To be published in: Notes on Numerical Fluid Mechanics, Vieweg 1994

Flow simulation with FEM on massively parallel systems

Frank Lohmeyer, Oliver Vornberger

Department of Mathematics and Computer Science
University of Osnabrück, D-49069 Osnabrück
E-Mail:lohmey@informatik.uni-osnabrueck.de

Jan 1994

Summary

An explicit finite element scheme based on a two step Taylor-Galerkin algorithm allows the solution of the Euler and Navier-Stokes Equations for a wide variety of flow problems. To obtain useful results for realistic problems one has to use grids with an extremely high density to get a good resolution of the interesting parts of a given flow. Since these details are often limited to small regions of the calculation domain, it is efficient to use unstructured grids to reduce the number of elements and grid points. As such calculations are very time consuming and inherently parallel the use of multiprocessor systems for this task seems to be a very natural idea. A common approach for parallelization is the division of a given grid, where the problem is the increasing complexity of this task for growing processor numbers. Here we present some general ideas for this kind of parallelization and details of a Parix implementation for Transputer networks. Results for up to 1024 processors show the general suitability of our approach for massively parallel systems.

Introduction

The introduction of the computer into engineering techniques has resulted in the growth of a completely new field termed *computational fluid dynamics*. This field has led to the development of new mathematical methods for solving the equations of fluid mechanics. These improved methods have permitted advanced simulations of flow phenomena on the computer for a wide variety of applications. This leads to a demand for computers which can manage these extremely time consuming calculations within acceptable runtimes. Many of the numerical methods used in computational fluid dynamics are inherently parallel, so that the appearance of parallel computers makes them a promising candidate for this task.
One problem arising when implementing parallel algorithms is the lack of standards both on the hardware and software side. As things like processor topology, parallel operating

system, programming languages, etc. have a much greater influence on parallel than on sequential algorithms, one has to choose an environment where it is possible to get results which can be generalized to a larger set of other environments. We think that future supercomputers will be massively parallel systems of the MIMD class with distributed memory and strong communication capabilities. On the software side we see two possible models: message passing or virtual shared memory – both of them integrated into standard programming languages.
In the CFD field there is another important point: the numerical methods for the solution of the given equations. As we are mainly computer scientists, we decided not to invent new mathematical concepts but to develop an efficient parallel version of an algorithm which was devoloped by experienced engineers for sequential computers and which is suitable for the solution of problems in the field of turbomachinery [1]. The hardware platforms which are availiable for us, are Transputer systems of different sizes, which fulfill the demands mentioned above (the only problem in the meantime is the weak performance of a single processor node). We think the model for parallelism should be message passing (at least until now). When we started we had to use an OCCAM environment [2], which of course was neither comfortable nor portable, later we changed to Helios [3], which definitely is not suitable for massively parallel algorithms with high communication demands. Here we will present an implementation using Parix. This programming model seems to be very close to a message passing standard which have to be set up in the near future.
The following two sections give a brief overview about the physical and mathematical foundations of the used numerical method, and an outline of the general parallelization strategy. The next section describes in detail some grid division algorithms which are a very important part for this kind of parallel algorithms, because they determine the load balancing between processors. Another section compares some speedup results for different parameter settings, while the last section closes with a conclusion and suggestions for further research.

Foundations

This section gives a brief description of the equations which are necessary for the parallel flow calculations. For a detailed description see [4], [5] and [6].
For our flow calculations on unstructured grids with the finite element method we use Navier-Stokes Equations for viscous flow and Euler Equations for inviscid flow. The Navier-Stokes (Euler) Equations can be written in the following form,

$$\frac{\partial U}{\partial t} + \frac{\partial F}{\partial x} + \frac{\partial G}{\partial y} = 0, \tag{1}$$

where U, F and G are 4-dimensional vectors. U describes mass, impulses and energy, F and G are flow vectors. The flow vectors are different for the Euler and Navier-Stokes equations, in the latter case we have to add two equations to close the system.
The solution of these differential equations is calculated with an explicit Taylor-Galerkin two step algorithm. Therefore, at first a Taylor series in time is developed, which looks like

