Hierarchical Hybrid Grids: Achieving TERAFLOP Performance on Large Scale Finite Element Simulations

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

The design of the Hierarchical Hybrid Grids framework is motivated by the desire to achieve high performance on large-scale, parallel, finite element simulations on super computers. In order to realize this goal, careful analysis of the low-level, computationally intensive algorithms used in implementing the library is necessary. This analysis is primarily concerned with identifying and removing bottlenecks that limit the serial performance of multigrid component algorithms such as smoothing and residual error calculation. To aid in this investigation two metrics have been developed: the Balance metric, and the Loads Per Miss metric. Each of these metrics makes assumptions about the interaction of various data structures and algorithms with the underlying memory subsystems and processors of the architectures on which they are implemented. Applying these metrics generates performance predictions that can then be compared to measured results to determine the actual characteristics of an algorithm/data structure on a given platform. This information can then be used to increase performance.

In this paper, we first present an overview of the Hierarchical Hybrid Grids framework. Next, we introduce the details of the two performance metrics. These metrics are then applied to three different data structures used to implement a Gauß–Seidel smoothing algorithm. Performance results and an interpretation of the underlying interactions of the data structures with several relevant supercomputing architectures are given. Finally, we present a brief discussion of some performance results of the HHG framework, followed by some concluding remarks.

Introduction

In practice, it is often the case that the codes used to perform simulations in scientific computing applications achieve only a small percentage of the theoretical peak performance of the CPU on a given architecture. In many cases, this poor performance can be attributed to memory bandwidth limitations that are imposed on the underlying algorithm by an imbalance between the CPU throughput and the burst rate of the memory subsystem. Although these...
Hierarchical Hybrid Grids

performance problems cannot generally be avoided, it is unacceptable to point to their ubiquitousness and simply accept results of 5 – 10% of the CPU’s theoretical peak performance without understanding why these limitations exist. It is especially important in situations where funding for scientific research is at stake to be able to adequately explain to other scientists and lay-people why a certain level of performance is all that can be achieved, and may in fact represent a substantial improvement in terms of overcoming limitations of the underlying hardware. A possible approach that can be used to elucidate the real interactions that occur during computation, in an attempt to understand what level of performance is actually possible, is to hypothesize various models of memory access that may be compared to measured results. Through the use of these models, which we will refer to here as metrics, we will show that it is possible, not only to adequately explain, but also to accurately predict what percentage of the theoretical CPU throughput can be achieved by an algorithm on a particular architecture.

In this paper, we introduce two performance metrics: the balance metric and the loads per miss metric. Each of these is formulated by making different assumptions about the memory access characteristics of an algorithm implemented using a specific set of data structures\(^1\). These metrics are then used to predict and interpret measured results of various implementations of the standard, lexicographic Gauß-Seidel algorithm, which is a popular smoother for use with geometric multigrid algorithms [7, 10]. As part of this analysis, we give a brief discussion of the performance of a code implemented using the hierarchical hybrid grids HHG library presented in [3], and show that good serial performance is necessary to be able to achieve reasonable scalability when performing large-scale simulations. We begin our investigation with an overview of the HHG framework.

Hierarchical Hybrid Grids (HHG)

The hierarchical hybrid grids framework is an attempt to combine some of the flexibility of unstructured grid representations with the performance and efficiency of structured grid data structures. To realize this goal, HHG makes use of a combination of regular refinement and grid decomposition that allows structured regions of the refined grid hierarchy to be treated using stencil-based data structures. These data structures allow for more efficient implementations of the standard multigrid component algorithms, both in terms of performance and memory usage. The fact that regular refinement, as applied

\(^1\)As we will see, it is important to consider not only what algorithm is being analyzed, but also how that algorithm is implemented, since this affects the way in which data are accessed.
Regular Refinement and Grid Decomposition

The basis for the HHG approach can be described as follows: Beginning with a purely unstructured input grid, regular refinement is applied by adding new vertices along each edge of the input grid and then connecting the appropriate vertices with new edges and faces. This results in a new, logically unstructured grid that is a superset of the input grid. This process is repeated successively on each new grid, forming a nested hierarchy. Figure 1 shows an example of regular refinement of a two-dimensional input grid. Each grid in the hierarchy is then decomposed into the primitive types: elements, faces, edges, and vertices. This decomposition allows each class of primitive to be treated separately during the discretization and solver phases of the simulation, so that the structure of the added points can be exploited. This is essentially a variant of a block-structured approach, with the advantage that the resolution of the block interfaces is generated automatically.

In applying HHG, we make the assumption that the input grid is fairly coarse and that it primarily resolves only the large-scale features of the domain. The most favorable problem type for this approach begins with an input grid that is patch-wise constant with respect to material, and that requires several levels of regular refinement to properly resolve the fine-scale features of the domain. This type of problem leads to a grid hierarchy with large collections of structured unknowns that have a constant-coefficient discretization within each patch.

