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

**Lehrstuhl für Informatik 10 (Systemsimulation)**

# Scalability on All Levels for Ultra-Large Scale Finite Element Calculations

Tobias Gradl, Christoph Freundl, Ulrich Rüde

Technical Report 07-5

# Scalability on All Levels for Ultra-Large Scale Finite Element Calculations

Tobias Gradl, Christoph Freundl, and Ulrich Rüde

Chair for System Simulation, University Erlangen-Nuremberg, Cauerstr. 6, 91058 Erlangen, Germany

May 7, 2007

**Abstract**

The development of software for computational science problems is a difficult task, especially when it comes to large problems that must be solved with multi-processor systems. The term "scalability" has been established to indicate the quality of parallel software. This paper extends the notion of "scalability" to the whole life-cycle of numerical software, from the choice of the algorithms to the code design and development and its execution on very large parallel systems consisting of thousands of processors. We use Finite Element problems to explore our holistic notion of scalability and present two packages we have developed for solving Finite Element problems in the Peta-Scale area, capable of solving problems with in excess of $10^{11}$ unknowns on an SGI Altix system with 9728 processor cores.

## 1 Introduction

Current high end computing is characterized by a rapidly increasing complexity on all levels: more complex applications require more complex software that is difficult to design due to a rapid growth of parallelism into the hundreds of thousands of processor cores and beyond. Additionally, to achieve good performance requires more and more that programmers tune the software for increasingly complex processor architectures. Such architectures may e. g. have specialized vector processing capabilities and many levels of memory hierarchy that may require complex cache blocking schemes, and in the near future we will see multi-core processors adding another set of problems.

When techniques and methods accommodate growth well, the high end computing community traditionally labels them with the attribute of *scalability.* The scalability of architectures and systems has thus been a primary driving goal (this is not to say "the holy grail") of high end computing ever since parallel systems have become available.

However, in the light of the seemingly explosive growth of complexity on all levels, we believe that the notion of scalability must be extended and must be seen from a much broader perspective. Scalability cannot be limited to the systems and architectures perspective alone, but should include aspects of the scientific computing pipeline, such as software complexity and algorithmic scalability, or the seemingly simple but often critical problem of achieving reasonable performance of an application for the node architecture of a particular parallel system. The ultimate goal is to accomplish the best *time-to-solution* with a given set of resources in hardware, software, and man power. The latter is critical, since we cannot neglect the human effort needed to develop and debug software as a considerable contribution to the time-to-solution. On the other hand, some critical low level computational kernels may more than ever require careful design and tuning since this effort will be easily amortized over the lifespan of the code.

These needs of high end computing are on the verge to gain much wider relevance with the advent of multi- and many-core systems — a trend that will directly drive many of these problems into the mainstream of the computing business. At this stage it even seems that a good portion of the wider computing community is being caught unaware of what could be the most fundamental shift in computing of this decade, and it is well possible that we are heading into a major shakeup leaving those behind who do not address these issues early and aggressively enough.

With this paper we present a prototype holistic approach to the design of scalable parallel Finite Element (FE) software for the petascale area. To illustrate our approach we have picked typical techniques and methods that we will describe and discuss to illustrate a possible route towards *holistic scalability*.

The paper is organized top-down. To make the use and the design of complex high end computing software more scalable, we first discuss our use of object oriented techniques and Expression Templates within the framework of the *Parallel Expression Templates for Partial Differential Equations* (ParExPDE) system. Next, algorithmic scalability for FE problems inevitably requires a multiscale approach, since these are the only known algorithms that scale linearly with problem size. Subsequently, we will discuss our techniques to achieve (classical) parallel scalability using message passing on massively parallel systems for these algorithms.