$$U^{n+1} = U^n + \Delta t \frac{\partial U^n}{\partial t} + \frac{\Delta t^2}{2}\frac{\partial^2 U^n}{\partial t^2} + O(\Delta t^3), \tag{2}$$

and in other form

$$U^{n+1} - U^n = \Delta U = \Delta t \frac{\partial}{\partial t}\left(U^n + \frac{\Delta t}{2}\frac{\partial U^n}{\partial t}\right) + O(\Delta t^3). \tag{3}$$

The expression in parenthesis can be seen as

$$U^{n+1/2} = U^n + \frac{\Delta t}{2}\frac{\partial U^n}{\partial t}. \tag{4}$$

If we take no consideration of the $O(\Delta t^3)$-term from equation (3) we achieve

$$\Delta U = \Delta t \frac{\partial}{\partial t} U^{n+1/2}. \tag{5}$$

With equation (1) and a replacement of the time derivation of equation (4) and (5) the two steps of the Taylor-Galerkin algorithm are:

$$U^{n+1/2} = U^n - \frac{\Delta t}{2}\left(\frac{\partial F^n}{\partial x} + \frac{\partial G^n}{\partial y}\right) \tag{6}$$

and

$$\Delta U = -\Delta t \left(\frac{\partial F^{n+1/2}}{\partial x} + \frac{\partial G^{n+1/2}}{\partial y}\right). \tag{7}$$

The differential equations can be expressed in a weighted residual formulation using triangular finite elements with linear shape functions [7]. Therefore, in the first step the balance areas of the convective flows for one element have to be calculated on the nodes of each element. In the second step the balance area for one node is calculated with the help of all elements which are defined with this node. A pictorial description of these balance areas of the two steps is given in figure 1.

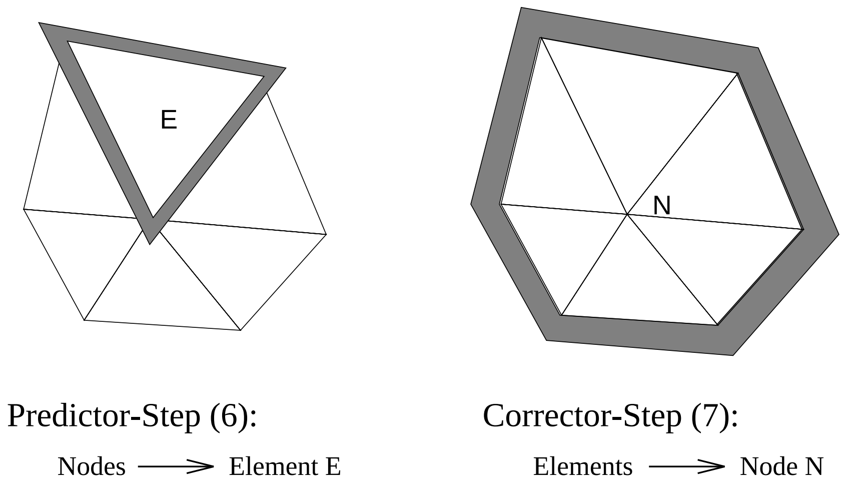

Figure 1: Balance areas

The calculation with the finite element method, which divides the calculation area into triangles, leads to the characteristic summation of the element matrices into the global mass matrix M and to the following equation system

$$M\,\Delta U = \Delta t\, R_S(U^n), \tag{8}$$

where R_S is the abbreviation for the summation of the right hand sides of equations (7) for all elements. The inversion of the Matrix M is very time consuming and therefore we use, with the help of the so called lumped mass matrix M_L, the following iteration steps:

$$\Delta U^0 = \frac{\Delta t\, R_S}{M_L}, \tag{9}$$

$$\Delta U^{\nu+1} = \Delta U^{\nu} + \frac{\Delta t\, R_S - M\Delta U^{\nu}}{M_L}. \tag{10}$$

