Parallel Calculation of Non-Equilibrium Reentry Flows

Thomas Bönisch, Panagiotis Adamidis *and* **Roland Rühle**,
High Performance Computing Center Stuttgart

ABSTRACT: *The flow simulation program URANUS has been developed to calculate non-equilibrium flows around space vehicles reentering the earth's atmosphere. The program simulates not only the supersonic flow but also the chemical reactions occuring as they have a non-negligible influence on the flow. As a result of the calculation the heat flow and the heat load at the surface is of main interest. During the last years a parallel multiblock version of the simulation program has been developed as sequential systems cannot feed the programs needs in computing power and memory. The development was done on a Cray T3E using Fortran90 and MPI for Message Passing. Therefore, the resulting program is portable and was also tested on a wide range of high performance architectures like Hitachi SR8000 and NEC-SX5.*

1. Introduction

Calculating flows around space vehicles during the reentry phase of their mission is a challenging task. Besides the normal flow one has to consider the occurring chemical reactions as they have a significant influence on the flow. The reason for these chemical reactions is the high temperature of the gas flow during the reentry, while the space vehicle is slowed down by the friction of the air. At these temperatures the air's components mainly nitrogen and oxygen react with each other.

Modern space vehicles have complex geometries. To calculate flows around such a geometry, there exist several approaches: unstructured meshes, block structured meshes and other technologies like hybrid or overset meshes. Unstructured meshes can be generated automatically, mesh generation is much easier than generating a multiblock mesh. But the calculation of a flow using them costs a lot of computational power as indirect memory access methods have to be used. Therefore, efficient cache usage and vectorization is limited. As the development of the memory speed cannot keep track with the increase in processor performance this will be worse in the future. The second method is using structured meshes, but allowing several blocks of them to be assembled in an unstructured way. So, it is possible to mesh complex geometries. The calculation of such structured topologies profits from cache and vector technology. A good performance is much easier to obtain and it is normally higher than with unstructured meshes. But the mesh generation is a costly task. Sometimes it costs several weeks of an expert to generate a good so called multiblock mesh.

An already existing parallel version of the URANUS [1] flow solver for ideal gases uses structured 3D C-meshes for its calculation. However, single block c-meshes contain a singularity in the mesh which is complicated to handle and which also limits the convergence speed. Moreover, current topologies like an X-38 (Figure 1) with body flap cannot be meshed with such a single block c-mesh.

Figure 1. X-38 Reentry Vehicle

Considering this and the arguments presented above, we decided for the more natural step using multiblock meshes for our calculations instead of rewriting the code completely to handle unstructured meshes.

2. URANUS

In the URANUS (**U**pwind **R**elaxation **A**lgorithm for **N**onequilibrium Flows of the **U**niversity of **S**tuttgart) [5,6] flow simulation program the unsteady, compressible Navier-Stokes equations in the integral form are discretized in space using the cell-centred finite volume approach. The inviscid fluxes are formulated in the physical coordinate system and are calculated with Roe/Abgrall`s approximate Riemann solver. Second order accuracy is achieved by a linear extrapolation of the characteristic variables from the cell-centres to the cell faces. TVD limiter functions applied on forward, backward and central differences for non-equidistant meshes are used to determine the corresponding slopes inside the cells, thus characterizing the direction of information propagation and preventing oscillation at discontinuities. The viscous fluxes are discretized in the computational domain using classical central and one-sided difference formulas of second order accuracy.

Time integration is accomplished by the Euler backward scheme. The resulting implicit system of equations is solved iteratively by Newton`s Method, which theoretically provides the possibility of quadratic convergence for initial guesses close to the final solution. The time step is computed locally in each cell from a given CFL number. To gain full advantage of the behaviour of Newton`s method, the exact Jacobians of the flux terms and the source term have to be determined.

The resulting linear system of equations is iteratively solved by the Jacobi line relaxation method with subiterations to minimize the inversion error. A simple preconditioning technique is used to improve the condition of the linear system and to simplify the LU-decomposition of the block-tridiagonal matrices to be solved in every line relaxation step.

