HICFD – Highly Efficient Implementation of CFD Codes for HPC Many-Core Architectures

Achim Basermann, Hans-Peter Kersken, Andreas Schreiber, Thomas Gerhold, Jens Jägersküpper, Norbert Kroll, Jan Backhaus, Edmund Kügeler, Thomas Alrutz, Christian Simmendinger, Kim Feldhoff, Olaf Krzikalla, Ralph Müller-Pfefferkorn, Mathias Puetz, Petra Aumann, Olaf Knobloch, Jörg Hunger, and Carsten Zscherp

Abstract The objective of the German BMBF research project *Highly Efficient Implementation of CFD Codes for HPC Many-Core Architectures* (HICFD) is to develop new methods and tools for the analysis and optimization of the performance of parallel computational fluid dynamics (CFD) codes on high performance computer systems with many-core processors. In the work packages of the project it is investigated how the performance of parallel CFD codes written in C can be increased by the optimal use of all parallelism levels. On the highest level MPI is utilized. Furthermore, on the level of the many-core architecture, highly scaling, hybrid OpenMP/MPI methods are implemented. On the level of the processor cores the parallel SIMD units provided by modern CPUs are exploited.

Achim Basermann, Hans-Peter Kersken, and Andreas Schreiber
German Aerospace Center e.V. (DLR), Simulation and Software Technology
e-mail: {achim.basermann, hans-peter.kersken, andreas.schreiber}@dlr.de

Thomas Gerhold, Jens Jägersküpper, and Norbert Kroll
DLR, Institute of Aerodynamics and Flow Technology
e-mail: {thomas.gerhold, jens.jaegerskuepper, norbert.kroll}@dlr.de

Jan Backhaus and Edmund Kügeler
DLR, Institute of Propulsion Technology; e-mail: {jan.backhaus, edmund.kuegeler}@dlr.de

Thomas Alrutz and Christian Simmendinger
T-Systems SfR GmbH; e-mail: {thomas.alrutz, christian.simmendinger}@t-systems-sfr.com

Kim Feldhoff, Olaf Krzikalla, and Ralph Müller-Pfefferkorn
Technische Universität Dresden, ZIH
e-mail: {kim.feldhoff, olaf.krzikalla, ralph.mueller-pfefferkorn}@tu-dresden.de

Mathias Puetz
IBM Deutschland GmbH; e-mail: mathias.puetz@de.ibm.com

Petra Aumann and Olaf Knobloch
Airbus Deutschland GmbH, Aerodynamic Tools and Simulation
e-mail: {petra.aumann, olaf.knobloch}@airbus.com

Jörg Hunger
MTU Aero Engines GmbH; e-mail: joerg.hunger@mtu.de

1 Introduction

The research project HICFD is funded by the German Ministry for Education and Research within the programme *IKT 2020 - Research and Innovation*. The projects objective is to develop new methods and tools for the analysis and optimization of the performance of parallel CFD codes on high performance computer systems with many-core processors and to exemplarily apply these to DLRs CFD programs TAU [11] (computation of external flows) and TRACE [12] (simulation of internal flows).

In the workpackages of the project it is examined how the performance of parallel CFD codes can be increased by the optimal exploitation of all parallelism levels. On the highest, with MPI (Message Passing Interface) parallelized level an intelligent mesh partitioning in order to improve the load balance between the MPI processes is promising. For the block-structured grids used in TRACE a many-core compatible partitioning tool is developed.

Furthermore, on the level of the many-core architecture, highly scaling, hybrid OpenMP/MPI methods (OpenMP: Open Multi-Processing) are implemented for the CFD solvers TAU and TRACE. Within the block-structured CFD code TRACE the iterative algorithm for the solution of linear systems of equations has to be optimized, among other things by preconditioning methods adequate for many-core architectures.