For the determination of ΔU three iteration steps are sufficient. If we consider stationary flow problems only the initial iteration has to be calculated.
The time step Δt must be adjusted in a way where the flow of information does not exceed the boundaries of the neighbouring elements of a node. This leads to small time steps if instationary problems are solved (in the case of stationary problems we use a local time step for each element). In both cases the solution of a problem requires the calculation of many time steps, so that the steps (6), (7), (9) and (10) are carried out many times for a given problem. The resulting structure for the algorithm is a loop over the number of time steps, where the body of this loop consists of one or more major loops over all elements and some minor loops over nodes and boundaries (major and minor in this context reflects the different runtimes spent in the different calculations).

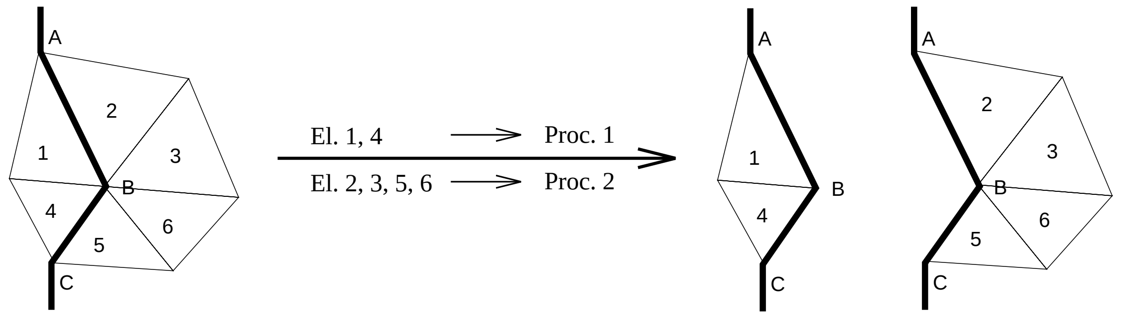

Figure 2: Grid division

Another important characteristic of this method is the use of unstructured grids. Such grids are characterized by various densities of the finite elements for different parts of the calculation area. The elements of an unstructured grid differ in both size and number and orientation of adjacent elements, which can result in a very complex grid topology. This fact is one main reason for the difficulties arising in constructing an efficient parallel algorithm.

Parallelization

If we are looking for parallelism in this algorithm we observe, that the results of one time step are the input for the next time step, so the outer loop has to be calculated in sequential

order. This is not the case for the inner loops over elements, nodes and boundaries which can be carried out simultaniously. So the basic idea for a parallel version of this algorithm is a distributed calculation of the inner loops. This can be achieved by a so called grid division, where the finite element grid is partitioned into sub grids. Every processor in the parallel system is then responsible for the calculations on one of these sub grids. Figure 2 shows the implication of this strategy on the balance areas of the calculation steps.
The distribution of the elements is non overlapping, whereas the nodes on the border between the two partitions are doubled. This means that the parallel algorithm carries out the same number of element based calculations as in the sequential case, but some node based calculations are carried out twice (or even more times, if we think of more complex divisions for a larger number of processors). Since the major loops are element based, these strategy should lead to parallel calculations with nearly linear speedup. One remaining problem is the construction of a global solution out of the local sub grid solutions. In the predictor step the flow of control is from the nodes to the elements, which can be carried out independently. But in the corrector step we have to deal with balancing areas which are based on nodes which have perhaps multiple incarnations. Each of these incarnations of a node sums up the results from the elements of its sub grid, whereas the correct value is the sum over all adjacent elements. As figure 3 shows the solution is an additional communication step where all processors exchange the values of common nodes and correct their local results with these values.