The current HHG implementation is designed to accommodate three-dimensional grids. The two-dimensional example is included because it is easier to visualize.
Hierarchical Hybrid Grids

In order to exploit the structure of the refined primitives, data structures with contiguous memory must be allocated for each primitive. Then, the primitives of each type that has a regular structure of unknowns can be treated as independent structured grids. This allows the use of stencil-based discretization and solver techniques that are more efficient than those available for use with standard unstructured grid data structures, as we will show in our experimental results. As a consequence of the grid decomposition and the use of separate memory data structures for each primitive, a halo containing dependency information for the unknowns adjacent to each primitive must be stored and updated. This introduces some local copying that may be treated with techniques analogous to those used in distributed memory parallelism on structured grids.

Because of the grid decomposition, it is convenient to apply component algorithms to each set of primitives as a group, with local dependencies being updated only after all primitives of a particular type have been altered. This means, for instance, that in applying a smoothing algorithm such as Gauß–Seidel, all vertices are updated, followed by all edges, then all faces, and finally, all elements. An example of a generic update is shown in Algorithm 1.

**Algorithm 1** Algorithm Decomposition for HHG

1: **for** each vertex **do**
2: 
3: **end for**
4: **for** each edge **do**
5: 
6: **end for**
7: **for** each face **do**
8: 
9: **end for**
10: **for** each element **do**
11: 
12: **end for**
13: 
14: 
15: 
16: 
17: 
18: **end for**

As a consequence of this approach, certain dependencies that exist between primitives of the same type are ignored. Fortunately, as shown in [3], this does not adversely affect convergence, and this design allows a much more efficient implementation of parallel communication, which is the topic of the next section.
Parallelization Strategy

The HHG framework employs a distributed memory parallel communication model using the message passing interface (MPI). In order to avoid excessive latency during inter-node communication, certain design paradigms that decrease the number of messages that must be sent have been instituted. This design exploits the principle, introduced in the last section, that each type of primitive is updated as a group. Because of the grid decomposition, primitives on a particular processing element (PE) can be updated without requiring any parallel communication. This means that, rather than sending many small messages over the interconnect, each type of primitive can have its parallel dependencies updated as a single large message. This reduces communication latency and allows a more efficient implementation.

In the case of a vertex or edge primitive, parallel communication requires that updated values be copied into a message buffer before they can be communicated to other PEs. This is a consequence of the unstructured nature of the couplings between these types of primitives with their adjacencies, e.g., an edge may adjoin arbitrarily many face primitives at an interior boundary between PEs. However, since a face primitive can only be shared by at most two PEs, the parallel implementation for this type of primitive is particularly elegant. In this case, a face that occurs on an internal boundary requests a contiguous block of memory from a single message buffer. This memory block is used to represent the unknowns defined on that face and thus avoids a copying stage between the face data structure and a separate MPI buffer. In this way, all parallel face dependencies can be updated between two PEs by sending and receiving only one message on each PE, with no additional copying between buffers necessary.

HHG and 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 the coarser 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 correc-
tion 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".

Because of the grid decomposition employed in the HHG framework, component algorithms developed for structured multigrid implementations are directly applicable to the structured regions of the refined grid. Care must be taken on the unstructured regions. However, these do not differ substantially from standard unstructured grid treatments so that implementing a multigrid algorithm on the HHG data structures is quite straightforward.

**The Balance Metric (BM)**

The performance of many algorithms of interest in scientific computing is said to be *memory bound*. This means that the algorithm is not able to run at the theoretical peak processor speed because the memory subsystem of the machine is not able to provide data quickly enough to keep up with the CPU. In such cases, it is useful to have a metric to predict what percentage of the processor’s theoretical peak can be achieved by a given algorithm. Therefore, we introduce the *Balance Metric* which may be defined in the following manner: First, we determine the *Machine Balance* \(B_{m}\) of the architecture in question. We define this to be the ratio of the theoretical number of double words per second that the memory subsystem can move to the CPU to the theoretical number of floating point operations per second that can be performed by the CPU. We denote this ratio by \(B_{m}\). Next, we compute the *Algorithmic Balance* of the particular algorithm that we are trying to measure. This is defined as the ratio of the number of double words that must be loaded to update one iterate to the number of floating point operations that are performed per update of one iterate. We denote this ratio by \(B_{a}\). Then, the Balance Metric for
our algorithm on the architecture in question is given by the ratio:

$$B = \frac{E_m}{E_a}.$$  

One can think of $B$ as representing the maximum application performance that can be achieved under the limits introduced by the Balance Metric as a fraction of peak performance. This can be used to predict the performance of a particular algorithm in MFLOPS/s by multiplying $B$ with the $R_{peak}$ of the machine in question. In general, consistency with this metric suggests that an algorithm is memory bandwidth limited. When this is the case, the metric serves as an upper bound on the performance that can be achieved on the architecture in question. Note that this version of the metric is based on theoretical machine and algorithm properties, and does not take variations such as cache effects or actual sustainable memory bandwidth into account. However, the metric can be tuned to reflect finer grained properties of the architecture or algorithm to more closely model particular parts of the system, e.g., the metric can be tuned to model memory accesses from an L3 cache if it is suspected that a large working set of data stays in that cache level during the course of computation. One might determine that this is the case using the Loads Per Miss Metric introduced in the next section.