On the level of the processor cores a pre-processor is developed which makes a comfortable exploitation of the parallel SIMD units (Single Instruction Multiple Data) possible, also for complex applications. For a detailed performance examination of SIMD operations the tracing abilities of the performance analysis suite VAMPIR *Visualization and Analysis of MPI Resources* [13] is further developed.

2 A Generic SIMD Preprocessor

Data parallelism at the level of the processor cores using their SIMD units cannot be exploited simply by hand. First, using SIMD instructions makes the code highly platform dependent. Second, data-level parallel programming using SIMD instructions is not a simple task. SIMD instructions are assembly-like low-level, and often steps like finalization computations after a vectorized loop become necessary. Thus it is essential to find or develop a tool suitable to comfortably and automatically vectorize CFD codes.

A vectorization tool is best built in a compiler. All current C compilers provide autovectorization units. We tested the compilers being in consideration for the HICFD project wrt. their autovectorization capabilities but had to decide not to rely on them regarding the vectorization process due to the following reasons:

- Each compiler has its own means to provide meta information. It is impossible to hide the different syntactic forms in a catch-all macro since some annotations like pragmas cannot be generated by the C preprocessor. Thus, in order to gain

the required relative independence from a particular compiler we would have to pollute the source code with a lot of vendor-dependent annotations.
- During our tests a lot of subtle issues arose around compiler-generated vectorization. For instance in one case a compiler suddenly rejected the vectorization of a particular loop just when we changed the type of the loop index variable from `unsigned int` to `signed int`. A compiler expert can often reason about such subtleties and can even dig in the documentation for a solution. An application programmer, however, normally concentrates on the algorithms and cannot put too much effort in the peculiarities of each compiler used.
- The vectorization of certain (often more complex) loops was rejected by all compilers regardless of inserted pragmas, given command-line options etc.

We checked other already available vectorization tools [4, 7], but the development focus of these tools did not match our requirements. Thus eventually we decided to develop a new tool in order to comfortably exploit the parallel SIMD units. Based on the requirements imposed by the numerical kernels of the CFD codes and based on experiences collected in former projects [6] the main function of the tool is the vectorization of loops. Beyond the specific target of CFD codes, the tool is usable as a universal vectorizer and aims at becoming an industrial-strength vectorizing preprocessor.

2.1 The Vectorizing Source-to-Source Transformator Scout

We opted for a strict semi-automatic vectorization. That is, as with compilers, the programmer has to mark the loops to be vectorized manually. Thus the analysis phase can be limited to the minimum necessary for vectorization and the focus can be put on code generation. We have called this vectorization tool Scout.

Most properties of Scout are direct consequences of the demands of the HICFD project:

- The input language is C since this is the programming language of the CFD codes considered in HICFD.
- The output language is C, too. This comes due to the fact that the simulation based on the CFD codes has to run on heterogeneous platforms each with different C compilers available.
- Scout focuses on short SIMD architectures. These architectures are very common in the current processor cores of even high performance computer systems.
- Scout is configurable in order to support the various already existing SIMD instruction sets as well as upcoming ones.
- Scout is usable on a day-to-day basis in the software production process. That is, at first a programmer can experiment with the tool in order to find the best transformation possible. In addition he is able to check the correctness of the output with a graphical user interface. Secondly, Scout exposes a command line interface as well and thus can be used as part of an automatic build chain.

2.2 Practical Results

We have applied Scout to two different CFD production codes used in the German Aerospace Center (DLR). Both codes are written using the usual array-of-structure approach. That approach is rather unfriendly wrt. vectorization because vector load and store operations need to be composite. Nevertheless we did not change the data layout but only augmented the source code with the necessary Scout pragmas. Thus the presented performance results were generated using the source code as is. All presented measurements were done on an Intel® Core™ 2 Duo P8600 processor with a clock rate of 2.4 MHz, operating unter Windows 7™ using the Intel® compiler version 11.1. Other systems showed similiar results, however the GNU compiler 4.4.1 appeared to optimize the SSE intrinsics differently and thus produced slightly slower code.

