Iterative Algorithms on High Performance Architectures

Ulrich Rüde[1]*

Institut für Mathematik, Universität Augsburg

Abstract. The numerical solution of partial differential equations leads to large, sparse systems of equations with up a several millions of unknowns. Fast iterative algorithms for the solution of these systems are typically based on the multilevel principle. Unfortunately, some of the commonly used programming techniques lead to a high overhead on many advanced computer architectures. The two main sources of this performance degradation are the effects of non-uniform data access combined with indirect addressing that prohibits the use of instruction-level parallelism. This, however, is essential to exploit many modern CPU designs. The second, and possibly more fundamental problem arises from hierarchical memory architectures with several layers of caches. Their effective use requires programs with data access locality. Unfortunately, iterative solvers are typically implemented by using global sweeps over the whole data set, and thus their performance is essentially limited by the speed of the memory system. These problems are addressed in the patch-adaptive multigrid method, a recently developed experimental software system implementing an adaptive multigrid method based on a locally uniform mesh data structure.

1 Introduction

Theoretical analysis as well as practical experience show that typical linear systems with N unknowns arising in the numerical solution of elliptic partial differential equations (PDE) can be solved in $100N$ floating point operations by an iterative multigrid algorithm.

On the other hand, current microprocessors are capable of performing more than 10^9 (1GFlop) floating point operations per second and beyond 2 Gflops peak performance are expected by the end of 1997.

Consequently, we may predict that the solution of a system with 10^7 unknowns, corresponding to the discretization of a scalar PDE on a grid of $100 \times 100 \times 1000$ points should be possible in less than 1 second on a suitably equipped single processor workstation or PC. The current[1] base price for such a machine is below US$ 5,000 plus approximately. US$ 6,000 for a memory upgrade to 512 MB that is required for problems of the above size.

* Partly supported by grant RU 422/7-1 of Deutsche Forschungsgemeinschaft

[1] This is written in May 1997

Unfortunately, the above projections are far from the performance observed in practice. A number of multigrid based programs is available, e.g. from MGNet,[2] but all current codes seem to miss the predicted performance by a factor between 100 and 10000. Clearly, this gap between (naive) theory and real life performance needs to be analyzed.

Of course much can already be said about the predicted 100 operations per unknown to solve the problem. For many complicated PDE, the efficiency that is promised by the multigrid principle [3] has not (yet) been fully obtained. However, there are also important practical problems where this efficiency has been demonstrated. For simple problems, like potential problems in two dimensions (2D), a even significantly better rate of 30 operations per unknown has been obtained and even for PDE in three dimensions (3D) in irregularly shaped domains and varying coefficients, a work estimate of 100 operations per unknown is quite realistic. But the detailed discussion of these algorithms, their analysis and their extensions to more general problems not the topic of this paper.

An evaluation of several multigrid codes shows that they often use algorithms that are designed to treat more general problems and therefore are not optimal for the simple ones. Many codes also include costly extra functionality e.g. by providing automatic mesh refinement, error estimates, and by solving the discrete systems with higher than necessary accuracy. All this leads to a numerical complexity which may significantly exceed the 100 operations per unknown, but generally, this is by far not enough to explain why some of the codes are 10000 times slower than predicted.

In this paper we will show that to a large extent the performance degradation is caused by a mismatch between the algorithms and the architectures they are executed on. The basic reason is quite simple: On almost all current machines iterative PDE solver codes are not limited by the number of operations, but by the memory access costs.

Memory access as a fundamental bottleneck of the von-Neumann architecture has been discussed at least 20 years ago, see e.g. Backus' Turing award lecture [1]. In practical numerical processing, however, this has become a serious issue only during the past 5 years.

The consequences are potentially dramatic. At this stage it is still open whether we just experience a temporary misdevelopment in computer architecture or a fundamental limitation in one of our core computing paradigms, as Backus has suggested. The traditional measure of cost for an algorithm — not only in numerical applications — is the number of operations executed. But if the above is true, this has been become irrelevant to predict the real life performance of a program. Thus ultimately we may have to develop a new, more complicated complexity theory based on memory operations. Here in this paper we will merely discuss some of the aspects arising before this background from the perspective of iterative linear system solvers.

