Paper C

Implementation Issues for High Performance CFD

Markus Nordén[*], *Malik Silva*[†], *Sverker Holmgren*[*], *Michael Thuné*[*], *Richard Wait*[*]

Abstract

Many important phenomena in nature are modeled by partial differential equations (PDEs). Such equations describe flow of air, water, oil and other substances and propagation of electromagnetic and sound waves in different media. Also, PDEs arise in many other application areas ranging from chemistry to economy. In the present paper, the focus is on computational fluid dynamics, CFD. Issues regarding efficient implementation of CFD programs on modern parallel computers are considered.

We study a computational kernel for a solver for compressible flow problems, e.g. arising when computing the air flow around an aircraft, using a state-of-the-art algorithm of Jameson-type. The solver uses a finite volume discretization combined with a Gauss-Seidel-Newton technique to solve the Euler equations. This PDE is non-linear, the solution is vector valued and the computations are carried out on a three dimensional curvilinear grid. Furthermore, modern numerical techniques such as multigrid and flux-splitting schemes are used in the solution process.

We have experimented with different optimization techniques in order to make the implementation efficient. The program uses the red-black ordering for the Gauss-Seidel smoother, and preliminary results for a parallel version show that it scales well on a shared-memory machine.

1 Introduction

Many important phenomena in nature are modeled by partial differential equations (PDEs). Such equations describe flow of air, water, oil and

[*]Department of Scientific Computing, Information Technology, Uppsala University, Sweden.

[†]Department of Statistics and Computer Science, Colombo, Sri Lanka and Department of Scientific Computing, Information Technology, Uppsala University, Sweden.

other substances and propagation of electromagnetic and sound waves in different media. Also, PDEs arise in many other application areas ranging from chemistry to economy. In the present paper, the focus is on computational fluid dynamics, CFD.

For realistic problem settings, the numerical solution of the PDEs requires large-scale computations. Powerful parallel computers have to be used, and even on these it is important that the computations are carried out efficiently. The present study addresses the performance issues involved in the implementation of a CFD code.

In general, there are four issues to take into account when implementing a PDE solver:

1. How to design an efficient and sufficiently accurate numerical scheme, so that the cost — in terms of arithmetic operations and memory requirements — for obtaining the numerical solution to the PDE problem is kept low

2. How to parallelize the scheme, so as to distribute the arithmetic work evenly among the processors, and to avoid unnecessary communication/synchronization between them

3. How to use data structures, and to restructure the code in order to make efficient use of the memory hierarchy

4. How to restructure the code to optimize CPU performance

Optimization at any one of these levels might introduce changes at other levels.

A necessary requirement for the code to be effective for realistic application problems is that the numerical scheme is as efficient as possible. Here, we address this issue by studying a state-of-the-art CFD solver. More precisely, we consider a computational kernel for a solver for compressible flow problems, arising, e.g., when computing the air flow around an aircraft. The kernel uses the algorithms described by Jameson [7] to solve the Euler equations to steady state. This system of PDEs is nonlinear, the solution is vector valued and the computations are carried out on a three dimensional curvilinear grid. In the kernel, a finite volume discretization is combined with a flux-splitting scheme, and a symmetric Gauss-Seidel-Newton technique. Furthermore, multigrid [6] is used for convergence acceleration. Similar techniques are used in many of today's industrial aerodynamics simulation codes, but the algorithms presented in [7] may prove to represent further improvements.

The parallel scheme is usually derived by first implementing an efficient serial method, and then possibly introducing minor changes to allow for/improve parallelization. For the solver studied in this paper, the symmetric Gauss-Seidel method is parallelized by introducing a red-black ordering of the computational cells. A prototype parallel version of our program has been implemented using OpenMP, and timings show that it scales well.

On modern computers, cache friendly computing can drastically improve the execution speeds of data intensive scientific computations. This is an issue that is becoming more and more important [12]. The techniques involve either algorithm or data structure changes to make the best use of the computer cache hierarchy. A PDE solver of the type studied in this article does perform a rather large number of arithmetic operations for each grid cell in each iteration. However, the complexity of the PDE and the curvilinear computational grid imply that a lot of cell dependent data are required to perform the computations, resulting in that the computations are data intensive anyway. Our earlier experiments introducing cache friendly algorithm modifications and cache friendly data layouts have resulted in significant performance improvements for model problems [13]. In the present article we investigate the applicability of such techniques further.

For analyzing the memory utilization of our program we use the tool SIP [2]. SIP is a cache profiling tool that runs the program in a simulator and collects statistics about all memory and cache accesses. In this way, we get detailed information about the cache utilization for our code.