**The Loads Per Miss Metric (LPMM)**

In many cases, the behavior of an algorithm, with respect to memory bandwidth limitations, changes with the size of the problem being solved. This is especially true on architectures with deep memory hierarchies, i.e., those that have caches. In such instances, we need to identify: First, whether or not the algorithm is actually memory bound, and second, if necessary, the threshold at which the behavior of the algorithm changes. To this end, we introduce an algorithm specific metric for measuring how efficiently the CPU is able to use data that has been loaded into a particular cache level, i.e., we are attempting to predict the cache hit rate. We refer to this metric simply as the *Loads Per Miss Metric* and it is related to the cache hit rate by

$$\text{cache hit rate} = 1 - \frac{1}{\text{LPMM}}.$$  

As the name implies, this metric is defined as the ratio of loads per some particular cache level miss with the added assumption that each cache line of data is used in a *maximal* way each time it gets loaded into the cache. Another way to think of this, is that the metric measures the *temporal* characteristics
of the algorithm, in the sense that, algorithms that have good temporal locality [11] of data will have a high ratio of loads to misses for problem sizes where all or some part of the data fits into cache. Such algorithms, are exactly those that are less likely to suffer from memory bandwidth limitations.

As an example of how this metric might be counted\(^1\) for a particular algorithm, consider Figure 2, and assume that we are performing a standard Gauß-Seidel smoothing step on a Poisson problem that has been discretized with the constant-coefficient, nine-point stencil in Part (B) of the figure. Updating a single unknown then has the form

\[
u_{i,j} = \frac{1}{c} \left( f_{i,j} + c \cdot u_{i+1,j} + nw \cdot u_{i-1,j+1} + n \cdot u_{i,j+1} + ne \cdot u_{i+1,j+1} + sw \cdot u_{i-1,j-1} + s \cdot u_{i,j-1} + se \cdot u_{i+1,j-1} + w \cdot u_{i-1,j} \right).
\]

When the cache line of data, displayed in gray in Part (A), is loaded into the cache we incur one cache miss. Assuming that the single cached datum for an unknown, denoted by the shaded region, stays in the cache as long as it is useful to the computation, then, on average, it will be loaded into a register nine times, i.e., once as it plays the role of each member of the stencil, as the stencil moves over the points \(a, b, \ldots, i\). For the load function \(f\), we will only have one load per miss. Each of these numbers is then multiplied by the number of double words that make up a cache line, which for our example,

\(^1\)The example given here is only one possibility for counting this metric. If we were to make different assumptions about the cache behavior of the algorithm, we would get a different count.
yields

\[
\frac{9 \times 4DW + 1 \times 4DW}{2 \text{MISSES}} = 20 \frac{\text{LOADS}}{\text{MISS}}.
\]

Using this ratio, we expect that for any measured result of this example algorithm, if we observe that the loads per miss are greater than or equal to 20, then the processor is working primarily out of cache and the algorithm is not severely memory bound. This would imply that the cache is large enough to hold at least three lines of data (or equivalently, three planes of data in 3D) for as long as they are useful. When this is the case, we say that the algorithm is cache bound, meaning that performance is bound by the bandwidth and latency properties of the particular cache level under consideration. If the number of unknowns is increased systematically, eventually this working set of data will fall out of cache and the performance will depend on the efficiency of the main memory subsystem. A possible remedy for this that would preserve spatial locality would be to introduce some type of blocking to the algorithm [6,8,11]. Of course, in a real simulation code, we would expect many more loads per miss, because of the larger cache line size.

Data Structures Analysis

In order to better understand the effects of using different data structures to represent sparse matrices, we consider the following experiment: Beginning with a structured partitioning of the unit cube, we form a finite difference discretization of the Poisson equation with Dirichlet boundary conditions, resulting in the standard symmetric, banded matrix with 27 non-zero entries per row (excluding boundary effects). The performance of three data structures is benchmarked by performing ten iterations of a Gauß–Seidel smoothing algorithm on each. The data structures used in the experiment are as follows: The CRS format [1]; 27-point, variable-coefficient stencils (VCS); and 27-point, constant-coefficient stencils (CCS). The performance of the three data structure can then be analyzed and compared on a more level playing field. The Gauß–Seidel algorithm was selected because it is a popular smoother for use with multigrid algorithms [7,10]. Recall that CRS employs indirect indexing to store only the non-zero entries of a sparse matrix. Algorithm 2 illustrates the steps needed to update \( N \) unknowns using this storage scheme where the discretization weights are stored in the \( \text{values} \) array.

For the results of the experiment presented here, each test program was implemented as a C++ program that calls a Fortran 77 subroutine that implements the appropriate Gauß–Seidel algorithm for the corresponding data
Hierarchical Hybrid Grids

Algorithm 2 Gauß–Seidel using Compressed Row Storage for $Au = f$.