Then we turn to the next lower level and describe the techniques that are necessary to make FE software and the corresponding computational kernels perform well on current high performance compute nodes with a special emphasis on the Itanium architecture. This is motivated by the 10 000 processor core SGI-Altix system at LRZ of the Bavarian Academy of Sciences in Garching [14] that we use for large scale applications and our multiprocessor scalability studies. The *Hierarchical Hybrid Grids* (HHG) paradigm of [3] is used to achieve the flexibility to accommodate a wide range of applications, multilevel algorithms, but without sacrificing parallel scalability or node scalability. The conclusions section contains results using our software, algorithms, and architectures. The largest problem solved to date is a tetrahedral mesh Finite Element problem leading to a sparse linear system with $3 \times 10^{11}$ unknowns that is solved by our software in little over a minute and thus beats the world record by Bergen et. al. from 2005 [4] by more than an order of magnitude.

## 2 Scalability for Software Development

This section deals with the issue of controlling the code complexity for computational applications. In some way, a programmer has to translate the mathematical formulation of an algorithm into a suitable programming language. The question arises how the complexity and therefore also the readability and maintainability of the program code depends on the complexity of the algorithm per se.

Though the main focus of this paper is Finite Element calculations, we will illustrate the basic concepts of Expression Templates for vectors which are ultimately also required for FE software. ParExPDE is then extending these concepts, as described below.

The computation of a vector sum $\overrightarrow{z} = a\overrightarrow{x} + \overrightarrow{y}$ poses at first sight no problem since we might link the BLAS daxpy routine or one of its parallel variants.

However, already the combination of two daxpy expressions to form a more complex one, e. g. $\overrightarrow{z} = a(b\overrightarrow{x} + \overrightarrow{y}) + \overrightarrow{w}$, inherently creates a problem. If BLAS routines should be used exclusively, we have two alternative ways to proceed, either

- call daxpy($b$, $\overrightarrow{x}$, $\overrightarrow{y}$) to compute the intermediate result $\overrightarrow{v}$ followed by the call to daxpy($a$, $\overrightarrow{v}$, $\overrightarrow{w}$) to compute the final result $\overrightarrow{z}$, or
- call daxpy($a$, $\overrightarrow{y}$, $\overrightarrow{w}$) to compute the intermediate result $\overrightarrow{v}$ followed by calling daxpy($a \cdot b$, $\overrightarrow{x}$, $\overrightarrow{v}$).

Either way, the complexity of the program increases and the (memory, cache) efficiency potentially suffers, since an intermediate vector is used. Also, the resulting code obscures the mathematical notion of what is to be accomplished.

Numerically oriented languages, such as Fortran 90, permit to write the expression as is. Ultimately, however, functionality that is rigidly built into the language is not sufficient, since we will inevitably come to the point where extensions are required. C++ is a programming language which provides the functionality to extend the language. A typical C++ programmer would use a (non-numerical) vector class anyway, and would just have to overload the `+`, `*`, and the assignment operators in order to be able to write `z = a * (b * x + y) + w`. This would achieve our goal of keeping the source complexity low, but unfortunately the efficiency will suffer, since evaluating each operator independently and storing the intermediate results in temporary vectors creates a significant overhead on all current architectures.

The computation of intermediate results followed by the copying of the temporary vectors is in fact unnecessary if we find a way to treat the expression *as a whole* instead of evaluating it part by part. In C++ this can be achieved by so-called *Expression Templates* (ET) [20] as they are used in several well-known libraries like Boost[1] or, for applications in the field of scientific computing, MET[2], FreePOOMA[3], PETE[4], and Blitz++[5].

We are following a similar route with the *Parallel Expression Templates for Partial Differential Equations* (ParExPDE) library. The current implementation is limited to block-structured (unstructured regularly refined hexahedral) 3D grids using Finite Elements. The use of Expression Templates provides an easy-to-use interface, introducing *differential operators* so that problems and solvers can be designed easily. Here we do not only use Expression Templates for the implementation of differential operators and their discretization with Finite Elements, but we extend the evaluation mechanism in order to hide the parallelization (cf. [15]). A user of the library will therefore be able to program without having to deal with communication or synchronization, since this is completely (and efficiently) handled by the high level operators. The computation of the FE local stiffness matrices is additionally handled by the library Colsamm of [9] that uses ET. The library design fulfills the "easy-to-use" requirement stated above without compromising efficiency even for very large scale problems; see the results in section 7.