In Section 2 we will briefly introduce the basic algorithms and data structures used in iterative solvers and Section 3 will complement this with a brief descrip-

[2] MGNet can be found at URL http://www.ccs.uky.edu/mgnet/mgnet.html

tion of current computer architectures as seen from the program developers perspective. After a brief digression to consequences for the parallel programming on multiprocessors, we will then discuss two possible strategies for restructuring iterative PDE solvers to better exploit current machine architectures.

2 Iterative methods for the solution of PDE

The simulation of complex scientific or technical processes often relies on mathematical models using PDE. Such equations describe *local* relations of physical quantities, like the diffusion of heat, the motion of a fluid, or the deformation of a solid. The solution consists in constructing from the local relations the *global* ones.

The development of a numerical simulation starts with the *discretization*. Here the physical continuum is replaced by a discrete sets of points, and thus the differential equation is approximated by a large system of equations relating the unknowns associated with neighboring locations. The coefficient matrices are thus large with up a several millions of unknowns but they are sparsely populated. Typically the number of nonzero entries in each row (or column) is bounded by a small constant.

Unfortunately, this sparsity is partly lost when standard elimination methods are applied to solve the systems. Fill-in destroys the sparsity and leads to a (in 2D, and more so in 3D) dramatic memory and processing overhead. Thus, iterative methods methods are generally preferred for these problems.

Elementary iterative methods are of relaxation type. A *sweep* of the classical Gauss-Seidel (GS) method consists in looping through all unknowns and replacing its value such that the corresponding row if the system is satisfied. This method is guaranteed to converge e.g. when the system is symmetric positive definite (SPD), as it is in turn the case when the original PDE is linear, second order, self-adjoint, elliptic. For this paper we limit the discussion to this problem type, since it already includes many important applications. Furthermore it is a subproblem required in many of the more complicated problems.

Unfortunately, even for SPD problems, the Gauss-Seidel algorithm is slow. When the discrete mesh has $O(n)$ unknowns in each spatial dimension, and the linear system thus has $O(n^2)$ or $O(n^3)$ unknowns in 2D and 3D, respectively, then the plain Gauss-Seidel method requires $O(n^2)$ sweeps, before the global solution is obtained with sufficient accuracy. The overall complexity of the solution process is therefore $O(n^4)$ operations in 2D and $O(n^5)$ operations in 3D. Therefore, methods to speed up the convergence of iterative methods have been explored intensively. The method of over-relaxation (SOR) and the family of conjugate gradient (CG) methods are among the most popular ones.

Instead of discussing the details of these methods, we point out the fundamental bottleneck in all iterative methods. In the original physical problem, each location effects each other one. This is reflected in the algebra. Though the system matrix itself is sparse, its inverse is dense. It contains the any-to-any dependency of the unknowns and it is thus much too expensive to be computed explicitly.

In a relaxation method only local relations are executed and thus information propagates essentially only between neighboring unknowns in each sweep. Here neighboring refers to the spatial closeness in the physical domain.

Thus all iterative processes are limited by how many sweeps are necessary to disperse information globally, when each sweep can propagate it just between adjacent points. In this sense SOR achieves an optimum, requiring $O(n)$ sweeps (instead of $O(n^2)$, as for plain Gauss-Seidel). The complexity to solve the system with SOR is therefore $O(n^3)$ for 2D and $O(n^4)$ for 3D problems.

CG-type algorithms employ a global variable that can be interpreted as a means for the global dissemination of information. However, this single variable is not sufficient, and thus a plain CG algorithm does only improve the efficiency to the same asymptotic complexity that can also be obtained by SOR. Better CG algorithms rely on so-called preconditioners, the best of which are of hierarchical type and are thus closely related to the multigrid algorithms discussed below.

For the problem class under consideration, the best iterative methods achieve a complexity proportional to a single sweep through the data. To our knowledge, however, all these methods are of multilevel type which directly, or indirectly rely on coarser meshes covering the physical domain. These coarser meshes provide a means for faster and more efficient exchange of information.

The basic *multigrid* method does so by using a hierarchical sequence of successively coarser meshes, typically with twice the distance between mesh points as in the finer one. Domain decomposition may also be asymptotically optimal, but this is accomplished also by using of coarser meshes. To focus the study we will limit the further discussion to multigrid-like algorithms.