The boundary conditions are formulated in a fully implicit manner to preserve the convergence behavior of Newton`s method [7].

The usage of the sequential URANUS program on high-end workstations and on vector computers shows that the compute time and the memory requirements are too high to use the program on these platforms without the ability to use processors in parallel. Especially when changing to the real gas model or using fine meshes for real world problems. Therefore, that program version was parallelized in an earlier effort, because only massively parallel platforms and modern hybrid parallel computers can fulfil the program's needs in memory size and computing power.

We now want to introduce a parallel multiblock structure into this code. The main focus besides the multiblock structure itself when doing these enhancements was the efficiency of the program.

3. Requirements to an Efficient Parallel Multiblock Solver

To be able to calculate correct solutions on a multiblock mesh in parallel, a flow solver has to have several additional features.

Physical Boundaries

In the serial program using c-meshes each of the different physical boundaries is fixed to a dedicated plane of the mesh. This means, e.g. that the outflow boundaries are always fixed to the mesh plane, where the first index reaches its maximum. In the multiblock case this is not longer true. Theoretically, each boundary type can appear at each of the six planes of the 3D mesh. However, usually the mesh is generated in a way where the most complicated boundary is fixed to one specific plane of a block. So, the boundary with the condition of the body wall has only to be implemented for one index direction. Nevertheless, this limits the usability of the code, because this does not work for some special topologies where at least one of the blocks has to calculate wall boundary conditions at least at two of its boundaries.

Local Coordinate System

The ability of turning the mesh blocks such that always the same block surface fits to the body wall requires that each of the blocks can have its own local coordinate system. This is necessary anyway as the blocks itself are disposed unstructured around the geometry. At particular points where more ore less than four blocks in 2D or eight blocks in 3D respectively are connected together, the local coordinate system of at least one of the blocks neighbours has to be different. This is illustrated in Figure 2.

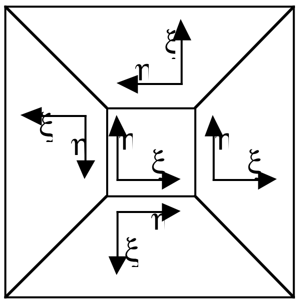

Figure 2. The different local coordinate systems and how they occure

Furthermore, in a multiblock mesh each block can have multiple neighbours on each of its block sides, not only one as it happens, when you are cutting a C-mesh into parts, or when using special mesh generators.

Different Mesh Block Sizes

In a multiblock mesh the arrangement and occurence of the blocks depends only on the meshed topology and the decisions of the mesh generartor. This means in general, that the blocks can be completely different not only in shape but also in size, where block size is measured in number of mesh cells per block. There can be orders of magnitude in block size between the largest and the smallest block.

Additional Requirements

As we are not using our own meshes within this project, we are depending on meshes of our partners. Moreover, we wanted to be able to calculate results for as many cases and with as much project partners as possible. So we were not interested in limiting ourselves too much in choosing one special multiblock mesh type. We wanted to keep a high flexibility of our code. So, using meshes from project partners should be as easy as adapting and extending the input routines.

The program parts for the parallelization the multiblock structure and the flow simulation core should be separated as far as possible. This increases maintainability as the modeling people still can recognize and update their flow patterns as the parallelization people can enhance the parallel performance.

4. The Parallel Multiblock Approach

As the flow simulator mentioned is memory and calculation intensive we aim at using the newest available hardware platforms. This requires a portable code to be able to move easily from one platform to another. In the parallel case this requires the usage of a standard parallel programming model. In order to be able to run on all platform types, like MPP's, SMP's and the new hybrid architectures consisting of distributed SMP nodes, we decided to use MPI[3] as communication library.

Currently, we use a pure distributed memory approach as this performs also well on today's SMP machines and on hybrid architectures. In a future version it would also be possible to add OpenMP[2] directives to profit in addition from SMP architectures. A pure SMP approach was not considered, because this would limit the platforms to be used. Additionally, the experience of NASA shows that such an approach on today's ccNUMA architectures requires an effort comparable to an implementation with MPI to achieve a good performance[4].