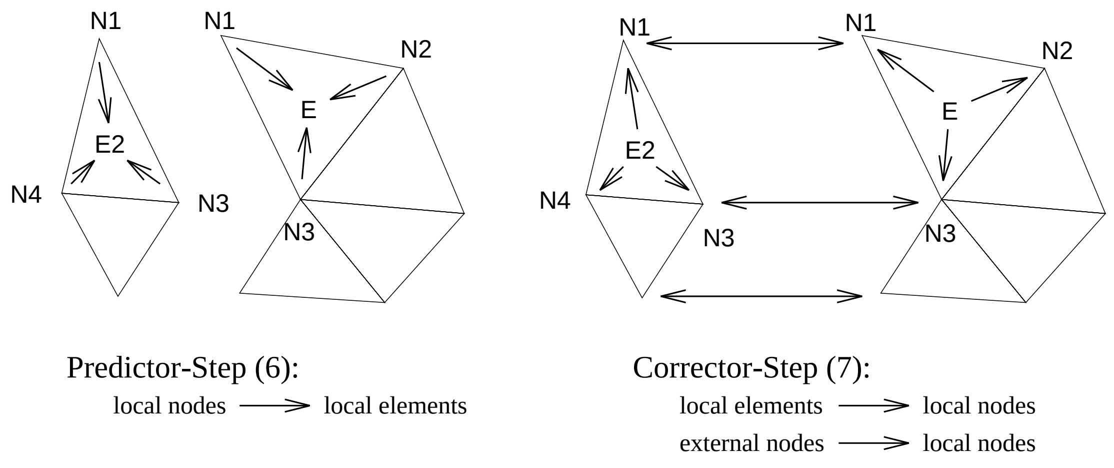
N1
N1
N2
E2
E
N4
N3
N3
N1
N1
N2
E
E2
N4
N3
N3
Predictor-Step (6):
local nodes ⟶ local elements
Corrector-Step (7):
local elements ⟶ local nodes
external nodes ⟶ local nodes

Figure 3: Parallel calculations

This approach where the sequential algorithm is used on distributed parts of the data sets and where the parallel and the sequential version are arithmetically equivalent is usally described with the key word *data decomposition*. The structure of this algorithms implies a MIMD architecture and the locality of data is exploited best with a distributed memory system. This special algorithm has a very high communication demand, because in every time step for every element loop an additional communication step occurs. To obtain high efficiencies a parallel system with high communication performance is required and the programming model should be message passing. Our current implementation is for Transputer systems and uses the Parix programming environment, which supplies a very flexible and comfortable interprocessor communication library. This is necessary if we think of unstructured grids which have to be distributed over large processor networks leading to very complex communication structures.

Grid division

If we now look at the implementation of the parallel algorithm, two modules have to be constructed. One is the algorithm running on every processor of the parallel system. This algorithm consists of the sequential algorithm operating on a local data set and additional routines for the interprocessor communication. These routines depend on the general logical processor topology, so that the appropriate choice of this parameter is important for the whole parallel algorithm. In Parix this logical topology has to be mapped onto the physical topology which is realized as a two-dimensional grid (this statement only holds for T8-systems, in the future this topology will change to a three-dimensional grid). For two-dimensional problems there are two possible logical topologies: one-dimensional pipeline and two-dimensional grid. They can be mapped in a canonical way onto the physical topology, so that we have implemented versions of our algorithm for both alternatives.