Algorithmically, multigrid consists of a relaxation phase, applying one or more sweeps of Gauss-Seidel or a similar process. After performing this local exchange of information on the current grid, a step of global communication is required. Global communication is done recursively by transferring the data to the next coarser grid. There again a few sweeps of relaxation are used to exchange data locally, relative to the characteristic mesh width on this grid. This is repeated recursively on all coarser grids. Performing a few sweeps of relaxation on each grid provides the data exchange on the distance scale induced by the characteristic mesh width. Finally, from each coarser grid, a correction for the next finer one is computed to update the values there. After this coarse grid correction another relaxation phase may be applied. This algorithm is called a *V-cycle*, and it is the basic algorithmic structure of a multigrid method. For a more complete introduction to multigrid, see Briggs [4], and for a monograph with an in-depth discussion see Hackbusch [7]. Further information, including an extensive bibliography is available electronically from *MGNet* [5].

Additional issues must be considered in many practical applications. Often, the solution varies drastically throughout the domain, and an effective solution requires an appropriately adapted mesh resolution. Therefore, many of the more recent multilevel implementations use flexible mesh structures and means for adaptive mesh refinement. Unfortunately, this makes the the data structures considerably more complex.

Note that using 2D or 3D arrays does not mean that the physical domain is limited to rectangles or bricks when appropriate mappings from computational to physical domain are used. Such domains are often called *logically rectangular*. Though more general than plain rectangles, they are still quite limited, especially when mesh adaptation is necessary.

While a basic multigrid code could use plain arrays to store the grid data, typical adaptive codes use *unstructured* grids. This, however, requires to store data in lists or trees and thus requires the use of pointers. Besides storing the neighbor relations of a single mesh, the data structure must also provide means to perform the data transfer from fine to coarse grids and vice versa. This requires more pointers, as shown in Fig. 1.

A structured grid may e.g. exploit that each node has a fixed number of neighbors, corresponding to a fixed number of entries in the matrix rows. On an unstructured grid, this number is variable, and the neighbors need to be accessed via pointers instead of simple index operations. In practice, therefore

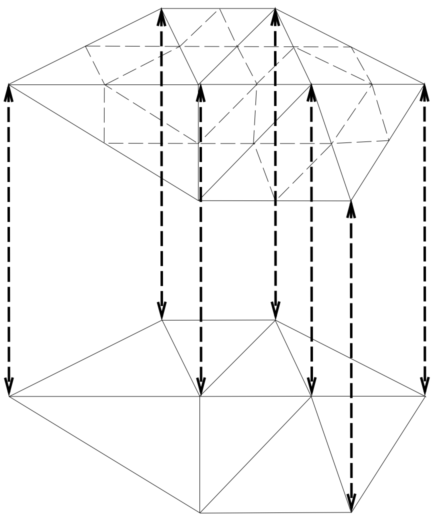


Fig. 1. Hierarchy of two unstructured meshes

unstructured grid methods show a significant memory overhead and also a serious performance degradation that we will discuss in detail further below.

Traditionally, multigrid as well as other iterative methods are programmed in terms of *global sweeps*. The elementary operations, relaxation, information transfer between grids, and further auxiliary operations are typically implemented in separate subroutines. In each of these routines all the nodes of a mesh must be accessed: In the case of structured meshes (consisting of arrays) the node location can be computed by index calculations. For unstructured meshes, pointers (index arrays in FORTRAN based implementations) must be used. Thus the working set is typically a full grid, and since we are mostly interested in large scale problems, we must expect the finest grids to use a significant fraction of the main memory capacity.

With this in mind, we now take a look at the computer architecture that we will eventually have to put our programs on. The discussion refers primarily to workstation (or PC) designs and thus also to the architecture of most parallel computer nodes.

3 CPU and memory architectures

The continuing evolution of microprocessors brings a quick increase in processing speed that is estimated to be about 80% annually. This agrees with the past 10-20 years of technological development, and though it is unlikely that this trend will continue forever, an extrapolation for the next 10 years seems to be quite realistic.