Data Structures

The new parallel program needs a newly designed data structure to meet the requirements of the multiblock structure. Using Fortran90 we have the whole ability of dynamic and structured data types. Therefore, it is not longer necessary to implement a data management which stores the 3D arrays of all blocks contiguously into a one dimensional array or to waste memory by allocating the same amount of memory for each block calculated on one processor, like it was usual when using FORTRAN77 [8].

In the new multiblock flow simulation program all information regarding one block is combined in a new data type. Each block is a instance of this data type having exactly the size the block needs. The blocks residing onto one process are organized in a pointered list of these instances. So, there is no waste of memory as only the really required memory is allocated. There is no additional programming effort in manually handling the memory layout as the run time system cares for this. The information is easily accessible and the data structure itself is easy to extend to meet future requirements. The data structure for a block includes the ability to use several meshes with different resolutions for one block. So the data structure is already prepared for additional features like local refinement and multilevel or multigrid technologies.

Domain Decomposition and Communication between the Blocks

For the parallelization we used the domain decomposition approach. Each block has a two cell overlap region at the inner boundaries to maintain the second order scheme without additional communication. The values of the overlap region are stored according to the local mesh values. This means, that they are stored converted to the local coordinate system of the block, even if the coordinate system of the block where they are originally located is different. Consequently, the conversion of the halo cell data to the particular local coordinate system is done during the data transfer between the blocks.

This communication between neighbours is necessary to update the results of the cells in the overlap region. During the solving step additional communication between the neighbours is performed to exchange intermediate data. This ensures a more accurate solution and a better convergence.

Due to the irregular structure the mesh blocks have within the meshed topology, the identification number of a neighbour block cannot be calculated out of the blocks own number and the block side. Additionally, a block can have several neighbours at each of its sides as mentioned above. Accordingly, there is a data structure implemented to store all the information about the relationship of the block and its particular neighbours. This information enfold the block number and the block side the local block is connected to, the neighbour blocks processor as far as the orientation of the neighbour block and the part of the local blocks side which is adjoined to the neighbour.

Each block is treated separately on its processor. There is no difference in the handling of its neighbour blocks independent where they are located. This means, the communication between two blocks residing on the same processor is also done using the regular communication routines of the program. In this case from the processor point of view the processor is sending a messages to itself.

To be able to hide the communication as good as possible behind some computation, non-blocking communication is used. Currently, the blocks on one processor are handled one after the other also during the communication. First all blocks for sending, then all blocks for receiving. An improvement would be possible by letting the processes wait for messages to receive, not blocks. Then a received message is passed to the respective block who probably can do some computational work then.

For the calculation of global values where each block has a contribution, the results of one processor's blocks are calculated locally and then exchanged between the processors using the collective communication patterns of the programming model.

A specific data structure also exists for each physical boundary type of a block, where all the data of one physical boundary type is specified. It contains the subtype and the exact position of this physical boundary on the block. Using this data structure, there is no branching necessary for each of the six block sides where a physical boundary can reside. Additionally, all physical boundaries of one type at a block can be handled efficiently in a loop.

Load Balancing

To obtain good performance on any kind of parallel computers it is essential to have a good load balance between the processors allocated to the parallel job. Thus, we should allocate the same portion of load to each processor. In our case this means that every processor should make calculations for more or less the same number of mesh cells.

Compared to an approach where the blocks which are too large are cut in the middle as long as there are empty processors[9], we used a different technique: we also have to cut all blocks which are too large to fit onto one processor. But our cutting algorithm first calculates the number of necessary pieces for each block to be cut into. This number must only contain powers of two and powers of three as divider. If the calculated number does not accomplish this requirement the next appropriate larger number of pieces doing so is calculated. As the resulting parts may become to small we allow a slight overload of the processors to avoid cutting into too much pieces. Then, the number of cuts necessary in each of the three dimensions is calculated in a way that the resulting blocks are not misshapen. This is also the reason for allowing only part numbers containing powers of two and three, because than we can guarantee a good shape.