1: \textbf{for} $j \leftarrow 1$ \textbf{to} $N$ \textbf{do}
2: \hspace{1em} $tmp \leftarrow f(j)$
3: \hspace{1em} \textbf{for} $i \leftarrow \text{offsets}(j) + 1$ \textbf{to} $\text{offsets}(j + 1) - 1$ \textbf{do}
4: \hspace{2em} $tmp \leftarrow tmp + \text{values}(i) \times u(\text{columns}(i))$
5: \hspace{1em} \textbf{end for}
6: \hspace{1em} $u(j) \leftarrow 1.0/\text{values}(\text{offsets}(j)) \times tmp$
7: \textbf{end for}

structure. The programs were compiled using the same flags on each platform considered. In all cases, a lexicographic ordering of the unknowns was used.

The idea behind this experiment is to try to isolate the effects that the indirect indexing and variable coefficients used by the CRS data structure have on performance. The use of the same underlying grid structure in all three test cases is intended to minimize the ill effects generally associated with cache accesses for the CRS data structure. So, although the compiler will not be able to exploit the spatial locality of the data accesses for the CRS data structure, the actual memory locations accessed by the CRS based algorithm will not be random, and should benefit from a cache based memory subsystem. By comparing the CRS results with the results from the variable-coefficient, stencil-based implementation, we should be able to identify the effect that indirect indexing has on performance. Since the underlying memory accesses are the same, the only difference between these two implementations should be the use of indirect indexing by the CRS data structure. If we then compare the variable-coefficient, stencil-based implementation to the constant-coefficient, stencil-based implementation, we should be able to isolate the performance effect caused by the additional memory bandwidth needed by the variable coefficient data structure.

The approach to performance modeling presented here could be described as bottom-up, \textit{i.e.}, we concentrate on analyzing and achieving robust single processor performance with the goal of scaling to many thousands of processors. This is in contrast to the top-down approach favored by Davis \textit{et al.} \cite{5} who model parallel scalability and then fit single processor performance.

\textbf{Test Platforms}

In this section we give a brief overview of the different architectures used in our tests. This is not meant to be a comparison, just a description of some of the pertinent attributes of each platform with respect to our intended analysis. Table 1 shows the $R_{peak}$ and machine balance for each architecture.
Hitachi SR8000-F1 The Hitachi SR8000 is the oldest platform considered here, first introduced in 1999. In spite of this, it is still an interesting architecture because of its ability to achieve a high percentage of its $R_{peak}$ on certain codes. The SR8000 is based on a modified IBM Power 3 processor [9], with an extension to 160 floating point registers, and the addition of pseudo vector processing (PVP). Although the Power 3 is a standard RISC processor which can utilize dynamic scheduling and out-of-order execution, the modifications made by Hitachi in designing the SR8000 make it behave more like a vector processor. This is due to the fact that the SR8000 employs very aggressive software-based optimizations in implementing PVP. This is analogous to static scheduling, in that, the compiler schedules prefetch and preload requests for data in a pipelined manner. The result is that arithmetic execution can be performed in a non-blocking fashion similar to a vector processor. Prefetch instructions transfer data from main memory into the small (128 KB) L1 data cache, while preload instructions transfer data from main memory directly into the large floating-point register set. The processor is able to execute multiple outstanding prefetches so that prefetching is overlapped with execution, while data that does not fit in the cache is accessed using preload instructions.

While the explanation of PVP is complicated, the implications are simple. In order for the compiler to make efficient use of PVP, the data access patterns of a given code must be visible to the compiler. The SR8000 is designed to be efficient even on codes that do not exhibit good spatial locality, but only when it can schedule the memory access using preload instructions. This makes the SR8000 extremely efficient on structured codes, where it is capable of achieving more than 50% of its $R_{peak}$ [2]. However, this architecture is also very sensitive to indirection, and, if it is not possible for the SR8000 to make efficient use of PVP, performance suffers. In spite of this, the SR8000 is able to achieve relatively high performance even on unstructured codes provided that the inner-most loop length is long enough to allow the use of certain prefetch optimizations. Codes using a JDS format can be well optimized by the compiler by issuing prefetch instructions on the indeces used to reference data. Preload instructions can then be scheduled for these prefetched indeces, with the result that good performance is again obtained. However, this process requires a long inner-most loop so that there is enough time to overcome the

<table>
<thead>
<tr>
<th>$R_{peak}$ (MFLOPS/s)</th>
<th>$B_m$ (Main Memory)</th>
<th>$B_m$ (Target Cache)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Hitachi SR8000</td>
<td>1500</td>
<td>0.333</td>
</tr>
<tr>
<td>Pentium Xeon</td>
<td>6800</td>
<td>0.098</td>
</tr>
<tr>
<td>SGI Altix</td>
<td>6400</td>
<td>0.125</td>
</tr>
</tbody>
</table>

Table 1. $R_{peak}$ and machine balance for selected architectures. In our analysis, we restrict ourselves to the largest on-chip cache level. These are indicated in the table.
initial latency of the prefetched indices.