The first code, TRACE, uses a structured grid. Direct indexing is used in loops, and array indices are linearly transformed loop indices. We extracted four different computation kernels from the code and measured each individually. Kernel 1 computes the viscous fluxes of the RANS equations over a cell area of the control volume while kernel 2 determines the corresponding convective fluxes. In kernel 3, the derivatives of the fluxes according to the state vector (density, velocity etc.) are calculated due to the implicit time discretization of the RANS equations. Kernel 4 solves the equation system resulting from the implicit discretization with the Backward-Euler scheme by an incomplete *LU* decomposition of the matrix and a subsequent Gauss-Seidel iteration. The four kernels contribute to the wall clock time of a TRACE run with distinctly more than 50%. Fig. 1 shows typical speedup factors of the vectorized kernels produced by Scout compared to the original kernels. On the right, the speedup using double precision is shown (two vector lanes), on the left, the speedup using single precision is shown (four vector lanes). The overall picture does not change for larger problem sizes. Usually the number of iterations are roughly problem size cubed since most computations are done in triply nested loops.

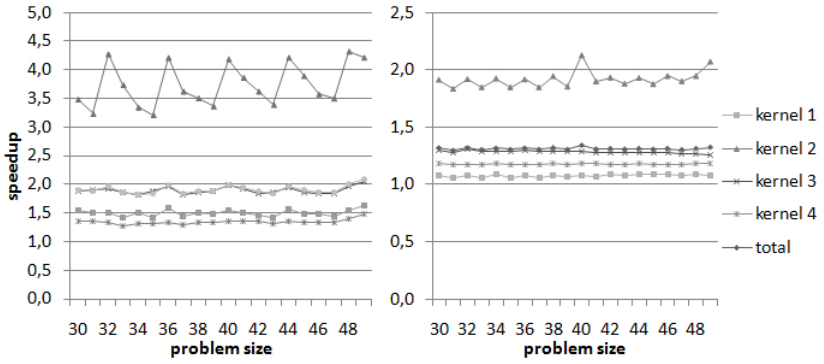
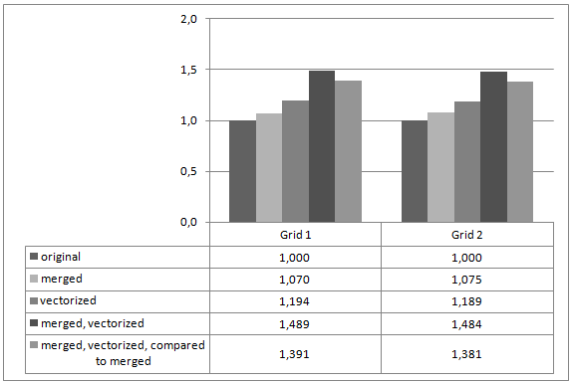


Fig. 1 Speedup of TRACE kernels gained due to the vectorization performed by Scout.

As expected, we gain a higher speedup with more vector lanes since more computations can be executed in parallel. The speedup of each particular kernel heavily depends on the computation workload compared to the number of load and store operations. Kernel 2 even outperforms its theoretical maximum speedup which is a result of further transformations (in particular function inlining) performed by Scout implicitly.

Unlike TRACE, TAU exploits unstructured grids. Thus the loops mostly use indirect indexing to access array data elements. We concentrated our efforts on the most performance critical kernel. The loops in that kernel could only partially be vectorized and to our best knowledge there is currently no other tool capable of vectorizing these loops automatically. Nevertheless we could achieve some speedup as shown in Fig. 2. We had two different grids as input data to our disposal. First we vectorized the code as is. However the gained speedup of about 1.2 was not satisfying. Hence, we merged some loops inside the kernel together to remove repeated traversal over the indirect data structures. This not only resulted in a better performance of the scalar code but also in a much better acceleration due to vectorization.