The finest level of optimization involves techniques to use the resources within the CPU as efficiently as possible. Examples of such techniques are subroutine inlining, loop unrolling and avoiding if-constructs. These, and other optimizations such as instruction scheduling, prefetching and register optimizations to avoid pipeline stalls, are usually introduced by the compiler. However, in [9] it was shown that manual code restructuring in order to improve register utilization lead to a performance improvement of 20 percent in comparison with the original compiler-optimized code.

Performance issues for PDE solvers have been studied by many authors. In one type of study the focus is on model problems, where it is relatively easy to isolate various performance aspects and study them in a controlled way. See, e.g., [8]. In that line of research, the authors of the present paper have earlier studied cache aware data layouts for the Gauss-Seidel smoother applied to a scalar PDE [13], and parallelization of a finite difference solver for the Euler equations of fluid dynamics [9]. In the latter case, the application is realistic, but the numerical method is simpler than the Jameson-type approach used in the present investigation.

Insights generated from studies such as those mentioned above form the basis for another kind of investigation, where the same types of issues are considered, but within more complex settings. In such studies, the interplay between the various performance aspects becomes significant. For example, Gropp et al. [1] make a detailed study of performance issues for a CFD code based on unstructured grids. The present article is an account of preliminary results in the same spirit, but addressing an alternative numerical methodology, using curvilinear, structured grids, which is common in aerodynamics simulations. Hoeflinger et al. [5] also address the same issues for two different application codes. One of their codes uses a numerical scheme of the simpler type that we addressed in [9], whereas the second code has many of the features of the kernel we consider here. However, they do not use the Jameson-type flux-splitting and multigrid techniques that our kernel includes. As a final example, Roe and Mehrotra [11] consider the parallelization of a multigrid solver for incompressible, viscous flow. In contrast to us, they focus exclusively on parallelization, and they do not include flux-splitting.

2 The Computational Kernel

The emphasis of the present contribution is on parallelization and cache utilization of the computational kernel. However, before we go on to discuss these topics, it is relevant to describe the computational kernel in more detail.

As mentioned, the equations being solved are the compressible Euler equations in three space dimensions. The conservative form is assumed:

$$\frac{\partial W}{\partial t} = \frac{\partial}{\partial x}f(W) + \frac{\partial}{\partial y}g(W) + \frac{\partial}{\partial z}h(W)$$

$$W = \begin{pmatrix} \rho \\ \rho u \\ \rho v \\ \rho w \\ \rho E \end{pmatrix}$$

$$f = \begin{pmatrix} \rho u \\ \rho u^2 + p \\ \rho uv \\ \rho uw \\ u(\rho E + p) \end{pmatrix}, g = \begin{pmatrix} \rho v \\ \rho vu \\ \rho v^2 + p \\ \rho vw \\ v(\rho E + p) \end{pmatrix}, h = \begin{pmatrix} \rho w \\ \rho wu \\ \rho wv \\ \rho w^2 + p \\ w(\rho E + p) \end{pmatrix}$$

Here, ρ is the density; u, v, and w are the velocity components, corresponding to the three space directions x, y, and z, respectively; p is the pressure, and E is the total energy per unit of mass.

The equations are discretized by means of a finite volume method. The flux vectors are split, so that the discretization takes into account the directions of propagation in the solution (so called flux-splitting, see e.g. [4]).

The resulting discrete scheme is a set of nonlinear algebraic equations, which has to be solved in order to advance the solution from time level n to time level $n + 1$. Applying Newton's method to these nonlinear equations yields a linear system for each Newton iteration.

These linear systems are solved iteratively, via a Symmetric Gauss-Sediel (SGS) method. We modify the scheme presented in [7] slightly, and introduce a red-black-ordered version of the SGS method. This will result in a much more efficient parallelization than the original SGS algorithm. At this point, we also let the time step go to infinity, since we are solving for steady state.

Finally, the resulting SGS-Newton iteration is embedded into a multigrid scheme for convergence acceleration. We use the full multigrid approach described by Jameson [6].

For the experiments presented below, we use a code that only implements the computational kernel described above. No initialisation and post-processing is included, and also the boundary conditions for the numerical scheme are ignored. In an application code, local boundary conditions are used, which have a similar structure to the upwind difference scheme used in the interior of the domain.

3 Cache Utilization

To summarize, the computational kernel described above consists of an inner SGS-Newton iteration, which is used as a smoother within the outer multigrid iteration. In the present section, we focus on the inner iteration, and in particular on efficient cache utilization.