The Hitachi SR8000 first appeared on the 13th top500.org list, ranked 4th. On the 15th list, the Hitachi SR8000 appeared in 5th and 6th places, with installations in Tokyo and München.

**Pentium Xeon** The Intel Nocona processor is the first of the extended memory (EM64T) implementing Intel's version of the x86-64 instruction set. This processor follows in the footsteps of previous offerings from Intel, with an extremely long pipeline (31 stages) and high clock speeds (currently up to 3.66 GHz). To offset the disadvantages of such a long pipeline, the Nocona has enhanced branch prediction capabilities with an increase in the size of the branch target buffer (BTB) (from 0.5 to 4 KB). This processor depends heavily on its dynamic scheduling capabilities to remove data hazards and avoid pipeline stalls. The model available to us for conducting test results did not have good machine balance, with a ratio of 0.0974 (3.4 GHz with 5.3 GB/s), although, even with the faster memory (6.4 GB/s) the ratio would only improve to 0.1176. The Nocona uses a 90nm process. Some of the additional on-chip space has been exploited to include a larger (1 MB) L2 cache that runs at the processor clock speed.

Because of good dynamic scheduling and a long pipeline, the Nocona is not overly sensitive to indirection and may be a good candidate for codes that exhibit this property. However, because of the imbalance in memory access to execution speed, it is not generally possible to achieve a high percentage of \( R_{peak} \). This means that the Nocona achieves good performance regardless of the code structure, but does not ever achieve extremely high performance as defined by a percentage of its \( R_{peak} \).

A partially Nocona based system is ranked 24th on the current top500.org list.

**SGI Altix** The SGI Altix 3700 Supercluster is based on the Intel Itanium 2 processor. This processor is the result of a collaborative effort (formed in 1995) by Intel and HP to develop a new architecture based on earlier attempts by HP to build very long instruction word (VLIW) machines. The result is a variant of VLIW called explicitly parallel instruction computing (EPIC), which explicitly indicates instruction level parallelism among instructions. Two important characteristics of this processor are that it uses static scheduling and that it has a very large (1.5–12 MB) low latency (7–12 cycles) L3 cache. Static scheduling makes the Itanium 2 very sensitive to indirection. However, for codes that provide a visible structure for the compiler, the Itanium 2 is able to achieve very high performance. The large L3 cache of this architecture
implies that codes that exhibit good spatial or temporal locality of data will likely achieve excellent performance. Such codes will be able to work primarily out of the L3 cache, thus, reducing the effects of memory latency. Another property of this processor that increases performance is the large floating-point register set (128 registers). This allows the processor to avoid register spills that would result in excessive load operations and increase the effects of latency.

A possible drawback of the Altix is that two processors share the same memory bus. The normal chipset used by SGI is capable of delivering 6.4 GB/s peak memory bandwidth. Since this bandwidth is shared by two processors, the Altix is not well balanced, and may exhibit poor performance on codes that are extremely memory-bandwidth limited. This deficiency is made up for, to some extent, by the large L3 cache. However, codes that randomly access memory will not be able to efficiently use the cache and are more likely to require fast main memory access to achieve good performance.

In spite of any drawbacks that the Altix may have, it is currently a widely used architecture for HPC applications (this is also true for the Itanium 2 in general). At the time of writing this thesis, the Columbia supercomputer at the NASA/Aimes Research Center (NAS) is ranked 4th on the 26th top500.org list. The Columbia is made up of 20 SGI Altix 3700 Superclusters.

### Prediction and Interpretation – The Balance Metric

To aid us in our initial interpretation of the results, we make use of the balance metric. In this case, we need to be precise about how the metric is counted for each implementation of the Gauß–Seidel algorithm. Therefore, the corresponding counts for the balance metric, for each data structure, are included in Table 2.

In each of the three implementations, the number of operations per update of an unknown is the same: $26N$ multiplications plus $26N$ additions for averaging the neighbors and adding the load vector, and $N$ multiplication for the quotient of the diagonal (we assume that the division has been pre-computed during setup), for a total of $53N$, where $N$ is the number of unknowns.

For the CRS implementation, we make the assumption that the unknown

<table>
<thead>
<tr>
<th>algorithm</th>
<th>loads (in double words)</th>
<th>operations</th>
<th>$\mathcal{B}_A$ (ratio)</th>
</tr>
</thead>
<tbody>
<tr>
<td>CRS</td>
<td>$(27 + 27 + \frac{N}{2} + 1 + 1)N = 69.5N$</td>
<td>$(26 + 26 + 1)N = 53N$</td>
<td>1.31</td>
</tr>
<tr>
<td>VCS</td>
<td>$(27 + 27 + 1 + 1)N = 56N$</td>
<td>$(26 + 26 + 1)N = 53N$</td>
<td>1.06</td>
</tr>
<tr>
<td>CCS</td>
<td>$(27 + 1 + 1)N = 29N$</td>
<td>$(26 + 26 + 1)N = 53N$</td>
<td>0.55</td>
</tr>
</tbody>
</table>

Table 2. Counts used in the balance metric for the various Gauß–Seidel implementations used in the current test case with $N$ unknowns.
being updated and all of its neighbors must be loaded from main memory each time they are accessed. Because of the indirection involved, this requires that 27 double words and 27 integers be loaded from memory, or 40.5 double words (integers are 4 bytes). The discretization weights require an additional 27 double words per unknown, as well as one double word for the load vector and we count one double word to save the updated value. Therefore, we assume that a total of 69.5 double words must be fetched from main memory to update $N$ unknowns.

