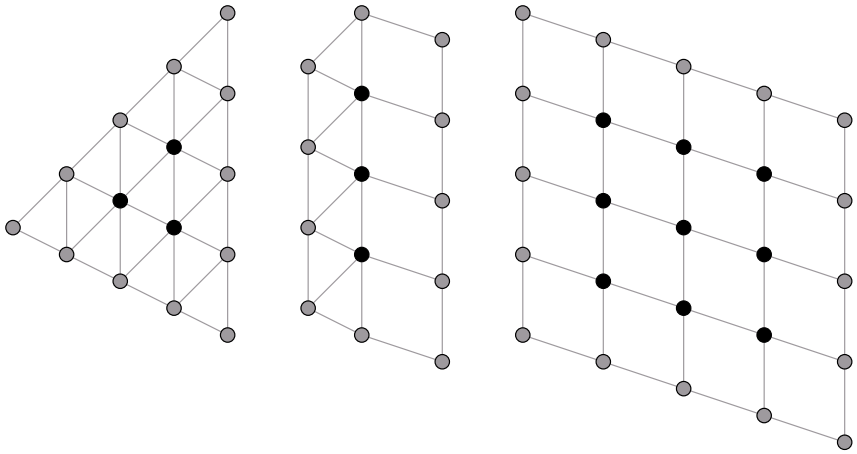

Figure 6: These three objects represent the result of decomposing two patches from the input grid into two elements and an edge. The black interior points may be updated independently provided that the gray ghost values are up to date.

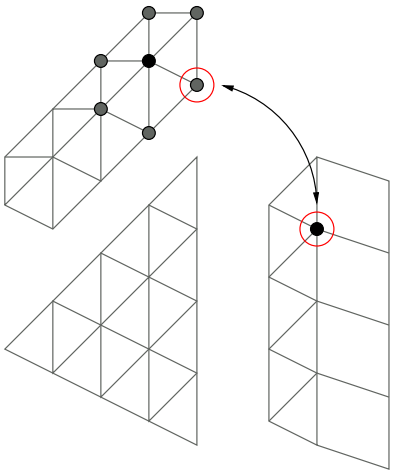

Figure 7: If the edge to the right of the element has already been updated then the edge above the element will not have the current value of the circled point.

```
DO I=1,NUM_VERTICES
   ...smooth vertex(I)...
END DO

update dependencies

DO I=1,NUM_EDGES
   ...smooth edge(I)...
END DO

update dependencies

DO I=1,NUM_ELEMENTS
   ...smooth element(I)...
END DO
```

The differences here are that the edges and elements both contain unknowns that are being updated, and that updates of each object type require dependency information to be traded. Otherwise, the operations are the same albeit with a different ordering of the unknowns.

In order to simplify our implementation and to cut down on the amount of communication that must occur during an update, we have used what might be called a *one-way* communication model. What we mean by this is that each object type only communicates with the next *lower/simpler* object type, i.e., elements only communicate with edges, and edges only communicate with vertices. There is also no direct communication between objects of the same type. For the element object type this communication model does not cause any problems because no interior points of an element depend directly on the interior points of another element[4]. However, this does cause a difficulty for the edge objects. The problem here is that the interior points at each end of an edge are directly dependent on interior points from its two neighboring edges. To see this consider figure 7.

In the course of updating all edge objects, some dependencies will be updated but not communicated since there is no direct communication between the edges. For a Gauss–Seidel iteration this will result in a Jacobi-like[5] iteration for some of the points. However, these missing dependencies do get updated before the next iteration by communication through the elements and for a specific input grid there are a fixed

[4]In this case there is always an edge object in between whose interior unknowns provide a buffer.

[5]For any of these points at most two of its neighbors will have *stale* values from the previous iteration so that this type of update is somewhere between Jacobi and Gauss–Seidel.

number of such points regardless of the level of refinement[6]. This communication model dramatically cuts down on the complexity of the HHG implementation and as we will see in the next section, it seems to have little or no effect on convergence.