The ET approach also has disadvantages. Many of the negative aspects are essentially caused by the fact that ET uses C++ in a way it was not originally designed for. It is far from trivial to design ET libraries that are efficient. This is also due to compiler limitations, and the efficiency of the generated code may vary much between compilers and systems. Additionally, compile times and error messages can be a nuisance. From our experience, we find that a programming language should provide extension mechanisms similar to C++ operator overloading. The ET mechanism opens interesting possibilities to implement these efficiently. However, it is less clear that this will be the ultimate solution, and there is a great need for better languages to implement scalable parallel software.

## 3 Scalability of Algorithms

We choose a scalar elliptic PDE as a prototype problem for comparing the scalability of different solvers. Mathematically, the problem is of the form

$$-\nabla \cdot a\nabla u + \beta u = f \quad \text{in } \Omega, \tag{1}$$

subject to suitable boundary conditions, where $\Omega \subset \mathbb{R}^3$ is a three dimensional domain, $a$ and $f$ are scalar fields, and $\beta$ is a nonnegative constant. Eq. (1) arises in many applications, such as in heat conduction or the pressure correction in each time step of incompressible flow simulations, the global potential in particle simulations with (e. g. Coulomb type) long range interactions, in each time step of implicit methods for time dependent parabolic or hyperbolic equations, and many more. When discretized by FE, these problems lead to large sparse systems that are symmetric and positive definite. Because of these numerous applications, this problem has been studied intensively and is suitable as a benchmark problem (see [1] for another FE-based benchmark). We will restrict our discussion to cases where $a$ is constant in each element with only mild jumps between elements.

It is well known that *multigrid algorithms* [19] constitute a class of asymptotically optimal solvers for this type of problems. Especially in our context, the phrase *asymptotically optimal* translates directly into *scalable*, since it means that a doubling of the problem size can be expected to lead to a doubling of the numerical cost. This is particularly important when dealing with the ultra-large problems that we intend to demonstrate.

It is possibly less well known that multigrid algorithms are also among the most efficient in terms of the absolute number of operations. The exact figures depend on the details of the discretization and the algorithm itself, but e. g. for the two-dimensional variant of (1) (and $a = 1, \beta = 0$), discretized by finite differences, multigrid algorithms have been constructed that compute a solution

[1]`http://www.boost.org/`
[2]`http://met.sourceforge.net/`
[3]`http://www.nongnu.org/freepooma/`
[4]`http://acts.nersc.gov/pete/`
[5]`http://www.oonumerics.org/blitz/`

within less than 30 operations per unknown, independent of the problem size. For 3D Finite Elements, no hard analysis of this is known to us, but we will demonstrate that our solvers do in fact achieve fast execution in absolute run time.

Multigrid algorithms have a recursive structure that needs to be considered carefully in a parallel implementation [10]. The system of successively coarser grids with less and less unknowns requires a number of communication steps per cycle that increases logarithmically with the problem size and that will always include short messages for the coarser levels. This also leads to the idle processor problem when processing coarse levels of the hierarchy and potentially creates overhead due to the latencies associated with many short messages. These observations have possibly discouraged potential applications of parallel multigrid, but in our opinion this is unjustified, since all alternative methods suffer from equivalent difficulties.

The need for global communication is inherent in any elliptic problem and there is *no* hope to avoid it. The question can only be how this unavoidable communication can be organized most efficiently and conveniently. In particular, domain decomposition algorithms might be considered as a potentially advantageous alternative. At a first glance, domain decomposition algorithms only require inter-processor communication between neighboring subdomains of the single fine scale mesh and therefore may seem to be more suited for parallel implementations, since they do not require the exchange of small messages. However, a closer analysis reveals that parallel domain decomposition methods will either suffer from a severe deterioration of convergence when many subdomains are being used or they need a coarse grid accelerator. In the case that the convergence rate is not independent of the problem size, a converged solution does require an increasing number of iterations and thus also the number of communication steps increases with problem size. Note that even a logarithmic increase of the number of iterations with problem size (as is typical for many domain decomposition algorithms) is enough to turn the argument asymptotically in favor of multigrid algorithms. Alternatively, the need for global communication is implicitly contained in the coarse grid solver that is necessary to avoid the deterioration of convergence in the case of many subdomains. The details are difficult to pin down because there are many variants on how such a coarse grid problem is being solved, but to our knowledge it is not possible to design a domain decomposition solver such that it has either less communication volume or less communication steps than a straightforward and well implemented parallel multigrid method.