The main driving factor in the technological development is the reduction of structure sizes on the chips. A reduction in size in the past has permitted proportionally increases clock rates, but this correlation is expected to slow down because other physical barriers become effective. Present commercially available chips, like the Alpha 21164 reach clock rates of up to 600 MHz.

The number of transistor functions on a chip will continue to increase rapidly. Presently, 10 Million transistors on a chip are typical, and this number of logic functions can only be exploited by using parallelism within the chip. All CPUs employ pipelining, and increasingly, they are additionally superscalar.

Current floating point units are typically 4 way superscalar and are pipelined with something like 10 pipeline stages. In this sense, a modern floating point unit is internally 40 way parallel. The effective use of such an architecture requires suitable compilers, but also suitable algorithms.

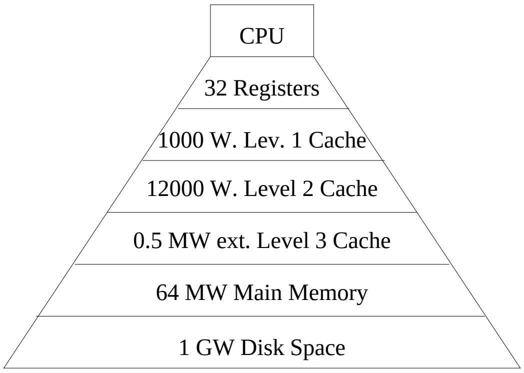
While the importance of the having good compilers and optimizers is generally accepted, the need to develop algorithms specifically for the new CPUs is controversial: Not many application developers like the idea of loosing the classical paradigm of sequential execution of instructions even in a single CPU.

However, it is impressive how much speed up can be obtained by restructuring even simple algorithms, like matrix multiplication, see e.g. [2]. We have observed performance improvements from a naive code to an optimized one of factors like 10 to 20, without having obtained the optimum yet. Note that these program modifications are not machine dependent (e.g. using assembly language coding), but are source code level optimizations that are effective on a wide range of different machines.

Thus we need to become aware of a fundamental fact: Modern CPUs are only superficially sequential processors. To use them effectively means to do parallel programming. Of course this parallelism is usually not programmed explicitly. Nevertheless, only a programmer aware of the parallel nature of a CPU can write codes that can be well optimized by the compiler. This optimization can effectively be understood as an automatic parallelization, and the programmer's task is to construct algorithms with sufficient internal parallelism.

The second important component of a computer is the memory. Unfortunately, the speed of memory has not kept up with the CPU evolution. The speed increase for DRAM chips is estimated at only 5% annually. Therefore, many modern computer designs involve caches: fast (and therefore expensive) memory of limited capacity to hold copies of data from main memory that is frequently accessed. Much of the performance of modern computers essentially depends on caches and thus the locality of memory access of a given program.

In Fig. 2 we display a typical memory architecture that includes the registers and disk as additional levels in the hierarchy. The figure also displays the



Level	Capacity	Throughput	Latency
Register	512 B	34 GB/s	3 ns
Cache 1	8 KB	9.6 GB/s	3 ns
Cache 2	96 KB	4.8 GB/s	10 ns
Cache 3	4 MB	1 GB/s	20 ns
Main Mem	14 GB	300 MB/s	200 ns

Fig. 2. Memory architecture for Digital Alpha-Server 8400 with a single Alpha 21164 CPU at 300 MHz

performance characteristics for each level in the hierarchy. Note that the bandwidth and latency are better on the higher levels but that the capacity becomes progressively smaller.

It is interesting to reflect on some of the features of the above architecture. Note that the CPU has two levels of on-chip cache and one may ask why this design has been favored over a design with a single cache of combined capacity. The reason is that even on-chip it is not possible to address much beyond the 8KB limit in a single cycle of a fast processor. The Alpha 21264 chip announced for the end of 97 will return to a single layer of on-chip cache, but will pay the price of having no memory (except the registers) that can be accessed in a single

CPU cycle. It is believed that all designs will have to face a similar trade-off, once they reach the clock rates achieved by the Alpha processors [6].