	Grid 1	Grid 2
original	1,000	1,000
merged	1,070	1,075
vectorized	1,194	1,189
merged, vectorized	1,489	1,484
merged, vectorized, compared to merged	1,391	1,381

Fig. 2 Speedup of a partially vectorized TAU kernel.

3 Performance Analysis of SIMD Codes

In the following, the performance of the parallel CFD programs TAU and TRACE with respect to the application of SIMD techniques for code acceleration are examined. By means of SIMD techniques, a single machine instruction can, at the same time, be applied to multiple data elements.

In order to investigate where and when a (parallel) program makes use of SIMD methods a special benchmark suite was developed and the performance analysis software VAMPIR was extended to visualize and analyze the SIMD operations. With the help of graphical VAMPIR displays, a user is able to verify where the SIMD preprocessor Scout (cf. section 2) produces source code on the basis of SIMD

techniques. In detail, the development of the displays targets answering the following questions: Are SIMD techniques used? If so where are they used? At which scale SIMD techniques are used? How can the quality of the application of SIMD techniques be measured?

Detailed information about the application of SIMD techniques during a program run can be collected by exploiting the information stored in hardware performance counters. These counters are special registers on microprocessors, in which values about hardware events (the total number of all executed SIMD intructions, e.g.) are stored.

For the development of the new graphical VAMPIR displays, it is advantageous to first analyze the SIMD performance of a program on a coarse-grained level. Therefore, a benchmark suite designed to statistically measure the SIMD performance of a program based on the information stored in hardware performance counters was developed. This SIMD benchmark suite (called *simdperfana*) consists of a C library (called *simdbench*) and a (*bash*) shell script collection (called *ccperfsimd*). SIMD instructions are recorded by means of hardware performance counters which will be accessed in the high level programming language C via PAPI (*Performance API*)[10].

simdbench and *ccperfsimd* interact in the following way: By means of the C library *simdbench*, a user can instrument source code parts of a program. By means of the shell script collection *ccperfsimd*, the SIMD performance of the instrumented program is automatically measured (program runs with different compiler options: vectorization type, optimization level, SIMD technique).

The benchmark suite is slim, clearly arranged and easy to use: *simdbench* offers four C macros (prolog, start, stop, epilog) to trigger a measurement, measurement parameters are changeable via *bash* environment variables. Furthermore, *simdperfana* supports PAPI and VampirTrace (a tool for tracing program runs which works together with VAMPIR) [14]. Thus the SIMD performance can easily be analyzed with VAMPIR.

On the basis of various tests (e.g. program runs of different numerical kernels), a VAMPIR display was adapted so that a user can detect where a program makes use of SIMD techniques and at which scale they are used.

4 Hybrid Shared-Memory/MPI Parallelization

In the modern process of the aerodynamic design of aircrafts or turbines for jet engines, CFD plays a key role. Large-scale computing resources allow more detailed numerical investigations with bigger numerical configurations. Up to 50 million grid points in a single simulation are becoming a standard configuration in the industrial design. These requirements demand a shift from the paradigm of *single-program single-data (SPSD)* per MPI-Rank to a more sophisticated approach with *single-program multiple-data (SPMD)* per MPI-Rank. The benefit and techniques of this

hybrid parallelization approach are demonstrated with the two DLR CFD solvers TAU and TRACE.

4.1 Hybrid Multistage Shared-Memory/MPI Parallelization of the TAU Flow Solver

At project start the parallelization of the TAU solver was based on MPI using a vertex-oriented domain decomposition per process with one layer overlap between the domains for the MPI communication. Typical simulations with TAU for three dimensional geometries use grid sizes between 10 to 50 million grid points and are computed with 32 to 512 MPI processes. Usually, such computations are performed employing a geometrical multigrid method with a 3- or 4-level W multigrid cycle. For e.g. a grid with 30 million grid points and 512 MPI processes, each grid domain contains approximately 60,000 grid points (without overlap) on the finest grid and 7500, 940, and 120 points on the second, third, and forth grid level, respectively. For such applications TAU has a very good parallel efficiency as can be seen in Fig. 3.