Consider the case if we have to cut a block into fourteen pieces. Then we have to cut one time in one dimension, six times in a second and we cannot divide in the third dimension. If we are dividing the block into 16 parts instead, we can freely choose how to place the cuts within the three dimensions.

This strategy has an additional advantage as we are cutting all the blocks in that way: assume a block being divided two times in one dimension and its neighbour being divided four times in the same dimension. Assuming the same local coordinate system. Two of the three block parts have to communicate with two neighbour blocks each and one has to talk to three neighbours. This is shown in figure 3. Having six parts within this dimension on the neighbouring block, each of the three blocks has two neighbours and the dependencies are simpler. Sometimes, it can happen that we have to adjust the cutting line by one cell to ensure this. That feature is currrently not implemented.

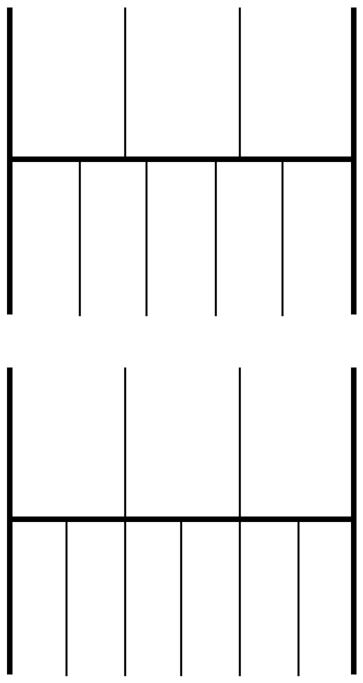
Figure 3. Different neighbour configurations

After having created the new blocks by really cutting the old ones, we have to rearrange the blocks in a way that no processor is overloaded. For this step we use the load balancing tool jostle[10]. Jostle was primarily designed to be used for unstructured meshes, but it does not know anything about meshes itself. As any other load balancing tool, jostle works on graphs. So, we only have to translate our new block structure, obtained after the cutting step, with its connections to the neighbours into a weighted graph where the block size is the weight of the node in the graph which corresponds to the block. If possible we get a load balanced block distribution and have to redistribute the blocks in the way proposed by jostle. As each block has its own data structure and is handled separately, it is easy to redistribute the blocks. Nevertheless, the time needed for this data transfer may be significant.

A problem with this procedure occur when all original blocks from the multiblock mesh have (nearly) the same size. Due to the algorithm we got a little bit more blocks than necessary, because we are only dividing using powers of two and three as parts. When this cannot be compensated by the slight overload factor, we end up with a block count which is a little bit larger than the number of processors. As all the blocks have still nearly the same size we end up with a significant load imbalance. This can either be solved by cutting blocks in parts of different size or by cutting one of the resulted blocks further on to fill the gaps on the processors with appropriate parts. As these problem can only be identified after having done the first load balancing step, this is to be done in a future step.

But we may not only have blocks which are too large for one processor, there may also be blocks which are too small to fully utilize the capacity of one processor.

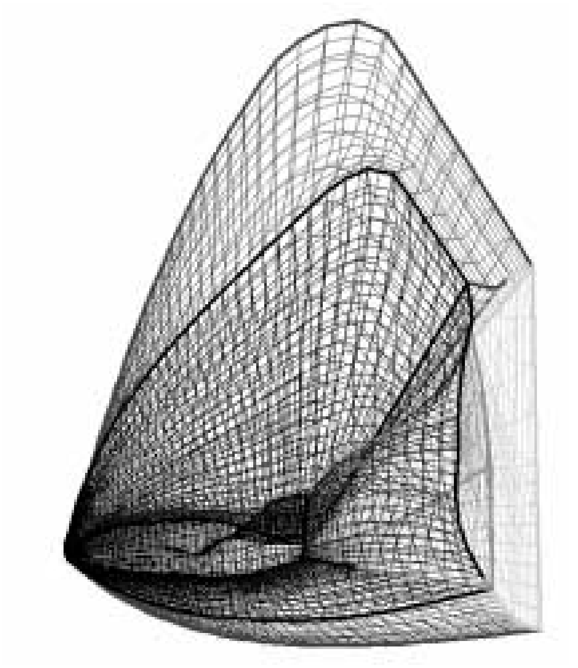

Figure 4. Original Multiblock Mesh

In order to gain all the cycles of these processors, the new program is able to handle more than one block on each processor. This is basically enabled by the already described data structure and Communication layout. The number of blocks possible on each processor is only limited by the processors memory.