The VCS implementation is similar to CRS without the indirection, so that we assume 56 double words to update $N$ unknowns. For the constant coefficient case, CCS, we assume that the discretization weights can be held in the registers, so we only require 29 double words per update of $N$ unknowns.

Table 3 gives both the predicted and measured performance for each data structure with $N = 5 \times 10^6$. The efficiency $E_{m/a}$ is given as the percentage of the predicted performance that is actually realized. We give an interpretation of these results for each platform used in the test:

<table>
<thead>
<tr>
<th></th>
<th>Hitachi SR8000</th>
<th>Pentium Xeon</th>
<th>SGI Altix</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>$B \times R_{peak}$ measured $E_{m/a}$</td>
<td>$B \times R_{peak}$ measured $E_{m/a}$</td>
<td>$B \times R_{peak}$ measured $E_{m/a}$</td>
</tr>
<tr>
<td>CRS</td>
<td>381 45 12%</td>
<td>508 496 98%</td>
<td>610 296 49%</td>
</tr>
<tr>
<td>VCS</td>
<td>473 300 63%</td>
<td>630 580 92%</td>
<td>757 1143 100%</td>
</tr>
<tr>
<td>CCS</td>
<td>913 884 97%</td>
<td>1217 1496 100%</td>
<td>1462 2810 100%</td>
</tr>
</tbody>
</table>

Table 3. Predicted versus measured performance in MFLOPS/s using the balance metric.

Interpretation (Table 3) – Hitachi SR8000: As expected, the Hitachi is extremely sensitive to indirection with only 12% efficiency for the CRS data structure. In this case, an overall performance of only 3.0% of $R_{peak}$ is achieved, giving this machine the worst relative performance for the CRS algorithm. In [12], Wellein et al. show that improved performance for unstructured grid representations can be obtained by using a jagged diagonals (JDS) storage scheme. The JDS data structure exposes longer straight line code sequences (basic blocks) that allow the compiler to schedule aggressive prefetch optimizations, which help to minimize the latency effects caused by indirection.

In contrast to the SGI Altix, the results for the VCS and CCS algorithms are accurately predicted by the balance metric. This is most likely due to the small L1 cache (128 KB) on the Hitachi. The Hitachi can minimize the effects of latency by issuing prefetch and preload instructions. However, because data cannot reside in the small cache for very long, the overall performance is still limited by the bandwidth to main memory. This is the primary assumption made by the balance metric. In light of this, the performance achieved by
the VCS algorithm is disappointing. Since the balance metric is designed to account for the memory limitations of the machine, the VCS algorithms should really achieve closer to 100% efficiency. Unfortunately, a full investigation of the causes of this anomaly are beyond the scope of the current work.

**Interpretation (Table 3) – Pentium Xeon.** The Xeon achieves results that are consistent with the predictions made by the balance metric for both the CRS and VCS data structures. Here, we see that this platform is much less sensitive to indirection, with 98% efficiency for the CRS algorithm, and an overall performance of 7.3% of $R_{peak}$. The overall performance is still only a fraction of the theoretical performance of the machine. However, in this case, the limitation is a consequence of the imbalance between CPU performance and memory bandwidth. The slightly better than expected performance of the CCS algorithm can be explained by the high-bandwidth, low-latency L1 cache of the Xeon.

**Interpretation (Table 3) – SGI Altix.** Again, as expected, this platform is quite sensitive to the indirection inherent to the CRS data structure and achieves a disappointing 49% efficiency, corresponding to an overall performance of only 4.62% of $R_{peak}$. However, for both the VCS and CCS data structures the performance exceeds our predictions (denoted by the bold percentages), indicating that the assumptions made in applying the balance metric are likely to be inaccurate. In particular, because of the large L3 cache on this platform, it is probable that data loaded into the cache will be reused, thus reducing the number of double words that need to be loaded per update. We present a revised model for this platform in the next section.

**Prediction and Interpretation – The Loads Per Miss Metric**

We now narrow our focus and consider a new model in order to better understand the results obtained for the SGI Altix. The assumptions made in constructing the balance metric represent the extreme case that a datum must be loaded from main memory every time it is accessed. To form the loads per miss metric, we consider the opposite extreme and assume that each time a datum is loaded into the cache, it will be used in a maximal way. Table 4 shows the revised counts based on this new assumption. In this case, there is a slightly different emphasis on the manner in which loads are counted. Instead of assuming that loads are from main memory into the cache, we now count loads from the cache into the register set. As an example, consider the load count for the CRS data structure. For each unknown being updated, we expect that it will be loaded from the L3 cache into a register once for each of the 27 positions it can take in the stencil. Additionally, the corresponding
Hierarchical Hybrid Grids

discretization weights must be loaded. Since the weights are different for each unknown, this results in 27 loads per unknown. Also, because of the indirect indexing involved in dereferencing each weight, we require 27 integer loads. Add to this the single load for the right hand side and we arrive at 82\(N\) loads for one Gauss–Seidel iteration over \(N\) unknowns.