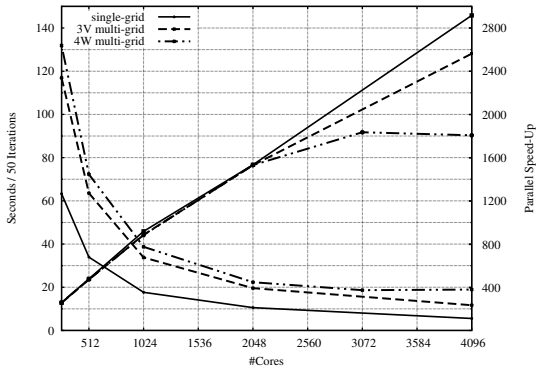


Fig. 3 TAU-Code performance up to 4096 cores on AMD Barcelona 1.9 GHz with Infiniband DDR ConnecX interconnect. Strong scaling for a mesh with 31 million points, cell-vertex metric, one MPI process per core. Explicit Runge/Kutta RANS solver with multi-grid (pre-smoothing only; 3V/4W: 3-level V/4-level W cycle): central discretization, scalar dissipation, Spalart/Allmaras turbulence model.

This figure also shows that, at least for a 4-level W multigrid cycle, the parallel efficiency decreases significantly when using more than 2000 MPI processes (see also [2]). The degradation of the efficiency is caused by the effect that the grid partitions become very small, especially on the coarse multigrid level so that the number of grid points in the overlap layer is no longer negligible. For instance with 4096 partitions, the number of additional points to be communicated is already about 35 percent of the total number of points in the non-decomposed grid. This effect is even worse on the coarse multigrid levels. One possibility to avoid such small grid partitions (for this number of processes) is a hybrid shared-memory/MPI parallelization. Following this concept, the grid is partitioned into as many domains as cluster nodes (or sockets) are to be used (rather than the number of cores). This increases the grid sizes per partition by one order of magnitude as today's clusters have nodes with 8

to 24 cores. Thus, the potential of this strategy is to improve the (strong) scalability, i.e., to shift the decrease of parallel efficiency to much higher core numbers.

Therefore, a prototype version of TAU has been implemented using multiple threads per cluster node (one per core) for the flow computation inside the grid domain. Between the grid domains, data is exchanged using MPI as before. The main computational work in the solver is in loops over points and edges of the grid. Points and edges of each grid domain are grouped into sub-domains, called colors, which are used to optimize the cache access. The computation of these colors is performed in parallel by multiple threads. As there are points that belong to more than one color, the updates of these points have to be serialized to avoid data races (when different threads update the same point at the same time). For this serialization, a locking mechanism for the colors has been developed to ensure mutual exclusion of neighboring colors. Namely, at any point in time, all colors processed (in parallel by multiple threads) do not share any points. The computation of the colors is asynchronous except at some synchronization points when MPI communication with neighboring domains has to be done. Performance benchmarks for the hybrid-parallel implementation are ongoing work; preliminary results are encouraging (cf. [2]).

4.2 Hybrid Shared-Memory/MPI Parallelization of the TRACE Flow Solver

In the design phase of multistage compressors and turbines for jet engines, or stationary gas turbines, complete computations of the configurations over night are mandatory for the engineers in order to analyze and improve the design throughout the next business day.

At the Institute for Propulsion Technology of the German Aerospace Center (DLR), the parallel simulation system TRACE (Turbo-machinery Research Aerodynamic Computational Environment) has been developed specifically for the calculation of internal turbo-machinery flows. TRACE applies a finite volume approach with block-structured grids for solving the associated Navier-Stokes equations. In the MPI parallelization of the code one MPI process is assigned to each block. Thus the scalability of TRACE is limited to the number of blocks. Moreover the load balance is dominated by the largest block. While it is possible to split the largest block into smaller ones, it is often not advisable to do so since smaller MPI domains have a larger overhead (due to the overlap region) and also decrease the convergence rate of an implicit solver. Hence a hybrid parallelization is the matter of choice in order to improve the performance and scalability of the TRACE code.