Summarizing, domain decomposition algorithms are neither inherently more computationally efficient, nor do they save parallel communication when compared to a parallel implementation of multigrid for the same problem. Naturally, domain decomposition methods may have advantages in practice, when it is e. g. possible to re-use existing software as subdomain solvers, or when the design of a multigrid method is complicated by the very nature of the problem. For our prototype problem (1), however, we have the luxury of designing a solver from scratch, and we believe that a straightforward multigrid implementation is the most promising approach. We also believe that our performance results in section 7 demonstrate this point quite well. To our knowledge there is currently no competing algorithm that can be used to solve FE problems of similar size in comparable compute time.

## 4 Scalability for Large Multi-Processor Systems

Multi-processor systems provide either shared or distributed memory access to the processors. Whatever memory structure is chosen, in all current large scale systems the access is non-uniform and implemented using a network: part of the memory is local to a specific processor and can be accessed with high bandwidth and low latency; the rest has to be accessed over a network with comparably low bandwidth and high latency. Parallel algorithms will require more or less communication over the network. Our FE prototype application is such that the ratio of communication to computation is given by a surface-to-volume ratio. Asymptotically, therefore, computation will dominate over communication, but for 3D applications and since our algorithm is computationally so efficient, having suitably large subdomain problems is critical to obtain good performance.

Therefore we must keep the number of remote memory accesses (or messages) as small as possible and the amount of data transferred in a single access as high as possible to avoid the startup latencies. In partial conflict with that we must enable the asynchronous operation of the processes, especially when going to large numbers. This is achieved by overlapping computation

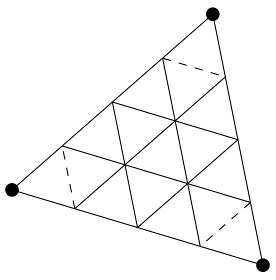

Figure 1: HHG communication and ignored dependencies

and communication.

The conflict arises typically when several data items have to be communicated between two processes. In order to avoid small messages and their startup cost, all the data should be collected in a single large message. But then the sender can only start sending the message after all required values have been computed, and thus there may be no useful operations left to do between starting the send and waiting for its completion. In such a case the algorithm designer has to find a good balance between message size and the number of messages.

Decoupling the processes by asynchronous communication has such a high value that it often pays to deviate from the original algorithm to a certain extent. An example can be found in the *Hierarchical Hybrid Grids* (HHG) framework [3] that is used for the benchmarks in section 7. HHG performs very memory-efficient Finite Element simulations by using a compromise between structured and unstructured meshes and a multigrid solver [4, 2]. A coarse input mesh is split into the grid primitives vertices, edges, faces, and volumes that are then refined in a structured way. The multigrid solver uses a Gauß-Seidel smoother that traverses the grid primitives in the above order: first, all vertices are smoothed, then all edges, and so on. When a Gauß-Seidel sweep traverses over the grid points, it always uses, in contrast to Jacobi, the most recent values of neighboring grid points. This makes communication during an iteration sweep necessary, as neighboring grid points might belong to different processes. However, if really all this communication is performed, it might completely dominate the run time. The current HHG implementation chooses to ignore a few dependencies of the Gauß-Seidel iteration, giving the smoother the characteristics of a Jacobi iteration at the affected points. Figure 1 illustrates the dependencies that are ignored in HHG's current communication pattern. During the smoothing of all unknowns of a type (e. g. edges) no communication is performed. This means that adjacent primitives of the same type (e. g. two edges in the corner of a triangle) do not exchange data directly; connections between them (the dashed lines in the figure) are thus smoothed in a Jacobi style. Numerically, this leads to only a slight deterioration of the convergence rates, but the gain in execution time more than outweighs this effect.