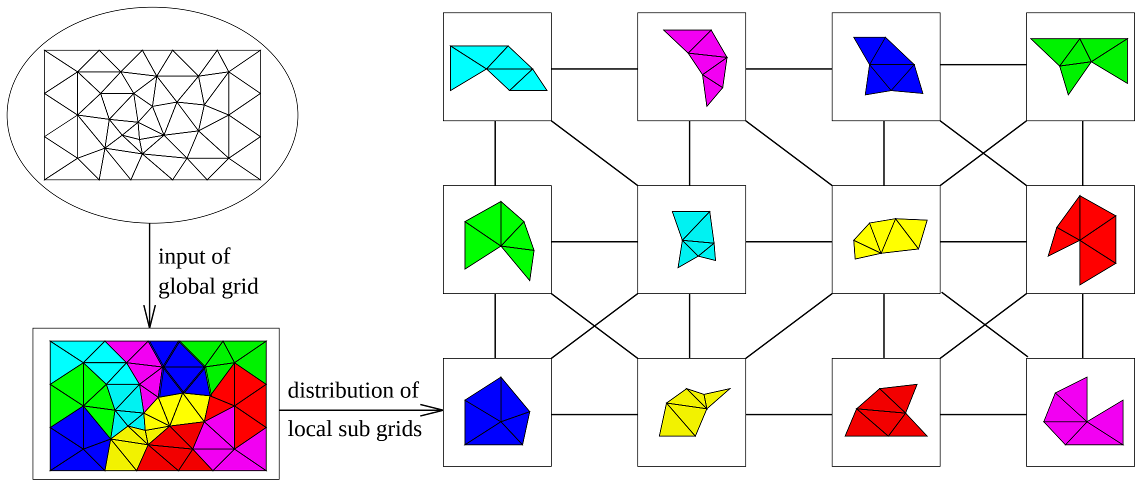

Figure 4: Decomposition algorithm

The second module we had to implement is a decomposition algorithm which reads in the global data structures and calculates a division of the grid and distributes the corresponding local data sets to the appropriate processor of the parallel system. The whole algorithmic structure is shown in figure 4, where we can also see that a given division often requires interprocessor connections which are not supplied by the basic logical topology. These connections are built dynamically with the so called virtual links of Parix and collected in a virtual topology.
The essential part of the whole program is then the division algorithm which determinates the quality of the resulting distribution. This algorithm has to take different facts into consideration to achieve efficient parallel calculations. First it must ensure that all processors have nearly equal calculation times, because idle processors slow down the speedup. To achieve this it is necessary first to distribute the elements as evenly as possible and

then minimize the overhead caused by double calculated nodes and the resulting communications. A second point is the time consumed by the division algorithm itself. This time must be considerably less than that of the actual flow calculation. Therefore we cannot use an optimal division algorithm, because the problem is NP-complete and such an algorithm would take more time than the whole flow calculation. For this reason we have to develop a good heuristic for the determination of a grid division. This task is mostly sequential and as the program has to deal with the whole global data sets we decided to map this process onto a workstation outside the Transputer system. Since nowadays such a workstation is much faster than a single Transputer, this is no patchedup solution, but the performance of the whole calculation even increases.
According to the two versions for the parallel module, we also have implemented two versions for the division algorithm. Since the version for a one-dimensional pipeline is a building block for the two-dimensional case, we present this algorithm first:

```
Phase 0:   calculate element weights
           calculate virtual coordinates

Phase 1:   find element ordering with small bandwidth
           a)  use virtual coordinates for initial ordering
           b)  optimize bandwidth of ordering

Phase 2:   find good element division using ordering and weights

Phase 3:   optimize element division using communication analysis
```

The division process is done in several phases here: an initialization phase (0) calculates additional information for each element. The weight of an element represents the calculation for this special element type (these times are varying because of special requirements of e.g. border elements). The virtual coordinates reflect the position where in the processor topology this element should roughly be placed (therefore the dimension of this *virtual space* equals the dimension of the logical topology). These virtual coordinates (here it is actually only one coordinate) can be derived from the real coordinates of the geometry or from special relations between groups of elements. An example for the latter case are elements belonging together because of the use of a turbulence model (we are using the Baldwin & Lomax model here). In this case nodes on a line in normal direction to a surface are strongly coupled by the type of calculations used in this model and so elements can be given virtual coordinates which reflect these relations.
Before the actual division an ordering of the elements with a small bandwidth is calculated (phase 1). This bandwidth is defined as the maximum distance (that is the difference of indices in the ordering) of two adjacent elements. A small bandwidth is a requirement for the following division step. Finding such an ordering is again a NP-complete problem, so we can not get an optimal solution. We use a heuristic, which calculates the ordering in two steps. First we need a simple method to get an initial ordering (a). In our case we use a sorting of elements according to their virtual coordinates. In the second step (b) this ordering is optimized e.g. by exchanging pairs of elements if this improves the bandwidth until there is no more exchanging possible.