3.4 Multigrid Convergence

To determine the effects of our communication model on multigrid convergence we solve the following problem,

$$\begin{aligned} -\Delta u &= f \quad \vec{x} \in \Omega = [0,1]^2 \\ u &= 0 \quad \vec{x} \in \partial\Omega \end{aligned}$$

using a standard finite element discretization. If we use as our input grid the unit square subdivided into eight regular, triangular elements and apply five levels of refinement we obtain a similar problem to that for which multigrid convergence rates are given in [2] where $h = \frac{1}{64}$. We apply four-color Gauss–Seidel to the elements and lexicographic Gauss–Seidel/Jacobi to the edges. Using the same procedure as in [2] we calculate our convergence rates by averaging the first five $\mathcal{V}$-cycles. The convergence rate for a $\mathcal{V}(2,1)$ cycle for HHG is 0.09 which is comparable to the rate of 0.04 for red-black Gauss–Seidel as reported in [2] and much better than Jacobi with a reported convergence rate of 0.24. We can therefore conclude that the ignored edge dependencies in our communication model do not have a serious effect on convergence.

4 Performance Results

In the following two sections we give results for solving the *Poisson problem* with patch–wise constant coefficients.

4.1 2D Results

Figure 8 shows performance results for varying levels of refinement produced using a single processor on a Hitachi SR8000–F1. Here, the left–most bar of the graph shows the MFLOP/s rate for performing colored Gauss–Seidel only on the finest level of refinement using the HHG data structures. This achieves the best overall performance due to the long line lengths in the inner–most loops of the smoothing algorithm. The second and third bars show results using the HHG data structures for smoothing on all levels, and geometric multigrid respectively. In both cases there is a slight performance loss caused by doing work on the coarse grids. The right–most bar shows the results for applying Gauss–Seidel to the finest level of refinement when the grid is treated in a purely unstructured manner using a *Compressed Row Storage*[1] storage scheme. Clearly there is a substantial gain in performance over the purely unstructured case when any of the HHG algorithms are used with the result that in some cases half of the processors theoretical peak performance is achieved.

4.2 3D Results

Figure 9 shows performance results for varying levels of refinement produced using all eight processors of a single node on a Hitachi SR8000–F1. These implementations use COMPAS(Cooperative Microprocessors in a Single Address Space) for producing shared memory parallelization. Here, the left–most bar of the graph shows the results obtained for a highly optimized Gauss–Seidel implementation for a purely unstructured grid using a *Jagged Diagonals Storage*[7] storage scheme. Great care has been taken in the implementation of this algorithm to exploit the procedural and architectural modifications made respectively to the IMB POWER instruction set, and to the IBM PowerPC

[6]Each edge object has two such points, one at each end. This does not depend on how many times an edge has been regularly refined.