As an example the figures 4 to 6 show how the load balancing procedure is working for a mesh with 68712 cells in 6 blocks. The largest block of the original mesh shown in figure 1 contains 32256 mesh cells, the smallest 1176 mesh cells.

Figure 5 shows the mesh produced by the automatic block cut algorithm, which cuts the blocks which are too large for one processor. It is assumed that the calculation will be done on 9 processors. The resulting mesh now has 11 blocks. The largest block (black) was cut into four pieces, the two blocks above and below the black (both dark grey) into two pieces each. The invisible block at the nose was not cut.

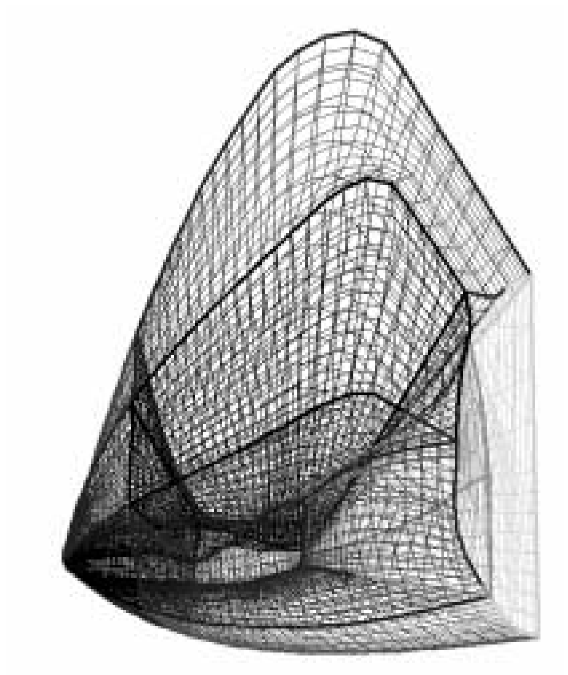

Figure 5. Mesh after blocks have been cut

In figure 6 the obtained distribution of the 11 blocks to the 9 processors is shown. Same numbers in the blocks mean same processor. The processor with the highest load has to calculate 8064 mesh cells, the processor with the lowest load has to calculate 6048 mesh cells. The load imbalance in this
case is 18 %. The load imbalance could be less, if we would do an additional run cutting a small block once more into pieces, in order to fill the small gaps on the less loaded processors. But this is currently not implemented.

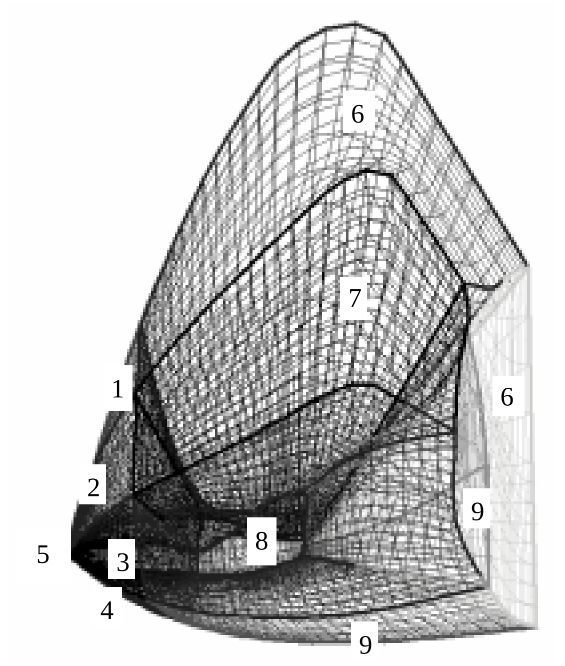

Figure 6. Mesh and block distribution after load balancer run

Due to the modular structure of the program, the replacement of the load balancing tool is easily possible.