The implementation of the hybrid parallelization concept made a complete redesign of the TRACE data structures necessary. In order to maximize spatial and temporal data locality the data structures of TRACE were reorganized into hot and cold parts. The employment of a novel hyperplane formulation of the Gauss-Seidel relaxation algorithm, where the respective elements were addressed directly in hy-

perplane coordinates, makes it possible to improve also the scalar efficiency by a considerable margin. Speedup factors of up to 2.4 were observed, depending on the use-case and hardware platform.

The hybrid parallelization concept itself is based on pthreads and MPI. In this implementation pthreads are used for all processing units of a CPU socket and the enveloping MPI process is bound to the socket.

With these improvements an engineer is now able to compute a complete compressor map over night. The example given in table 1 is a 15 stage compressor with an additional inlet guide vane and an outlet guide vane. The mesh has 19,11 million grid points overall for 32 blade rows. Table 1 shows the runtime for a single point on the compressor map. If Hyperthreading is enabled a speedup factor of 2.7 on the 4 cores of an Intel Xeon 5540 (Nehalem) is obtaind by the shared-memory parallelization introduced (see also [9]). The hybrid parallelization made the efficient use of a much higher node number for the compressor problem possible so that the execution time could be reduced from 6 h 45 min (pure MPI) to 2 h 31 min (hybrid concept), cf. Table 1 for details.

Table 1 Runtime comparison for the compressor use-case

Testcase	Compiler	MPI	# nodes	# MPI processes	# threads/MPI process	Runtime in h
Compressor	icc 11.1	Openmpi	8	60	1	06:45:30
Compressor	icc 11.1	Openmpi	30	60	8	02:31:46

5 Scalable Algorithms for Many-Core Architectures

In order to increase the scalability of CFD solvers with multi-block grids like TRACE the development of a generic partitioning tool for block-structured grids is required (cf. section 5.1). On the other hand it is necessary to improve the parallel efficiency of the linearized TRACE solvers by scalable preconditioning for the iterative solution method for sparse systems of linear equations (cf. section 5.2).

5.1 Partitioning for Block-Structured Grids

A promising approach to increase the scalability of parallel CFD solvers with block-structured grids on the highest, with MPI parallelized level is an intelligent grid partitioning for improving the load balance between the MPI processes. For TRACE, we developed a many-core compatible partitioning tool which will be available as open-source library after the end of the project.

A first version was already integrated into TRACE and exploits ParMETIS [5] for partitioning the block graph. The nodes of this graph are weighted with the

computational load per block while the edge weights represent the data exchange volume between the block faces. The advantage compared with the standard block distribution in TRACE which just considers the calculations per block is a reduction of the communication overhead. Fig. 4, left, shows that this advantage increases with increasing core number for TRACE sample runs on DLRs AeroGrid cluster (45 Dual-processor nodes; Quad-Core Intel Harpertown; 2.83 GHz; 16 GB main memory per node; InfiniBand interconnection network between the nodes).

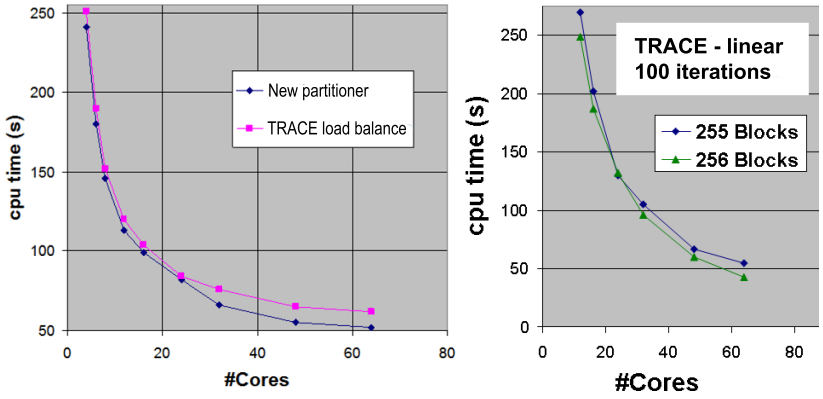


Fig. 4 Execution times of TRACE sample runs with the new partitioner and the standard TRACE load balancer (left) as well as with and without manual splitting of the largest block using the new partitioner (right).