In [13], we proposed cache-aware data layouts for the Gauss-Seidel smoother applied to a scalar PDE, and reported on experiments where these layouts resulted in significant performance improvements. Below, we briefly review those results, and discuss the modifications needed in order to address the more complex SGS-Newton iteration.

In a grid based computation there are two characteristics of cache usage that can be utilised to improve performance. The computation exhibits spatial locality, since computations at neighbouring grid points make use of the same data, or at least contiguous data. The computation involves a series of sweeps across the grid leading to repeated accesses of the data, this is known as temporal locality. In order to make efficient use of the cache system it is important to enhance these two types of access patterns. This can involve changes to the algorithm or to the data structures. In order to improve temporal locality it is possible to fuse together two consecutive sweeps over the data and to block the data into submatrices that can fit into the cache. To further improve performance, it is possible to specifically lay the data in the memory according to the access patterns of the algorithm. For the scalar Gauss-Seidel smoother the nodes were designated red or black, the two groups updated alternately by a single forward sweep in each case and there is a fixed pattern of spatial data dependence. It was found that fusing and blocking lead to dramatic improvements in the performance and that data laying also improved the performance but the overhead in laying the data was significant for out-of memory grid sizes.

In the SGS-Newton iteration, there are both forward and backward sweeps and the data dependence is governed by the flux splitting. Based on these considerations, we have made preliminary experiments (Table 1) by trying to improve the spatial locality of the data by merging two of the arrays (Q and dQ) together. The tests were performed on a Sun Ultra 10 (UltraSPARC-III 333MHz) machine. The machine has a memory of 128 Mbytes. It has a 16kB level 1 instruction cache and a 16kB level 1 data cache. This data cache is direct mapped. The machine also has a unified level 2 cache of 2 MB which is direct mapped with a cache-line size of 64 bytes. As can be seen from the results, there is a slight improvement in the version with improved spatial locality.

We also hope to optimize the algorithm for multiple iterations of the smoother in order to reuse the data that visit the memory system to the maximum. The importance of this is shown in the results given in Table 2 for a red-black smoother on the same computer for a model problem. In this table, we give results for three versions. One is the standard red-black smoother in which all reds are updated followed by all blacks. The

Ni	Nj	Nk	Non-merged (Mflop/s)	Merged (Mflop/s)
10	10	10	128	138
20	20	20	134	139
30	30	30	132	136

Table 1: Arithmetic performance for two versions of the SGS smoother. Three different grids – of size Ni x Nj x Nk – were considered. Merging of two arrays lead to a slight performance improvement.

Number of iterations	Ni	Nj	Nk	Standard Red Black	Fused Red Black	Optimized Red Black
50	300	220	220	8511	4198	210
100	300	220	220	15532	8131	370
100	402	220	220	28360	14262	552
200	402	220	220	57838	26327	1058

Table 2: Execution times (seconds) for different versions of the red-black smoother, for different grid sizes Ni x Nj x Nk and number of iterations.

second version is the fused version in which as soon as reds in a plane are updated then the blacks in the previous plane are also updated. This will ensure more data reuse than the standard version. In the final optimized version, as soon as we update one plane, we go back and update all the other planes that visited the memory system in the recent past, ensuring significant data reuse.

The conclusions from these preliminary investigations point to the benefits achievable through optimized data reuse. Currently we are investigating the application of this technique for the SGS smoother along with optimized data structures.

In addition to the studies reported above, we also studied the cache behaviour for the full parallelized kernel, where the smoother was implemented in a standard fashion, without using special techniques for "cache-awareness". For this purpose we used SIP [2], a simulator that allows for detailed analysis of features such as cache utilization. We were able to run the simulator for a $32 \times 32 \times 32$ grid, and found that in the loops where the smoother is applied, the cache miss rate was only 1.0 % for the L1 cache, and 17.5 % for the L2 cache. Most likely, these encouraging results are due to the fact that we do not use a model problem, but a rather complicated state-of-the art solver, with a very large number of arithmetic operations at each grid point.

4 Parallelization with OpenMP

Next, we turn the attention to the parallelization of the kernel. Among the different parallel programming models available, we focus on the global name-space model, for which OpenMP is a de facto standard. The aim is to explore the limits of this programming model, with respect to scalability. In a previous study [9], we demonstrated excellent scalability for an OpenMP implementation of a solver for the compressible Euler equations in non-conservative form, using a finite difference approximation, and including artificial dissipation. The performance of that OpenMP implementation was comparable to, in fact slightly better than, a corresponding implementation using the message passing programming model, represented by MPI. The key to that success for OpenMP on a Non-Uniform Memory Access (NUMA) machine, was that we used a computer platform with self-optimization (the Sun Orange [3]), so that memory pages were automatically migrated and/or cache lines replicated in order to move data to be close to the OpenMP threads that needed them most frequently.