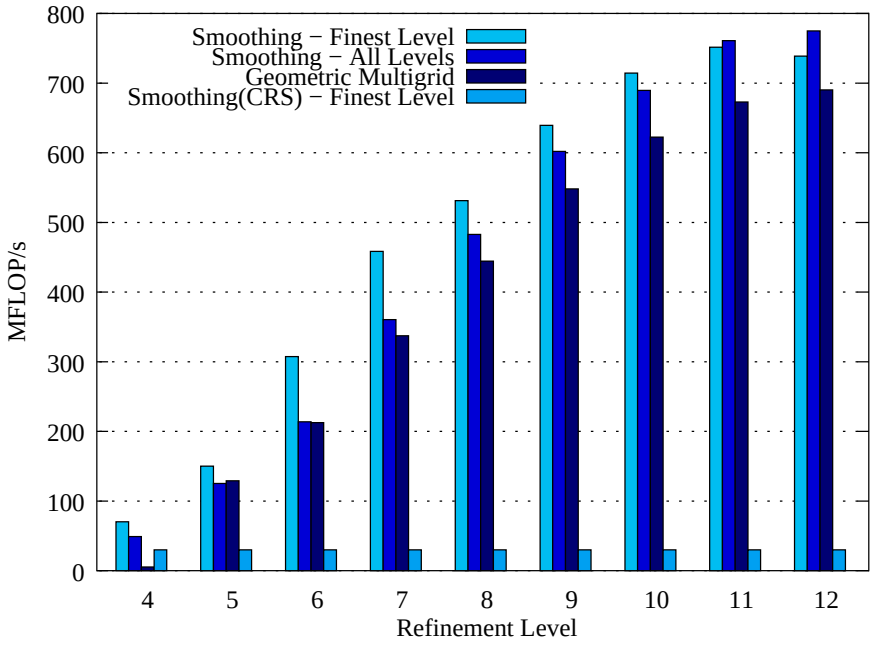

Figure 8: Results for Hitachi SR8000–F1 using a single CPU with theoretical peak performance of 1.5 GFLOP/s.

processors[3] used in the Hitachi SR8000–F1. In spite of this, the purely unstructured implementation still fails to achieve a high percentage of the nodes theoretical peak performance of 12 GFLOP/s. On the other hand, as the second and third bars show, the HHG data structures, when applied to homogeneous tetrahedral and hexahedral meshes respectively, again attain extremely high MFLOP/s rates. In fact, for seven levels of refinement, the hexahedral implementation achieves more than half of the theoretical peak.

Figure 10 shows similar results to figure 9, but for a single Intel Pentium 4 processor running at 2.4 GHz with a theoretical peak performance of 4.8 GFLOP/s. These results are qualitatively the same as those in figure 9, as they again show that the HHG data structures outperform a tuned JDS implementation. However, they are quite relevant, since they show that the HHG data structures are also very efficient on commodity hardware. At the time of this writing the Intel Pentium 4 is a popular processor for use in building *Beowulf Clusters*[7] which are routinely used for performing large scale simulations. In the past, commodity processors have not been as sensitive to the types of data structures used to represent unstructured grids, however, as vendors strive to overcome memory bandwidth limitations with more sophisticated designs and better compiler optimizations, truly efficient algorithms begin to show their worth as we have seen here.

5 Conclusion

As scientist and engineers attempt to solve more difficult problems on complex geometries there will be an increasing need to turn to unstructured grid representations of problem domains. Traditional data structures for these grids fail to make efficient use of the resources of many high-performance architectures. Fortunately, as we have shown, it is possible to create efficient data structures that achieve extremely high performance as measured in MFLOP/s that still retain the salient features of purely unstructured grids. The process of developing such data structures requires careful consideration of

[7] A Beowulf Cluster is a cost effective alternate to large supercomputers created by connecting commodity or *off the shelf* computers over a fast network.