Note that in the Digital 8400 architecture only the registers provide sufficient bandwidth to support the full superscalar capabilities of the processor. Even the level 1 cache can merely support sufficient data for the floating point units in a vector update operation of daxpy type, when all index calculations can be done in registers. The level 2 cache, still on chip, can only support bandwidth for 50% of the peak performance. The main memory bandwidth supports just about 3% efficiency.

The next processor generation (the Alpha 21264) has been designed to drastically improve the memory bandwidths, and will e.g. reach a maximum of 5.3 GB/sec throughput for the off-chip cache, but at the same time the new processor will at least quadruple the peak performance (this will depend on the clock rates achieved). Thus the basic relations will not improve that much despite the conscious efforts to better balance the memory and processor performance by the CPU architects. For more detail see [6].

Unfortunately the memory bottleneck is in sharp contrast to the requirements of iterative methods. Due to the conventional implementation in the form of global sweeps, all these algorithms are limited by the performance of the memory level that is large enough to hold the data. In most cases, this will be the main memory.

Each sweep typically consists of simple read-modify-update process involving only a few operations to be performed, but the data must be brought through the memory hierarchy. No cache re-use can occur, and in fact many of the machines would be faster in this application, when they didn't have caches.

In operations like the Gauss-Seidel relaxation this is not completely true, since neighboring points are also accessed, and some re-use naturally occurs when these data are re-encountered further down the sweep.

Finally we take a look at the use of array versus linked list data structures. For structured meshes the memory addresses can be computed by simple integer operations within CPU registers. Superscalar CPUs may thus overlap index calculations with floating point work. Either through hand optimization or automatically by suitable optimizing compilers, the memory access can also be be overlapped with the execution, so that memory latencies may be hidden and the bandwidth becomes the limiting factor.

For unstructured meshes this is different. First note that 64 bit pointers use as much space as a double precision number, and thus manipulating pointers requires as much bandwidth as the processing of double precision data.

To clearly demonstrate the fundamental problem, let us consider the extreme case of an array which is stored as a linked list with each atom consisting of a double precision number and the pointer to the next element in the list. This naturally doubles the memory requirements compared to an array data structure, but for processing the overhead becomes even worse. The next element of a linked list can naturally only be accessed, when the pointer is available. However, this pointer must be loaded itself from slow memory. Thus it is the *latency* instead of the *bandwidth* that limits the performance. If we assume that within 200ns (the main memory latency of the Dec Server 8400) a pair consisting of pointer and a double precision number can be brought into the CPU, we will be able to process the list at a rate of one element per 200 ns. This is equivalent to a bandwidth of 80 MByte/sec, 40 MB/sec of which are floating point numbers.

We might have expected an overhead of the linked list compared to an array of a factor of two, since it needs twice the amount of memory transfer. However, the argument above shows that we switch from a bandwidth limited operation to one that is latency limited. Thus the already low 300 MB/sec main memory bandwidth is reduced by another factor of 7.5. In this extreme case, the CPU may not even reach 0.5% of its peak performance.

The above argument may be too simple, since the situation in a real code may be substantially different. For example, we have assumed that data is never found in the cache, but often successive list elements may be stored in the same cache line, so that caches may help the performance. Additionally, it is possible that other data besides the single list must be accessed, and this may be overlapped.

However, there are also situations, where even a further degradation of the performance is possible. For example on many machines the loop performance depends essentially on unrolling, whether done by hand, or implicitly by the compiler However, loops terminating on a data dependent condition, like the encounter of a nil-pointer cannot be unrolled effectively.

In any case, it should have become clear that the extensive use of pointers, especially when the referenced objects are as lightweight as single real numbers, is potentially a performance killer. Ultimately the reason is that through the use of pointers like in a linked list, loop structures may result that cannot exploit the internal parallelism of the CPU.

Before discussing a few possible improvements, we present a little digression into what all this this may mean for “real” parallel processing

4 A look into parallel processing

As outlined above, the speed-up obtainable from using the instruction level parallelism is in a range similar to that of using a multiprocessor with a moderate number of processors. To find the most cost-effective way for speeding up a program, one therefore needs to compare single processor optimizations versus parallelization.

Additionally it should be noted that CPU-internal parallel programming (in the form of writing efficient single processor codes) and multiprocessor parallel programming are conflicting. It is not true that a bad program that is parallelized for a parallel system can always be optimized later after the program has been parallelized.