To count the expected L3 cache misses per update for the CRS data structure, we first assume that we only experience one miss per unknown for loading the unknown being updated. Next, because each unknown has different weights, we assume 27 cache misses per unknown for loading the discretization weights. In addition, loading the discretization weights incurs 27 integer misses to handle the indirection. Finally, the load vector will incur one miss. Notice from the table that it is necessary to divide the predicted miss counts by the cache line size (128 Bytes or 16 double words). This is due to the fact that a miss will result in an entire cache being loaded into the cache. The integer count is divided by 32 to account for the integer’s smaller size (4 Bytes). Therefore, we expect \(85/32 N\) misses to update \(N\) unknowns. The theoretical loads per L3 miss for each data structure are shown in the fourth column of Table 4 with the corresponding cache hit rate in column five.

Table 5 shows the actual measured results obtained using the \texttt{lipfpm}\(^1\) tool. The correlation between the predicted and measured results is very close, meaning that our new model must give an accurate picture of what is actually happening during processing. This metric is essentially a measure of the temporal locality of each algorithm. By considering the ratio of loads to L3 misses along with the cache line size of 16 double words, we see that for the CRS and VCS algorithms each datum is used approximately two times every time that it is loaded into the cache. Since both of these algorithms have similar memory requirements and exhibit approximately the same amount of temporal locality\(^2\), we can conclude that the poor performance achieved using the CRS data structure is due almost entirely to the indirection required to access the discretization weights. This is not to say that the data volume associated

---

Table 4. Counts used in the loads per miss metric for the various Gauss–Seidel implementations used in the current test case with \(N\) unknowns.

<table>
<thead>
<tr>
<th>Algorithm</th>
<th>Loads</th>
<th>L3 Misses</th>
<th>L3 Misses</th>
<th>Hit Rate</th>
</tr>
</thead>
<tbody>
<tr>
<td>CRS</td>
<td>((27 + 27 + 27 + 1)N = 82N) (\left(\frac{1+27+1}{16} + \frac{27}{32}\right) N = \frac{85}{32} N) (\frac{82N}{32N} \sim 31)</td>
<td>(16) (\frac{32}{32}) (\sim 96.8%)</td>
<td>96.8%</td>
<td></td>
</tr>
<tr>
<td>VCS</td>
<td>((27 + 27 + 1)N = 55N) (\left(\frac{1+27+1}{16}\right) N = \frac{27}{16} N) (\frac{55N}{27N} \sim 30)</td>
<td>(32) (\frac{32}{32}) (\sim 96.7%)</td>
<td>96.7%</td>
<td></td>
</tr>
<tr>
<td>CCS</td>
<td>((27 + 1)N = 28N) (\left(\frac{1+1}{16}\right) N = \frac{1}{8} N) (\frac{28N}{1N} \sim 224)</td>
<td>(32) (\frac{32}{32}) (\sim 99.6%)</td>
<td>99.6%</td>
<td></td>
</tr>
</tbody>
</table>

\(^1\)http://www.sgi.com/products/software/histx

\(^2\)Recall that all three algorithms have the same spatial locality by design.
with the use of variable coefficients does not play a role. Comparing the CCS results to the results for the CRS and VCS data structures, we see that the CCS algorithm has much better temporal locality. The prediction of the loads per miss metric is that each datum will be used 14 times each time it is loaded into the cache. The measured result is actually slightly better, showing that a value loaded into the cache is used approximately 16 times. This drastically reduces the strain placed on the memory subsystem and clarifies why the CCS algorithm has more than a two fold increase in performance over the VCS algorithm.

### Flexibility vs. Efficiency

Another way to interpret the results of this test is to consider the flexibility that each algorithm offers and the price that must be paid for that flexibility. The CRS format can be used to represent many linear systems of interest and is the most flexible data structure under consideration. Consequently, the CRS data structure consistently achieves the worst overall performance. As we have shown, this is due both to indirection and the data volume required to load the discretization weights for each unknown in the system. Somewhat more restrictive is the VCS data structure that is limited to use with structured grid geometries. This data structure provides the compiler with a visible data access pattern, and can, therefore, be optimized using static scheduling. This is an advantage. However, the performance gain still only accounts for approximately 33% of the gap between the CRS and CCS algorithms. The rest of the gap is accounted for by the much larger data volume needed in accessing the variable coefficient weights. The least flexible and most efficient data structure is the stencil-based CCS algorithm. Here, we have the best possible scenario with lower data volume and a clearly visible access pattern. However, many problems cannot be adequately represented using this type of data structure. As stated previously, the goal of the HHG framework is to combine the flexibility of unstructured grid representations where they are needed with structured patches that are constant-coefficient to obtain a method that can be applied to a wide range of problems while still achieving a high level of performance.
HHG Performance Results

In order to achieve high performance when solving an extremely large problem in parallel, it is first necessary to achieve high performance on a small problem in serial. To quote Nietzsche, "..., one cannot fly into flying.". Thus, the goals of our serial results are to show that we can obtain very high performance using the HHG data structures and that certain aspects of these data structures are scalable in a local sense. This idea will be clarified in what follows. We narrow our consideration to include only the Intel Itanium 2. This architecture gains significant benefits from the HHG data structures and is also a platform for which we were able to obtain access to a very large machine. The other platforms considered in the last section should also see performance improvements with the HHG data structures. However, this is a topic for future work.