## 5 Scalability in Node Performance

In the previous section we have shown techniques for distributed programming in a multi-processor environment. It is common to use the *scaleup* as a key value for measuring the parallel performance of a program, and this is often referred to as scalability in its ordinary sense. However, this neglects the performance on a single node and may thus lead to wrong conclusions.

Speedup and scaleup are only relative measures, and, consequently, they can be manipulated by choosing the reference basis. In other words: Good speedup or scaleup is trivial to achieve if we choose a slow serial base algorithm or a slow implementation. If, for example a code has 50 % parallel efficiency by spending one second computing and one second communicating, we can easily make it 90 % efficient by choosing an implementation that slows the compute time down to 9 seconds.

This is obvious, but nevertheless we believe that the focus on parallel speedup and scaleup as relative measures has given many computational scientists biased incentives to optimize primarily parallel efficiency, and possibly devote less attention to the efficiency of the code in itself. Parallel speedup and scaleup are important, but they are not the only aspects of scalability in our sense. In

| Problem size | Fortran Mflop/s | Best Assembler Mflop/s |
|---|---|---|
| $33^3$ | 1414 | 2942 |
| $65^3$ | 1089 | 2650 |
| $129^3$ | 1125 | 2941 |
| $257^3$ | 757 | 3032 |
| $513^3$ | 616 | 2773 |

Table 1: Performance comparison Fortran and assembler code

this section, we will therefore turn our attention to techniques for achieving the best possible node performance.

Basic techniques that will help to achieve node performance include inlining and basic loop transformations that can be applied automatically by highly optimizing compilers (cf. [5]). Unfortunately, not all issues can be resolved in that way.

A quite common reason for bad node performance is that a program's data access patterns do not match well with the deep memory hierarchy of current architectures. *Cache optimization techniques* can be used, which include both data layout transformations, such as array padding or array merging, and data access transformations, such as loop interchange and loop blocking. For details on these techniques in general, we refer to to [13, 8, 11] and for the application to iterative methods and multigrid algorithms in particular to [7, 16, 17, 12, 6].

Significantly more than by just using the general guidelines can be achieved when codes can (or must) be optimized for a particular architecture. The HHG framework uses this in a mild form by using mixed language programming, since it was found that on some machines significant performance gains could be achieved by implementing the compute kernels in Fortran 77 instead of C++.

Optimization can be driven to the extreme by using assembler coding. For the Itanium IA64 architecture and thus for the processor of the SGI Altix system described in section 6, it is important to exploit the instruction level parallelism of the EPIC architecture and also to hide the latencies for the in-order execution model by aggressive software pipelining.

In a recent Master thesis, Stürmer [18] has found that compilers currently fail to do these optimizations for three dimensional grid structures when advanced cache blocking techniques are employed that lead to complicated nested loop structures and additionally only moderately long loops. In this case, it is necessary to directly code in EPIC machine language to exploit the performance potential of the machine and to obtain speedups by up to four for a Gauß-Seidel smoother as compared to an equivalent (straightforward) Fortran code (cf. tab. 1). Clearly, these speedups are significant enough that it will be attractive to integrate these compute kernels into future versions of the ParExPDE and HHG frameworks, even if portability issues become more difficult. It is of course conceivable that some of these low level optimizations can be done by improved compilers. This is where they should be done ideally. However, this has been a hope for the past few decades, and despite all progress, the problem seems to be as severe as ever.

Additionally, we wish to point out that some optimization techniques depend on a high level choice of the data structures and algorithms. In particular both the HHG and ParExPDE frameworks were designed to avoid classical sparse matrix data structures and thus to enable a whole host of specific optimization techniques that can then either be done manually or automatically by a compiler. However, there is currently little hope that a compiler or any other tool could automatically convert an unstructured sparse matrix code into a structured grid code, when this is found to improve efficiency. Therefore, we believe that holistic scalability must also include an awareness of the developers of low level machine specific optimizations.