With the received ordering and the element weights the actual division is now calculated. First the elements are divided into ordered parts with equal weights (phase 2). Then this division is analysed in terms of resulting borders and communications and is optimized by reducing border length and number of communication steps by exchanging elements with equal weights between two partitions (phase 3).
If we now want to construct a division algorithm for the two-dimensional grid topology we can use the algorithm described above as a building block. The resulting algorithm has the following structure:

```
Phase 0 (initialization) similar to 1D-algorithm

for #processors in X-dimension do

    calculate meta-division M
    using phases 1 and 2 of 1D-algorithm

    divide meta-division M in y-dimension
    using phases 1 and 2 of 1D-algorithm

Phase 3 (optimization) similar to 1D-algorithm
```

The only difference between the one and the two-dimensional version of the initialization phase is the number of virtual coordinates which here of course is two. Phase 3 has the same task which is much more complex in the two-dimensional case. The middle phase here is a two stage use of the one-dimensional strategy, where the grid is first cut in the X-dimension and then all pieces are cut in the Y-dimension. This strategy can be substituted by a sort of recursive bisectioning, where in every step the grid is cut into two pieces in the larger dimension and both pieces are cut further using the same strategy.

Results

The algorithms described in the previous chapter were tested with a lot of different grids for various flow problems. As a kind of benchmark problem we use the instationary calculation of inviscid flow behind a cylinder, resulting in a vortex street. One grid for this problem was used for all our implementations of the parallel calculations. This grid has a size of about 12 000 grid points which are forming nearly 20 000 elements.
In figure 5 the speedups for some parameter settings are shown. The system where these measurements were made is a 1024 processor system located at the University of Paderborn. It consists of T805 Transputers, each of them equipped with 4 MByte local memory and coupled together as a two-dimensional grid. Our algorithms are all coded in Ansi-C using the Parix communication library.
In the speedup curves the difference between the logical topologies 1D-pipeline and 2D-grid is shown. In the left part of the picture we can see that for up to 256 processors we achieve nearly linear speedup with the grid topology, whereas the pipe topology is only linear for a maximum of 128 processors.

If we increase the number of processors like in the right half we observe that the grid topology again is superior to the pipe topology, but the increase of speedup is no longer linear. It is a common problem for most parallel algorithms, that for a fixed problem size there is always a number of processors where the speedup is no longer increasing proportionally to the number of processors. But here the gap between the actual and the theoretically possible speedup for 1024 processors seems to be too large.