Parallel Solver

An important step was the parallelization of the Jacobi line relaxation solver. The idea we found in [11] and we adapted to our needs was to use an additional splitting method to reduce the coupling between the matrix parts located on different processors. Let us assume we have a tridiagonal matrix A and have to solve the system $A\vec{u} = \vec{r}$ in parallel. We split the matrix *A* onto for example three processes,

$$A = L + M \tag{1}$$

with

$$L = \left(\begin{array}{cccc|cccc|cccc}
m_1 & u_1 & & & & & & & & & & \\
l_2 & m_2 & u_2 & & & & & & & & & \\
 & l_3 & m_3 & u_3 & & & & & & & & \\
 & & l_4 & m_4 & & & & & & & & \\
\hline
 & & & & m_5 & u_5 & & & & & & \\
 & & & & l_6 & m_6 & u_6 & & & & & \\
 & & & & & l_7 & m_7 & u_7 & & & & \\
 & & & & & & l_8 & m_8 & & & & \\
\hline
 & & & & & & & & m_9 & u_9 & & \\
 & & & & & & & & l_{10} & m_{10} & u_{10} & \\
 & & & & & & & & & l_{11} & m_{11} & u_{11} \\
 & & & & & & & & & & l_{12} & m_{12}
\end{array}\right)$$

and

$$M = \left(\begin{array}{cccc|cccc|cccc}
 & & & & & & & & & & & \\
 & & & & & & & & & & & \\
 & & & & & & & & & & & \\
 & & & & u_4 & & & & & & & \\
\hline
 & & & l_5 & & & & & & & & \\
 & & & & & & & & & & & \\
 & & & & & & & & & & & \\
 & & & & & & & & u_8 & & & \\
\hline
 & & & & & & & l_9 & & & & \\
 & & & & & & & & & & & \\
 & & & & & & & & & & & \\
 & & & & & & & & & & &
\end{array}\right)$$

This means *L* contains the local parts of the matrix *A* on each process and *M* contains the parts coupling these local parts. With this splitting of *A* we now have to solve

$$(L + M)\vec{u} = \vec{r} \tag{2}$$

moving the coupling parts on the right hand side we get

$$L\vec{u} = \vec{r} - M\vec{u} \tag{3}$$

Introducing an iteration scheme and now using the old solution on the right hand side for the update we have

$$L\vec{u}^{(j)} = \vec{r} - M\vec{u}^{(j-1)} \tag{4}$$

Putting the local parts of *A*, i.e. *L* on the right hand side by inverting *L* (locally on each process) we finally get

$$\vec{u}^{(j)} = L^{-1}\left(\vec{r} - M\vec{u}^{(j-1)}\right) \quad (5)$$

Now it is possible to solve the different matrix parts in parallel with a following communication step to exchange the results between the neighbours in order to update the right hand side for the next subiteration or iteration.

Based on this idea we developed a new solvers with a different communication patterns. The heptadiagonal system is split as shown above. The solver does a complete solving step of the system using the sequential solver on each process, exchanges the results with its neighbours and finally does an additional solving step without inverting the local matrix L a second time. This leads in total to an additional computational effort of about 20 % for each iteration.

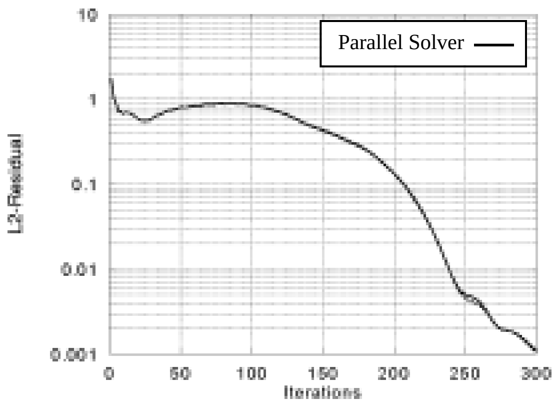

Figure 7. Convergence speed of the parallel solver

Figure 7 shows the convergence speed of the parallel solver which was measured to be 1% worse compared to the sequential one in a matchable case.