To illustrate this, assume that we start, parallelizing a non-optimized program for a multiprocessor and that we have achieved a reasonable speed up on a given machine configuration In particular assume that a certain program run uses 10 min execution time, 2 of which are spent in waiting for communication.

Thus we could claim having reached an acceptable 80% parallel efficiency. If now each node were accelerated by a factor of 8, due to optimization and better exploiting each CPU, the net compute time would decrease to 1 min. Since the communication time stays the same, the parallel efficiency has now dropped to $1/3 = 33\%$.

Let us present this argument from the other side: For successful parallel computing it is helpful to parallelize the worst sequential program one can find. Unfortunately, one sometimes gets the impression that this is not quite uncommon in current parallel programming practice.

After this somewhat polemic remark, we return to the main topic of the paper and try to outline some things that can be done to improve the performance of iterative solvers (before they are conventionally parallelized).

5 Fusing Sweeps

In the previous sections we have identified the slowness of main memory in combination of the program structure relying on global sweeps as a serious performance bottleneck of iterative methods. Here we suggest a first modification to improve the efficiency.

Global sweeps are not explicitly required by the algorithms, they are merely a programming convenience. In iterative methods like GS or SOR, the same sweep is repeated many times. Therefore it is possible to combine several of the sweeps into one, with the goal of re-using cached data. Of course this must be done carefully not to violate any data dependencies. In the simplest case of GS performed a 2D structured grid, the principle and its benefit are illustrated in Fig. 3

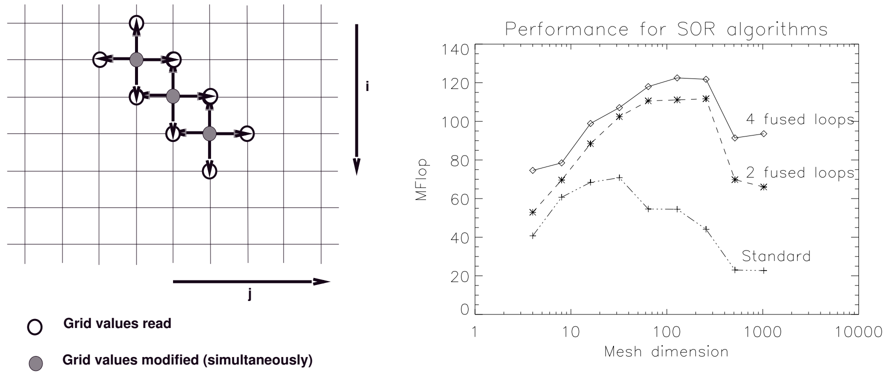


Fig. 3. Loop fusing and corresponding performance rates measured on SGI Power Indigo (R8000/75Mhz)

We assume a so-called 5-point difference discretization, where each point on a structured grid is related to its four orthogonal neighbors. The grid is accessed from top to bottom (outer loop) and left to right (inner loop). If the first sweep has progressed to the second element of the second line, it is already possible to start the second sweep on the first line on the first element. Similarly, when the first sweep reaches 3rd line, and the second sweep the 2nd line, then a third sweep can be started. Thus we may perform any number up to n sweeps simultaneously along the diagonals of the mesh. Of course this same scheme has been known as a parallelization (and vectorization) strategy for GS and SOR algorithms. Here we use it to improve the cache re-use, assuming a few lines of the grid are small enough to fit into the cache and thus as an instruction-level parallelization strategy.

The right half of Fig. 3 displays benchmark results for this *loop fusing* as a function of the grid size. Note that now even for large grids with dimension of 1000, a performance of close to 100 MFlops is maintained, about a factor of 4 more than obtained with simple global sweeps.

Here two effects play together: Once the cached data can be better re-used, but also the loop structure has an effect similar to unrolling which permits better optimization (that is instruction-level parallelism) when two or four iterations are fused together.

The same idea can be extended to multigrid algorithms. This complicates matters significantly, since with the above technique we may speed up the relaxation on each grid, but ultimately, we may have to fuse together more loops, including such performing other operations like the transfer of data to a coarser grid.