## 6 Scalable Architectures

Not all parallel applications put the same requirements on the quality of the computer architecture. Both codes presented in this paper use the multigrid algorithm, an algorithm that is computationally very efficient, and because of that does require fast communication. Compared to LINPACK, which is characterized by $O(n^3)$ computational versus $O(n^2)$ memory cost, multigrid has linear complexity

| # Processors | # Unknowns ($\times 10^6$) | Average time per $V$-cycle (sec) | Time to solution ($\lvert\lvert r\rvert\rvert < 10^{-6} \cdot \lvert\lvert r_0\rvert\rvert$) |
|---:|---:|---:|:---:|
| 4 | 66.5 | 19.62 | 19.9 + 58.8 |
| 8 | 133.1 | 19.55 | 19.7 + 58.7 |
| 16 | 266.3 | 19.60 | 20.0 + 58.8 |
| 32 | 532.6 | 19.82 | 20.0 + 59.4 |
| 64 | 1 065.3 | 19.67 | 20.1 + 59.0 |
| 128 | 2 130.7 | 19.70 | 20.5 + 59.1 |
| 255 | 4 244.8 | 20.20 | 21.6 + 60.6 |
| 510 | 8 489.6 | 20.63 | 24.0 + 62.0 |
| 1020 | 16 979.3 | 22.04 | 64.1 + 66.3 |
| 2040 | 33 958.6 | 25.57 | 57.5 + 79.9 |

Table 2: Scaleup results for ParExPDE. With a convergence rate of about 0.03, 4 $V$-cycles are necessary to reduce the starting residual by a factor of $10^{-6}$. The runtime is shown as the sum of the runtime of the first $V$-cycle (which includes the setup of the stiffness matrices) and the remaining three $V$-cycles.

in the amount of data. Therefore, multigrid depends on a high memory and network bandwith much more than LINPACK. Additionally, multigrid does not even sweep over all unknowns all the time. On each coarser level, the number of unknowns is reduced by a factor of 8 (for 3D grids). Naturally, there is less computation to be done between the communication points on coarse levels and, therefore, multigrid algorithms profit from computers that have a relatively large amount of memory for each core: a given problem will have to be distributed over less cores than otherwise, and less communication will be necessary. The *HLRB 2* at the *Leibniz-Rechenzentrum* of the Bavarian Academy of Sciences in Garching, Germany [14] is based on the SGI Altix 4700 architecture. In phase 1 until March 2007, the system comprised 4096 single core (Intel Itanium 2 Madison) nodes and was listed as number 18 on the TOP500 list. Installation phase 2, available since April 2007 and currently still under testing, is equipped with 9728 Intel Itanium 2 Montecito dual core processors, 9 MB level 3 cache per core, and in total 38 TB of main memory (4 GB per core). On 9170 cores of HLRB 2 we were able to compute a Finite Element simulation with $3 \times 10^{11}$ unknowns using 31 TB of main memory.

This machine had been selected based on a benchmark suite that emphasizes performance for fluid flow and other solvers for partial differential equations and is therefore also well suited for our FE multigrid solvers. For comparison, on a BlueGene/L, the computer ranked first on the TOP500 list of November 2006, the same simulation would have to use more than 64 000 (dual core) nodes, because this computer only provides 512 MB of main memory per node.

## 7 Conclusions

The library ParExPDE of section 2 addresses advanced software techniques for high end computing, in particular Expression Templates. It has been designed for multilevel algorithms and for the execution on multi-processor systems. Scaleup results for the HLRB 2 installation phase 1 are shown in table 2. They indicate a good scalability for up to 500 processors; for larger number of processors the efficiency deteriorates slightly so that we expect that an improved communication will yield even better results, see also the discussion in section 4. In particular the assembly of the stiffness matrix (whose timing is included in the first $V$-cycle) still shows some anomalies in the scaling behavior that are currently being explored.

Note, however, that this package achieves excellent multigrid efficiency with a convergence factor of 0.03 per iteration; it therefore reduces the initial residual by six orders of magnitude in merely 4 iterations and thus solves a problem with 33 billion unknowns on 2040 processors in about two minutes compute time. Note that this could be further improved by additional algorithmic techniques such as a nested iteration (full multigrid) [19].