5. Results

The obtained parallel multiblock flow solver has been used to perform reentry calculations for the colibri capsule and the X-38, the prototype of the crew rescue vehicle for the international space station (ISS). In figure 8 the result of an Euler nonequilibrium computation for the X-38 is shown. The angle of attack is 40 degrees and the mach number is 19.8 .

Due to the usage of Fortran90 and MPI the code is portable and was tested on NEC SX-5, Hitachi SR8000, Cray T3E, IBM SP3 a DEC alpha cluster and an IA-64 system. The performance on the NEC-SX5e system with a well sized problem is nearly 1.5 Gflops which is 37% of the

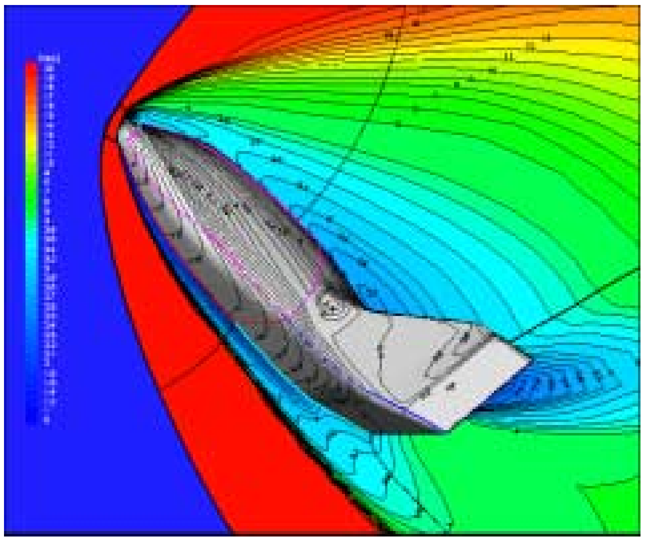

Figure 8. X-38 Calculated Mach number distribution, Euler nonequilibrium flow, angle of attack 40 degrees, mach number 19.8.

peak performance. Limiting factor on that system is currently the dynamic allocation of the temporary arrays, which is done within every iteration. This should be optimized in the future.

The scaled speedup on the Cray T3E is given in table1.

Mesh size	Mesh blocks	Proc. count	Simulation time	Efficency
24 000	5	18	255.3	1.0
192 000	5	144	285.7	0.893

Table 1. Scaled speedup on the Cray T3E

Figure 9 shows the Speedup for a 192000 mesh cell calculation for the colibri reentry vehicle. To compare the computing times, 20 iterations were calculated. The smallest case which could be calculated runs on 72 processors. Smaller processor numbers were not possible due to the memory usage of the program and the Cray T3E in Stuttgart having only 128 MB of memory per processor.

The given speedup values are compared to that case with 72 processors. The reason for the shown superlinear speedup for 108 and 216 processors is a better block shape in this cases compared to the case running on 72 processors.

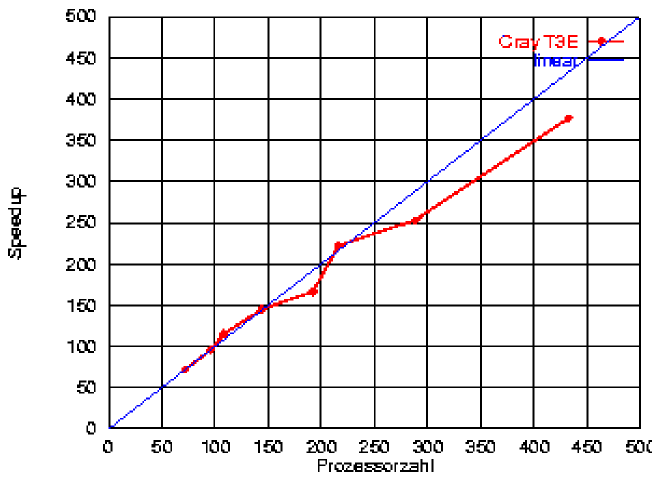

Figure 9. Speedup for 20 Iterations on a 192 000 cell colibri mesh on the Cray T3E

6. Acknowledgements

This work was particularly supported by the Deutsche Forschungsgemeinschaft (DFG) within SFB259.