A subtle tradeoff is relevant in this context. Though multigrid will generally benefit from doing more relaxations, it becomes inefficient if too many relaxations are applied in each multigrid V-cycle (relative to the number of coarse grid corrections). When the fusing strategy makes additional relaxation sweeps comparatively cheap, the traditional balance may shift. Conventional multigrid is optimal when between two and four relaxation sweeps are performed for each coarse grid correction on each level. This aspect is an interesting topic for a mathematical analysis of multigrid algorithms.

In any case, the number of relaxations will be limited to a fairly small number, and it is therefore attractive to fuse the relaxations together with the other operations, including the transfer of the data to the coarser grid, and even possibly some of the coarse grid processes. Unfortunately, fusing different types of loops creates an awkward programming problem, and we are presently searching for suitable tools to help with the code restructuring.

6 Patch-adaptive multigrid

Fusing sweeps as described above may also be useful for unstructured grids, though we do not have any evaluation how effective fusing is, when the data is accessed through pointers.

However, pointers may be such a bad thing for performance, that instead of working in this direction, we have started to explore ideas how an adaptive multigrid method could be designed without using unstructured grids.

The above discussion has shown that the basic trouble is the case of dynamic (pointer-) data when used for too *light-weight* objects. Having one pointer per each floating point number creates a large overhead. The situation automatically improves, if we assume that the basic problem is more complex, and the physical model requires several unknowns at each grid location. Then the objects in the list become automatically somewhat heavier in weight, and the relative overhead decreases.

We propose to exploit this tendency by grouping several grid points together by arranging them uniformly so that they become patches of fixed size. Thus we lose some of the flexibility but we can still adapt the meshes and by arranging the patches suitably. The typical situation of a patch-adaptive refinement is illustrated in Fig. 4, where we assume that the solution requires meshes that are refined towards the reentrant corner. A prototype implementation using this strategy has been described in [8,9].

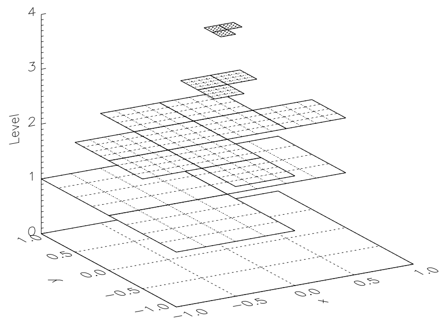


Fig. 4. Patch-adaptive multilevel structure

In the figure, the patches are of size 4×4, in practice we use sizes of 16×16 or 32×32. We also mention that adaptivity on the basis of using structured meshes has been used in the multigrid community for a long time and there are too many variants to cite them all, so that we refer to the multigrid bibliography [5].

The patch structure can be naturally used to exploit a cached architecture. Instead of performing a number k of global sweeps, we may perform one sweep through the global structure with k sweeps through each patch. Since the patch is small enough to fit in the cache, we can hope to improve the performance. Note, however, that the algorithm is *not* equivalent to the one with global sweeps. We first benchmark the execution times, giving the results in Table 1. The results

steps	time [sec]	speed (nodes/sec)
1	0.40	657 922
2	0.48	1 096 537
3	0.59	1 338 147
4	0.68	1 548 052
5	0.78	1 686 980
6	0.90	1 754 460

Table 1. Speed of patch-wise relaxation

show clearly that with this modification, each additional relaxation step has only a fraction of the cost of the first one.

On the other hand, since the algorithm is not identical to the global sweep method, it is interesting, how quickly a multigrid method deteriorates when it uses the patch-wise relaxation as compared with global sweeps. Table 2 displays results for both alternatives. The errors displayed are those remaining after a

	global		patch-wise	
Steps	[sec]	L_2-err	[sec]	L_2-err
1	0.75	$1.47\ 10^{-3}$	0.75	$1.47\ 10^{-3}$
2	1.61	$1.02\ 10^{-3}$	1.04	$1.02\ 10^{-3}$
3	2.35	$8.27\ 10^{-4}$	1.26	$8.28\ 10^{-4}$
4	3.28	$7.10\ 10^{-4}$	1.48	$7.22\ 10^{-4}$
5	4.16	$6.31\ 10^{-4}$	1.64	$6.57\ 10^{-4}$

Table 2. Convergence and speed of global vs. patch-wise sweeps

completed multigrid cycle employing a relaxation with the corresponding number of sweeps. Note that the patch-wise relaxation really shows an degradation in the error, when the number of relaxation steps is above 4, but that it leads to a significantly faster method.