The HHG program introduced in section 4 is designed to solve as large as possible Finite Element problems as fast as possible. Like ParExPDE, it uses multigrid algorithms and is parallelized with

| # Processors | # Unknowns ($\times 10^6$) | Average time per $V$-cycle (sec) | | Time to solution ($\|\|r\|\| < 10^{-6} \cdot \|\|r_0\|\|$) |
|---|---|---|---|---|
| | | Phase 1 | Phase 2 | Phase 1 |
| 4 | 134.2 | 3.16 | 6.38 * | 37.9 |
| 8 | 268.4 | 3.27 | 6.67 * | 39.3 |
| 16 | 536.9 | 3.35 | 6.75 * | 40.3 |
| 32 | 1 073.7 | 3.38 | 6.80 * | 40.6 |
| 64 | 2 147.5 | 3.53 | 4.93 | 42.3 |
| 128 | 4 295.0 | 3.60 | 7.06 * | 43.2 |
| 252 | 8 455.7 | 3.87 | 7.39 * | 46.4 |
| 504 | 16 911.4 | 3.96 | 5.44 | 47.6 |
| 2040 | 68 451.0 | 4.92 | 5.60 | 59.0 |
| 3825 | 128 345.7 | 6.90 | | 82.8 |
| 4080 | 136 902.1 | | 5.68 | |
| 6120 | 205 353.1 | | 6.33 | |
| 8152 | 273 535.7 | | 7.43 * | |
| 9170 | 307 694.1 | | 7.75 * | |

Table 3: Scaleup results for HHG. With a convergence rate of 0.3, 12 $V$-cycles are necessary to reduce the starting residual by a factor of $10^{-6}$. The entries marked with * correspond to runs on (or including) *high density* partitions with reduced memory bandwidth per core.

MPI. Table 3 shows HHG's scaleup results on HLRB 2, installation phase 1. The code meets its design goals with computing record-breaking numbers of unknowns. Compared to ParExPDE its convergence is slightly slower and thus more $V$-cycles are needed to reduce theresidual by six orders of magnitude. This is, however, compensated by the superior run times per $V$-cycle. Up to about 500 processes it also shows very good scaling; after that, the efficiency deteriorates slightly due to the synchronization effects described in section 4. Again, there is room for further tuning and optimizing the code; this is explored within an ongoing project.

Table 3 contains both older results for installation phase 1 of HLRB 2 with up to 3825 processors and brand new results obtained during the test of phase 2 on up to 9170 processor cores. The phase 2 configuration of HLRB 2 includes both low density partitions where two cores share a memory bus and high density partitions where four cores share one. Clearly, the effective bandwidth for the phase 2 configuration is lower than for phase 1 already in the low density partitions, and is further reduced in the high density partitions. The results in table 3 are labelled accordingly and show that this immediately results in a reduced performance. Since the machine is not yet in normal user operation, these results must still be considered preliminary. Nevertheless, the 9170 core run has delivered the solution of an FE-system with 0.3 Tera unknowns ($= 3.07 \times 10^{11}$) in about 90 seconds for 12 multigrid $V$-cycles.

## References

[1] M.F. Adams, H.H. Bayraktar, T.M. Keaveny, and P. Papadopoulos. Ultrascalable implicit finite element analyses in solid mechanics with over a half a billion degrees of freedom. In *ACM/IEEE Proceedings of SC2004: High Performance Networking and Computing*, 2004.

[2] B. Bergen, T. Gradl, F. Hülsemann, and U. Rüde. A massively parallel multigrid method for finite elements. *Computing in Science & Engineering*, 8:56–62, November 2006.

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

[4] B. Bergen, F. Hülsemann, and U. Rüde. Is $1.7 \cdot 10^{10}$ unknowns the largest finite element system that can be solved today? In *SC '05: Proceedings of the 2005 ACM/IEEE conference on Supercomputing*, page 5, Washington, DC, USA, 2005. IEEE Computer Society.