References

1. T. Bönisch, R. Rühle, 'Portable Parallelization of a 3-D Flow-Solver' in Emerson et. al. (Eds.) 'Parallel Computational Fluid Dynamics' Recent Developments and Advances Using Parallel Computers, North-Holland, 1998, pp. 457-464
2. OpenMP Fortran Application Program Interface Version 2.0, November 2000, Accessible via http://www.openmp.org/specs/
3. MPI: A Message-Passing Interface Standard, Message Passing Interface Forum, June 1995 http://www.mpi-forum.org/docs/docs.html
4. R.B. Ciotti, J.R. Taft, J.Petersohn: Early Experiences with the 512 Processor Single System Image Origin2000, Proceedings of the 42nd CUG Conference (CUG Summit 2000), 2000 (CDROM)
5. Schöll, E., Frühauf, H.-H.: An Accurate and Efficient Implicit Upwind Solver for the Navier-Stokes Equations in Notes on Numerical Fluid Mechanics, Numerical Methods for the Navier-Stokes Equations, Hebecker F.-K., Ranacher R., Wittum G. (Eds.), Proceedings of the International Workshop on Numerical Methods for the Navier-Stokes Equations, Heidelberg, Germany, October 1993, Vieweg, 1994.
6. Frühauf, H.-H., Daiß, A., Gerlinger, U., Knab, O., Schöll, E.: Computation of Reentry Nonequilibrium Flows in a Wide Altitude and Velocity Regime, AIAA Paper 94-1961, June 1994.
7. Gerlinger, U., Frühauf, H.-H., Bönisch, T. : Implicit Upwind Navier-Stokes Solver for Reentry Nonequilibrium Flows, 32nd Thermophysics Conference, AIAA 97-2547.
8. F.S. Lien, W.L. Chen and M.A. Leschziner: A Multiblock Implementation of a Non-Orthogonal, Collocated Finite Volume Algorithm for Complex Turbulent Flows, International Journal for Numerical Methods in Fluids, Vol. 23, pp. 567-588, 1996
9. H.M. Bleecke et al: 'Benchmarks and Large Scale Examples' in A. Schüller 'Portable Parallelization of Industrial Aerodynamic Applications (POPINDA)', Notes on Numerical Fluid Mechanics, Volume 71, Vieweg, 1999
10. C. Walshaw, M. Cross and M. Everett, 'Parallel Dynamic Graph Partitioning for Adaptive Unstructured Meshes', Journal of Parallel and Distributed Computing, 47(2), pp. 102-108, 1997
11. Hofhaus, J.: Entwicklung einer effizienten, parallelisierten und impliziten Lösung der Navier-Stokes Gleichungen für dreidimensionale, inkompressible und instationäre Strömungen, Master Thesis, RWTH Aachen, Germany, 1993

About the Authors

Thomas Boenisch is research associate at the High-Performance Computing Center Stuttgart (HLRS), Germany. His research interests include Parallel CFD, benchmarking of current high performance platforms and high performance (parallel) I/O. Thomas can be reached at the HLRS, Allmandring 30, D-70550 Stuttgart, Germany, E-Mail: Boenisch@hlrs.de.

Panagiotis Adamidis is research associate at the High-Performance Computing Center Stuttgart (HLRS), Germany. His research interests include Parallel Multiphysics and Parallel CFD. Panos can be reached at the HLRS, Allmandring 30, D-70550 Stuttgart, Germany, E-Mail: Adamidis@hlrs.de.

Prof. Dr. – Ing. Roland Rühle is the director of the Computing Center of the University of Stuttgart (RUS) and the HLRS. Prof. Rühle can be reached at RUS, Allmandring 30, D-70550 Stuttgart, Germany, E-Mail: Ruehle@rus.uni-stuttgart.de