Also note that as explained above, multigrid does not profit from increasing the number of relaxation steps too much. This is also visible from the post-cycle errors measured in Table 2. But with the additional performance provided by cache-oriented relaxation the question how many relaxation steps should be employed is again open and needs further consideration.

Finally, we present a benchmark, comparing the patch-adaptive method with unstructured adaptive multigrid methods. The graphs in Fig. 5 display the accuracy vs. the computing time on a logarithmic scale for different programs. The slope indicates the asymptotic efficiency of the method. Two versions of the patched method are included. The unbroken line corresponds to the method when it is not using adaptivity. It is therefore equivalent to using uniform grids. This suffers from using too few points where refinement is needed and too many points where the solution is uncritical. Therefore, the asymptotic complexity is

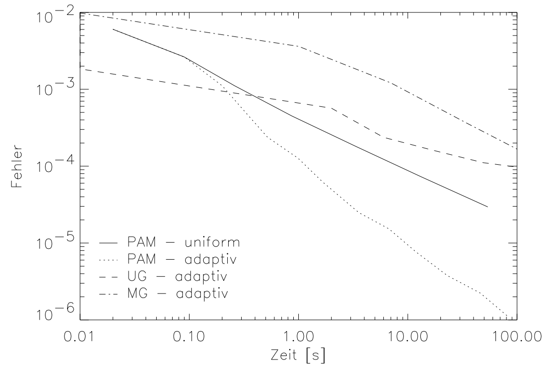


Fig. 5. Performance (accuracy vs. compute time) of different multigrid algorithms.

not optimal. When the patches are used adaptively, as shown by the dotted line, a much better complexity is obtained.

The two dashed lines show the performance of two different unstructured adaptive methods for the same problem. Clearly they are not competitive with the patch approach, providing almost a factor 100 less accuracy for the the same CPU time. Surprisingly, they even perform worse than the non-adaptive program, though this uses many more unknowns. The patch program is still better than the unstructured grid programs, since the handling of each unknown is so much more efficient.

7 Conclusions

We have presented some aspects on the performance of iterative PDE solvers. Our study has shown that current codes exhibit a surprising amount of inefficiency, and how this relates to parallel programming issues. We have outlined some strategies to improve the situation, but it should be clear that if current trends in computer architecture continue, this is just a first step on the a probably long and painful path towards how we should eventually design numerical methods.

References

1. J. Backus. Can programming be liberated from the von Neumann style? A functional style and its algebra of programs. *Communications of the ACM*, 21(8), 1978.
2. T. Bonk and U. Rüde. Performance analysis and optimization of numerically intensive programs. SFB Bericht 342/26/92 A, Institut für Informatik, TU München, November 1992.
3. A. Brandt, 1997. Talk given at the 8th Copper Mountain Cconference on Multigrid Methods, April 6–11, 1997.

4. W. L. Briggs. *A Multigrid Tutorial.* SIAM, Philadelphia, 1987.
5. C. C. Douglas and M. B. Douglas. MGNet Bibliography. In mgnet/bib/mgnet.bib, on anonymous ftp server casper.cs.yale.edu, Yale University, Department of Computer Science, New Haven, CT. Last modified on December 28, 1996.
6. L. Gwennap. Digital 21264 sets new standard. *Microprocessor Report*, 10(14), 1996. also available from URL http://www.chipanalyst.com/report/articles/21264/21264.html.
7. W. Hackbusch. *Multigrid Methods and Applications.* Springer Verlag, Berlin, 1985.
8. H. Lötzbeyer. Objektorientierte parallele adaptive Mehrgitterverfahren auf semistrukturierten Gittern. Diplomarbeit, Insitut für Informatik, TU München, 1996.
9. H. Lötzbeyer and U. Rüde. Patch-adaptive multilevel iteration, 1996. Accccepted for publication in BIT, also available from mgnet at http://casper.cs.yale.edu/mgnet/www/mgnet.html.