Our present investigation differs from the previous one in two ways. First, the numerical algorithm is more complex. In particular, the multigrid iterations are more demanding from a parallelization point of view, since the arithmetics-to-synchronization ratio is significantly reduced each time the computations are moved to the next coarser grid.

Secondly, the computer platform is not equipped with self-optimization mechanisms. We now use the new generation of commercially available NUMA architectures, in our case a Sun Fire 15K with 48 processors. These architectures exhibit a relatively small difference between local and global memory access times. In other words, the non-uniformity of the memory accesses is less severe than in older NUMA architectures. Consequently, it is of interest to see if OpenMP could be competitive in this type of architecture, even without self-optimization mechanisms.

When parallelizing the code, we introduced a red-black ordering for the SGS iterations. Since the red-black ordering removes the distinction, which is central in SGS, between forward and backward sweeps, the consequence was that the parallel implementation uses standard Red-Black Gauss-Seidel iterations for the smoothing operation.

This was the only algorithmic change introduced in parallelizing the code. Apart from this, the parallelization was achieved via a relatively straightforward insertion of OpenMP directives [10] into the original, serial code. These directives serve to specify parallel sections and parallel loops, and to declare variables as shared or private.

In order to assess the scalability of the code, we measured fixed-size speedup for the multigrid iteration. That is, the problem size was fixed, and speedup was measured as $T(1)/T(p)$, where $T(p)$ is the execution time for one multigrid iteration on p processors.

It can be argued against the speedup measure, that it varies in a counter-intuitive way with the efficiency of the implementation. Given two codes, A and B, for the same problem, where the parallelizable parts of the code are more efficiently implemented in A than in B, it can be shown [14] that B will exhibit better speedup than A. That is, the faster code will be worse in terms of speedup. Consequently, almost any code can be shown to exhibit good scalability in terms of speedup, if only the implementation is inefficient enough!

For that reason, it is important to note, that we have taken care to first ensure that we have an efficient implementation of the serial version of the code. When executing our parallel code on one processor, we obtained a speed of 481 Mflops/s for the grid discussed below. This is 27 % of peak performance, which means that the code is reasonably well optimized. Under these circumstances, speedup can be taken as a valid indication of the scalability of the code.

For our speedup measurements, we used a $256 \times 256 \times 256$ grid on the finest multigrid level, and each multigrid iteration involved six grids in total. Grid coarsening was with a factor of two in each space direction, i.e., a total refinement factor of eight for each coarser grid level.

In each experiment with a fixed number of threads, p, ten multigrid iterations were executed, and $T(p)$ was taken to be the shortest iteration time among these. (The rationale for this is that no other users were executing, so variations in time per iteration were caused by operating system processes that were interfering. Consequently, the shortest iteration time is assumed to be the most significant measure of the performance of our code.)

The results are presented in Figure 1. The broken line indicates the ideal speedup. The "single grid" results refer to the case when all computations were carried out on the finest grid (i.e, multigrid was turned off). Since it is easier to obtain good speedup there, than for multigrid, the single grid case is a useful reference in assessing how well the multigrid iterations scale. As can be seen, our parallel multigrid implementation

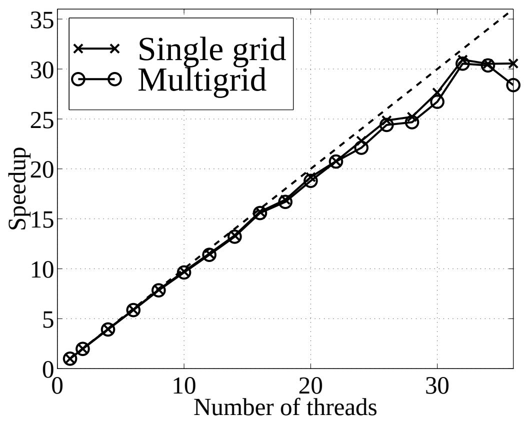

Figure 1: Speedup per iteration for the the OpenMP solver for the nonlinear Euler equations in 3D.

scales virtually on par with the single grid case, which is very satisfactory.

Figure 1 also shows that the speedup is deteriorating for 34 and 36 threads, respectively. This is because of load-imbalance. The 256 iterations of the outermost loop can not be evenly divided on 34 and 36 threads. Since 256/32=8, 256/34=7.5294, and 256/36=7.1111, some threads will be in charge of eight iterations, regardless of whether we use 32, 34, or 36 threads.