Following the presentation of our serial results, we perform a scalability study of the HHG data structures used to solve a problem with $1.7 \times 10^{10}$ unknowns.

Serial Results

To begin our investigation of the serial performance of the HHG data structures, we consider results for performing a constant-coefficient, row-wise, red-black Gauß–Seidel smoothing algorithm on a single tetrahedral element at various levels of refinement. This test case represents the best possible situation of a purely structured grid with a constant stencil, and serves in determining the absolute maximum performance that can be obtained by the HHG data structures. Figure 3 shows these results. Here, we see that the efficiency of the algorithm depends on the refinement level. The most likely cause of this behavior is that the inner-most loop length of the smoothing algorithm increases with each level of refinement. On the lower refinement levels it is likely that the loop overhead has a more pronounced effect on performance, where as, on the higher refinement levels there are longer periods of unbroken computation, resulting in better performance. As can be seen from the figure, the performance gain from having longer for loops only lasts until the ninth level of refinement. This is a concrete example of an algorithm changing its behavior with increasing problem size. Measured results show that at ten levels of refinement the number of loads per L3 cache miss falls below 224 (from Table 4), which is in line with our analysis using the LPMM. This clearly indicates that the drop in performance is due to the fact that all three planes of the computational stencil no longer stay in cache for the full lifetime of their usefulness, thus decreasing performance.

Another important aspect of serial performance when using the HHG data structures is that of serial scalability. We could also refer to this aspect as local scalability. By both of these terms we mean that the algorithm must be scalable.
Refinement Level

Figure 3. Serial smoothing performance per refinement level on a single tetrahedral input element for the SGI Altix. The maximum inner-most loop length with respect to the refinement level \( l \) is \( 2^l - 2 \). So, for example, at \( l = 9 \) the maximum inner-most loop length is 510. Because a tetrahedron is a simplex, the two inner-most loop lengths decrease during the course of the update.

as we add to the number of elements in the input grid. This is an important consideration for the HHG framework due to the grid decomposition discussed in the overview. Because of the grid decomposition it is necessary to perform local copying of ghost buffers. Adding to the number of elements in the input grid adds to the amount of copying that must be performed. This local copying could conceivably hurt performance. Here, we show that it does not.

Considering Figure 4, we see that adding to the number of input elements does not seem to have any negative effects on performance. It is probable that the algorithm scales well in this sense because adding to the number of input elements also adds to the amount of computational work that is done. Therefore, the local copying operations are balanced by added computation, and the extra work does not have a significant effect.

Parallel Scalability

The goals of our parallel test results are to show that extremely large problems can be solved with good efficiency using the HHG data structures. In reality, it is likely that we have solved the largest finite element system to date with performance rarely achieved by most scientific codes. In this section, we present the results for solving a Poisson problem with Dirichlet boundary conditions
Hierarchical Hybrid Grids

on the unit cube that has been partitioned using an unstructured tetrahedral grid. The choice of this domain was convenient for performing our scalability tests because we could easily generate a sequence of grids that doubled in size at each grid in the sequence. We must emphasize that the grids used for our tests are purely unstructured and that no structural feature of the input grids was exploited in obtaining these results.

Figure 5 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 Falls, Wisconsin. Each test used in the study was run with seven levels of refinement. Table 6 shows grid details for each test, including: The number of unknowns, the aggregate performance in GFLOPS/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 GFLOPS/s and time to solution show that the implementation is very efficient. At the time of writing this paper, the Leibniz-Rechenzentrenms in Garching, Germany is in the process of installing an SGI Altix 4700 cluster with 4096 CPUs, each with 4GB RAM. On this new machine, we predict that we will be able to solve a problem with $6.8 \times 10^{10}$ unknowns at approximately 3.5 TFLOPS/s.
Conclusion

In any scientific endeavor it is important not only to produce great results, but also to be able to explain them. In many cases the explanation requires extremely careful analysis. This is certainly true of scientific computing in general and particularly true of the techniques presented in this paper. The use of metrics to try and model the underlying interactions that take place during the execution of complex scientific codes has proved to be quite useful. The application of this type of analysis allows us to both remove barriers to code performance and to give adequate explanations of why certain codes perform as they do. In a time of increased competition for research funding this is certainly a useful tool. In this instance, one that has aided in the attainment of new milestones.

References