Another main source for scalability losses in TRACE is the significant size difference of the blocks. Therefore, the partitioning tool developed will be extended by a splitting feature of large blocks into smaller ones. This makes on the one hand a more advantageous load balance and on the other hand the efficient exploitation of a higher core number possible. Fig. 4, right, illustrates this for the case that the largest of 255 blocks is manually split into halves for TRACE sample runs on the AeroGrid cluster. The resulting 256 blocks are then distributed with the new partitioner.

5.2 Preconditioning for the Linear TRACE Solvers

For the parallel iterative solution of sparse equation systems to be solved within the linearized TRACE solvers, FGMRes with Distributed Schur Complement (DSC) preconditioning [8] for real or complex matrix problems has been investigated.

The DSC method requires adaquate partitioning of the matrix problem since the order of the approximate Schur complement system to be solved depends on the number of couplings between the sub-domains. Graph partitioning with ParMETIS [5] from the University of Minnesota is suitable since a minimization of the number

of edges cut in the adjacency graph of the matrix corresponds to a minimization of the number of the coupling variables between the subdomains. The latter determine the order of the approximate Schur complement system used for preconditioning.

For the the solution of TRACE linear equation systems, we developed a parallel iterative FGMRes algorithm with DSC preconditioning. In [3] and [1], we demonstrated the superiority of the DSC preconditionier over (approximate) block-Jacobi preconditioning. Block-local preconditioning methods like block-Jacobi are the standard preconditioners in TRACE. Here, we particularly discuss numerical and performance results of DSC methods for typical complex TRACE CFD problems on many-core architectures.

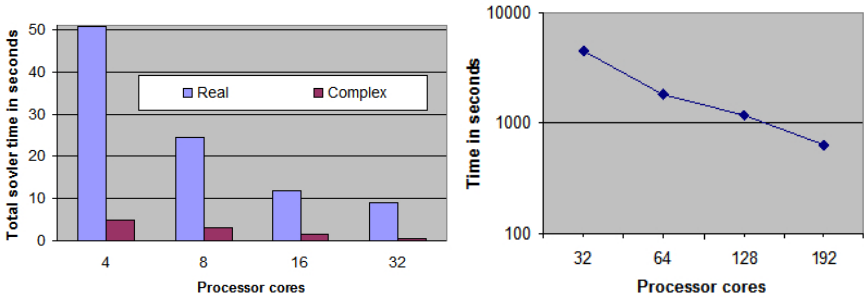


Fig. 5 Execution times of the DSC solver for a complex and the corresponding real small size TRACE matrix problem (left) as well as for a complex large TRACE matrix problem (right) on a many-core cluster.

Fig. 5, left, displays execution times of the real and complex DSC solver software developed on 4 to 32 processor cores of the AeroGrid cluster for a complex and the corresponding real small TRACE matrix problem of order 28,120 or 56,240. The FGMRes iteration was stopped when the current residual norm divided by the initial residual was smaller than 10^{-5}. The complex DSC solver version distinctly outperforms the real version. This is caused by the lower problem order and a more advantageous matrix structure in the complex case opposite to the real case. In addition, the complex formulation results in higher data locality (storage of real and imaginary part in adjacent memory cells) and a better ratio of computation to memory access due to complex arithmetics in comparison with the real formulation. Fig. 5, left, also shows an advantageous strong scaling behavior of the DSC method on the many-core cluster, even for this small matrix problem.