The parallel efficiency $E(p)$ of our code is very high. With the usual definition, i.e., $E(p) = S(p)/p$, the efficiency can at most be one. Our code maintains an efficiency between 0.88 and 1 for up to 34 threads (i.e., processors, since we schedule one thread per processor).

For comparison, the efficiency figures reported by Roe and Mehrotra [11] for a 256×256 grid drop to ca 0.7 for 8 processors, and to ca 0.55 for 16 processors. We can see two reasons for the better performance of our code. One is that we use three-dimensional grids, which increases the computation-to-synchronization ratio. The second reason is that they use an Origin 2000, i.e., an older generation NUMA system, which a larger difference between local and non-local memory accesses.

In conclusion, the parallel performance of our state-of-the-art CFD kernel, using OpenMP for the parallelization, is very good.

5 Conclusions

In view of the issues we set out to consider, the results reported here allow for at least one decisive conclusion. It is that parallellization with OpenMP gives excellent scalability, for a realistic application kernel. In fact, OpenMP is a more viable alternative in the case of a state-of-the-art 3D solver, than it would be for a simplified 2D model problem, with a lower computation-to-synchronization ratio.

It can also be concluded, at least tentatively, that the new generation of NUMA architectures, in our case the Sun Fire 15k, increases the competitiveness of the global name space programming model, represented by OpenMP. We were able to obtain good scalability with OpenMP without relying on additional features, such as self-optimization mechanisms or data placement directives.

With regard to cache utilization, we conclude, on one hand, that techniques such as fusing and blocking can be useful, and will be subject to further investigation as our project continues. On the other hand, and perhaps more importantly, the results from the SIP simulator show that for the application studied here, with a rather involved numerical scheme, inducing a very large number of arithmetic operations per grid point, even a standard implementation gives reasonably good cache utilization.

References

[1] W. K. Anderson, W. D. Gropp, D. K. Kaushik, D. E. Keyes, B. F. Smith. Achieving high sustained performance in an unstructured mesh CFD application. Report 2000-2, ICASE, 2000.

[2] E. Berg, E. Hagersten. SIP: Performance tuning through source code interdependence. To appear in the proceedings of Euro-Par 2002.

[3] E. Hagersten, M. Koster. WildFire: A Scalable Path for SMPs. In *Proceedings of the 5th IEEE Symposium on High-Performance Computer Architecture (HPCA)*, 1999.

[4] C. Hirsch. Numerical computation of internal and external flows. Vol. 2, *Computational methods for inviscid and viscous flows.* John Wiley & Sons, 1990.

[5] J. Hoeflinger, P. Alavilli, T. Jackson, B. Kuhn. Producing scalable performance with OpenMP: Experiments with two CFD applications. *Parallel Computing* 27:391–413, 2001.

[6] A. Jameson. Solution of the Euler equations for two-dimensional, transonic flow by a multigrid method. *Appl. Math. Comp.*, 13:327–356, 1983.

[7] A. Jameson, D. A . Caughey. How many steps are required to solve the Euler equations of steady, compressible flow: In search of a fast solution algorithm. In *15th Computational Fluid Dynamics Conference*, 2001.

[8] L. Noordergraaf, R. van der Pas. Performance Experiences on Sun's WildFire Prototype. In *SC99 Proceedings.* IEEE, 1999.

[9] M. Nordén, S. Holmgren, M. Thuné. OpenMP versus MPI for PDE solvers based on regular sparse numerical operators. In P. M. A. Sloth, C. J. K. Tan, J. J. Dongarra, and A. G. Hoekstra, editors, *Computational Science - ICCS 2002, Part III*, pages 681–690. Springer-Verlag, Berlin Heidelberg New York, 2002.

[10] OpenMP Architechture Review Board. OpenMP Specifications, www.openmp.org.

[11] K. Roe, P. Mehrotra. Parallelization of a Multigrid Incompressible Viscous Cavity Flow Solver Using OpenMP. ICASE Report No. 99-36, 1999.

[12] U. Ruede. Technological trends and their impact on the future of supercomputers. In H.-J. Bungartz, F. Durst, and C. Zenger, editors, *High Performance Scientific and Engineering Computing, Proceedings of the International FORTWIHR Conference on HPSEC*, pages 459–471. Springer, 1998.

[13] M. Silva. Cache aware data laying for the Gauss-Seidel smoother. In *Proceedings of the Tenth Copper Mountain Conference on Multigrid Methods*, 2001. Accepted for publication in ETNA.

[14] X. Sun, J. L. Gustafson. Towards a better parallel performance metric. *Parallel Computing*, 17:1093–1109, 1991.