[5] S. Carr, K.S. McKinley, and C.-W. Tseng. Compiler optimizations for improving data locality. In *ASPLOS-VI: Proceedings of the sixth international conference on Architectural support for programming languages and operating systems*, pages 252–262, New York, NY, USA, 1994. ACM Press.

[6] I. Christadler, M. Kowarschik, and U. Rüde. Towards cache-optimized multigrid using patch-adaptive relaxation. In J. Dongarra, K. Madsen, and J. Wasniewski, editors, *PARA 2004 Proceedings*, volume 3732 of *Lecture Notes in Computer Science (LNCS)*, pages 901–910. Springer, December 2005.

[7] C.C. Douglas, J. Hu, M. Kowarschik, U. Rüde, and C. Weiß. Cache optimization for structured and unstructured grid multigrid. *Electronic Transactions on Numerical Analysis (ETNA)*, 10:21–40, February 2000.

[8] S. Goedecker and A. Hoisie. *Performance Optimization of Numerically Intensive Codes*. SIAM, 2001.

[9] J. Härdtlein and C. Pflaum. Efficient and user-friendly computation of local stiffness matrices. In F. Hülsemann, M. Kowarschik, and U. Rüde, editors, *Simulationstechnique, 18th Symposium in Erlangen, September 2005*, Frontiers in Simulation, pages 748–753, 2005. ISBN 3-936150-41-9.

[10] F. Hülsemann, M. Kowarschik, M. Mohr, and U. Rüde. Parallel geometric multigrid. In A.M. Bruaset and A. Tveito, editors, *Numerical Solution of Partial Differential Equations on Parallel Computers*, volume 51 of *LNCSE*, chapter 5, pages 165–208. pub-SV, 2005.

[11] M. Kowarschik. *Data Locality Optimizations for Iterative Numerical Algorithms and Cellular Automata on Hierarchical Memory Architectures*. Number 13 in Advances in Simulation. SCS Publishing House, 2004. ISBN 3-936150-39-7.

[12] M. Kowarschik, C. Weiß, and U. Rüde. Data layout optimizations for variable coefficient multigrid. In P. Sloot, C. Tan, J. Dongarra, and A. Hoekstra, editors, *Proc. of the 2002 Int. Conf. on Computational Science (ICCS 2002), Part III*, volume 2331 of *Lecture Notes in Computer Science (LNCS)*, pages 642–651, Amsterdam, The Netherlands, April 2002. Springer.

[13] M.D. Lam, E.E. Rothberg, and M.E. Wolf. The cache performance and optimizations of blocked algorithms. In *ASPLOS-IV: Proceedings of the fourth international conference on Architectural support for programming languages and operating systems*, pages 63–74, New York, NY, USA, 1991. ACM Press.

[14] Leibniz Rechenzentrum München. Höchstleistungsrechner in Bayern (HLRB 2). Internet. `http://www.lrz-muenchen.de/services/compute/hlrb/`.

[15] C. Pflaum and R.D. Falgout. Automatic parallelization with expression templates. Technical Report UCRL-JC-146179, Lawrence Livermore National Laboratory, 2001.

[16] G. Rivera and C.-W. Tseng. Tiling optimizations for 3D scientific computations. In *Supercomputing '00: Proceedings of the 2000 ACM/IEEE conference on Supercomputing (CDROM)*, page 32, Washington, DC, USA, 2000. IEEE Computer Society.

[17] S. Sellappa and S. Chatterjee. Cache-efficient multigrid algorithms. In *ICCS '01: Proceedings of the International Conference on Computational Sciences-Part I*, pages 107–116, London, UK, 2001. Springer-Verlag.

[18] M. Stürmer. Optimierung des Mehrgitteralgorithmus auf IA 64 Architekturen. Diplomarbeit, Lehrstuhl für Informatik 10 (Systemsimulation), Friedrich-Alexander-Universität Erlangen-Nürnberg, May 2006.

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

[20] T.L. Veldhuizen. Expression templates. *C++ Report*, 7(5):26–31, June 1995. Reprinted in C++ Gems, ed. Stanley Lippman.