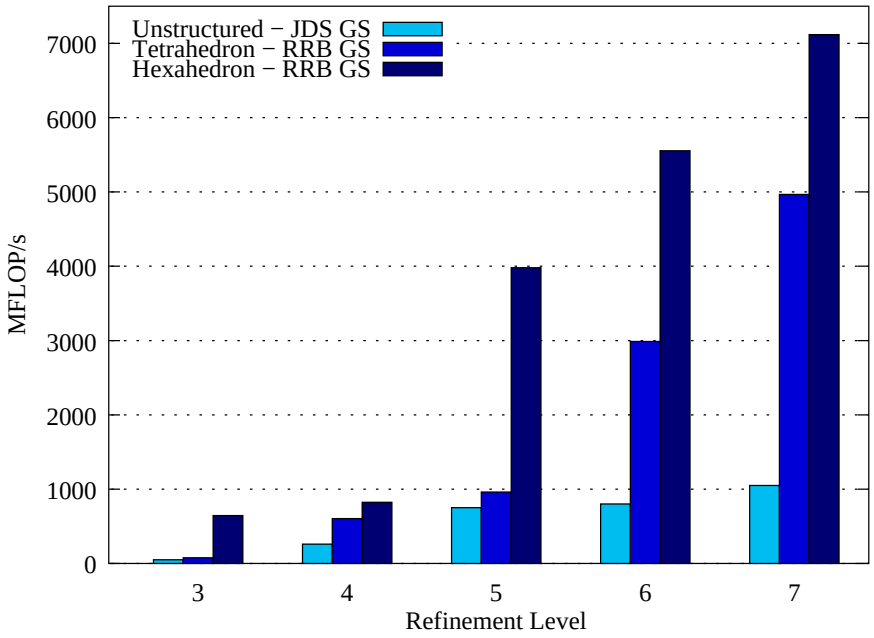

Figure 9: Results for Hitachi SR8000–F1 using 8 CPUs with combined theoretical peak performance of 12 GFLOP/s.

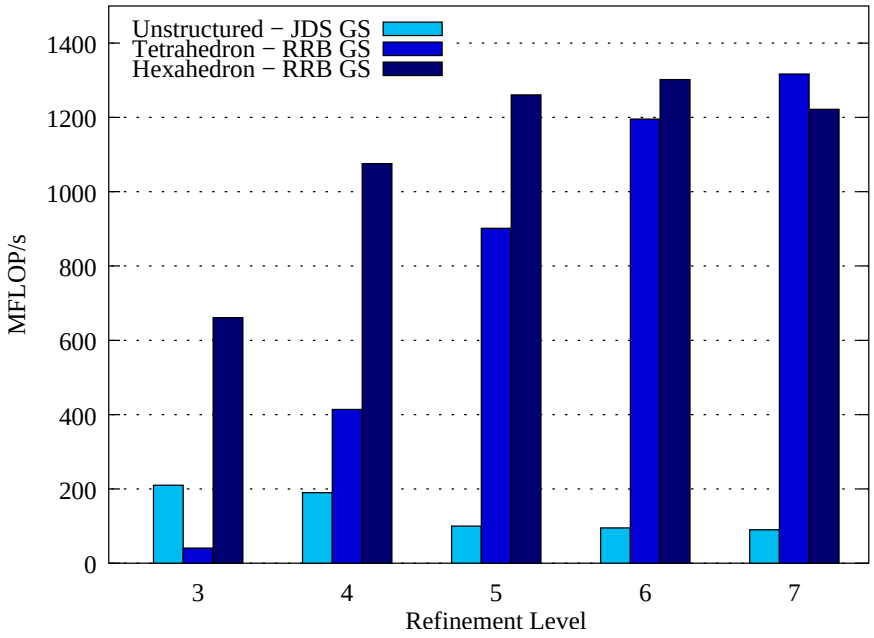

Figure 10: Results for single Pentium 4, 2.4 GHz processor with theoretical peak performance of 4.8 GFLOP/s.

the underlying processes that are being carried out and an increasingly acute understanding of how high-performance architectures work. However, in the end, we see that although HHG requires more elaborate data structures than a conventional approach the gain in performance is worth the effort.

References

[1] 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.

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

[3] C. May et al.(Ed.). *The PowerPC architecure: A specification for a new family of RISC processors*. Morgan Kaufmann Publishers, 2nd edition, 1994.

[4] F. Hülsemann, P. Kipfer, U. Rüde, and G. Greiner. gridlib: Flexible and efficient grid management for Simulation and Visualization. In *Computational Science - ICCS 2002*, Lecture Notes in Computer Science, pages 652–661. Springer, 2002.

[5] L. Stals and U. Rüde. Techniques for Improving The Data Locality of Iterative Methods. Technical Report MRR97-038, School of Mathematical Science, Australian National University, October 1997.

[6] C. Weiß, W. Karl, M. Kowarschik, and U. Rüde. Memory Characteristics of Iterative Methods. In *Proceedings of the ACM/IEEE SC99 Conference*, Portland, Oregon, November 1999.

[7] G. Wellein, G. Hager, A. Basermann, and H. Fehske. Fast sparse matrix–vector multiplication for teraflop/s computers. In *High Performance Computing for Computational Science – VECPAR 2002*, Lecture Notes in Computer Science, pages 287–301. Springer, 2003.