Fig. 5, right, shows execution times of the complex DSC method on 4 to 192 processor cores of the AeroGrid cluster for a large complex TRACE matrix problem of order 4,497,520 and with 552,324,700 non-zeros. In order to achieve the required accuracy in each component of the solution vector the FGMRes iteration was here stopped when the current residual norm divided by the initial residual was smaller than 10^{-10}. Fig. 5 demonstrates that the complex DSC algorithm scales very well on many-core clusters for differnt CFD matrix problems of varying size.

6 Conclusions

After two of three HICFD project years, the methods and tools developed showed promising performance results for the targeted CFD codes TAU and TRACE on all three parallelism levels considered. Significant speedups could be gained by the optimized exploitation of SIMD operations on the processor core level, hybrid parallelization techniques on the level of the many-core architecture and more scalable algorithms on the highest, with MPI parallelized level.

In the remaining project time, the developments on all parallelism levels will be completed and fully integrated into TAU and TRACE. An evaluation with industrially relevant test cases will demonstrate the increased competitiveness of the CFD simulations codes modified in HICFD.

Acknowledgements This work has been supported by the German Federal Ministry of Education and Research (BMBF) under grant 01IH08012 A.

References

1. Alrutz, T., Aumann, P., Basermann, A., et al.: HICFD – Hocheffiziente Implementierung von CFD-Codes für HPC-Many-Core-Architekturen. In: Mitteilungen - Gesellschaft fr Informatik e. V., Parallel-Algorithmen und Rechnerstrukturen, ISSN 0177 - 0454, pp. 27–35 (2009)
2. Alrutz, T., Simmendinger, C., Gerhold, T: Efficiency enhancement of an unstructured CFD-Code on distributed computing systems. In: Proceedings of ParCFD 2009 (2009)
3. Basermann, A., Cortial-Goutaudier, F., Jaekel, U., Hachiya, K.: Parallel solution techniques for sparse linear systems in circuit simulation. In: Proceedings of the 4th International Workshop on Scientific Computing in Electrical Engineering, Series: Mathematics in Industry, Springer (2002)
4. Hohenauer, M., Engel, F., Leupers, R., Ascheid, G., Meyr, H.: A SIMD optimization framework for retargetable compilers. ACM Trans. Archit. Code Optim. (2009) doi: 10.1145/1509864.1509866
5. Karypis, G., Kumar, V.: ParMETIS: Parallel graph partitioning and sparse matrix ordering library. Tech. rep. # 97-060, University of Minnesota (1997)
6. Müller-Pfefferkorn, R., Nagel, W.E., Trenkler, B.: Optimizing Cache Access: A Tool for Source-to-Source Transformations and Real-Life Compiler Tests. In: Danelutto, M, Vanneschi, M. Laforenza, D. (eds.) Lecture Notes in Computer Science **3149**, pp. 72–81. Springer, Heidelberg (2004)
7. Pokam, G., Bihan, S., Simonnet, J., Bodin, F.: SWARP: a retargetable preprocessor for multimedia instructions. Concurr. Comput. : Pract. Exper. (2004) doi: 10.1002/cpe.v16:2/3
8. Saad,Y., Sosonkina, M.: Distributed Schur complement techniques for general sparse linear systems. SISC **21**, 1337–1356 (1999)
9. Simmendinger, C., Kügeler, E.: Hybrid Parallelization of a Turbomachinery CFD Code: Performance Enhancements on Multicore Architectures. In: Pereira, J.C.F., Sequeria, A. (eds.) Proceedings of ECCOMAS CFD 2010 (to appear)
10. http://icl.cs.utk.edu/papi/. Cited 15 Dec 2010
11. http://www.dlr.de/as/. Cited 15 Dec 2010
12. http://www.dlr.de/at/desktopdefault.aspx/tabid-1519/2123_read-3615/. Cited 15 Dec 2010
13. http://www.vampir.eu/. Cited 15 Dec 2010
14. http://www.tu-dresden.de/zih/vampirtrace. Cited 15 Dec 2010