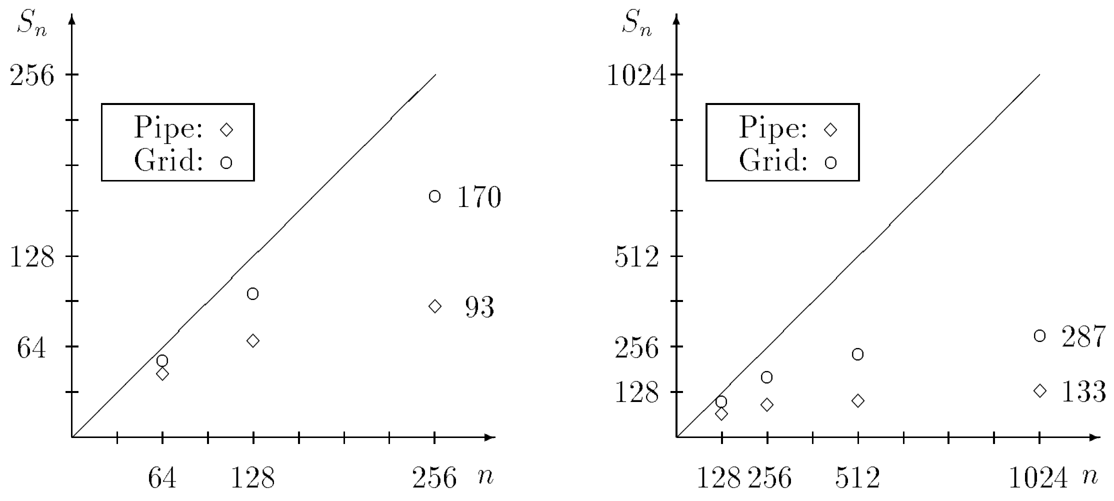

Figure 5: Speedups for different topologies

We had measured speedups for this grid in the older Helios version where we achieved a speedup of 86 (using 256 processors) for the pipe topology (the grid topology was first implemented under Parix). For a larger number of processors the Helios system was not availiable, even for 256 processors it was not able to execute the program every time. The Parix results can be slightly increased for the grid topology if we use the recursive bisection method for the grid division: on 1024 processors the speedup is then 307.

Conclusion

In this paper we have introduced a parallelization for the calculation of fluid flow problems on unstructured grids. An existing sequential algorithm has been adjusted for Transputer systems under Parix and investigations on the parallelization of this problem have been made. For two logical processor topologies we have developed different grid division algorithms and compared them for some benchmark problems. The grid topology has shown its superiority over the pipe topology. This was expected since a two-dimensional topology must be better suited for two-dimensional grids than a one-dimesional topology which is not scalable for large processor numbers. The speedup measurements on a 1024 Transputer cluster showed the general usefulness of the choosen approach for massively parallel systems, but also the limits of the current implementation for large processor numbers can be seen.

We think that one general problem is the use of an a-priori division algorithm with an a-priori optimization. Such an algorithm must estimate all parameters of a distributed calculation before the program starts. Since all these estimations are inaccurate even an optimal solution of the divsion problem will not lead to an optimal parallel execution. A solution which is able to deal with this *fuzzy* cost functions is a dynamic optimization of a given a-priori division. Such an optimization can detect load imbalances of processors at runtime and a dynamic load balancer can correct these errors by a dynamic redistribution of elements.

Our further research will concentrate on an improvement of the parallelization of the used flow algorithm. The dynamic load balancer described above will be implemented in the near future. The use of this concept has a lot of other advantages: As the a-priori division is only used as a starting point for the further balancing, a very simple algorithm can be used for this task which is faster than the actual algorithm. Dynamic load balancing is fully parallel and hardware independent, so that changes of the basic hardware nodes can be done without changing the developed algorithm. A very important last point is the parallelization of adaptive mesh refinement, where a dynamic load balancer can be used as an important building block.

References

[1] Armin Vornberger. *Strömungsberechnung auf unstrukturierten Netzen mit der Methode der finiten Elemente.* Ph.D. Thesis, RWTH Aachen, 1989

[2] F. Lohmeyer, O. Vornberger, K. Zeppenfeld, A. Vornberger. *Parallel Flow Calculations on Transputers.* International Journal of Numerical Methods for Heat & Fluid Flow, Vol. 1, pp. 159-169, 1991

[3] F. Lohmeyer, O. Vornberger. *CFD with parallel FEM on Transputer Networks.* Flow Simulation with High-Performance Computers I, Notes on Numerical Fluid Mechanics, Vol. 38, pp. 124-137, Braunschweig 1993

[4] W. Koschel, M. Lötzerich, A. Vornberger. *Solution on Unstructured Grids for the Euler- and Navier-Stokes Equations.* AGARD Symposium Validation of Computational Fluid Dynamics, Lisbon, May 1988

[5] W. Koschel, A. Vornberger. *Turbomachinery Flow Calculation on Unstructured Grids Using the Finite Element Method.* Finite Approximations in Fluid Mechanics II, Notes on Numerical Fluid Mechanics, Vol. 25, pp. 236-248, Aachen, 1989

[6] W. Koschel, A. Vornberger, W. Rick. *Engine Component Flow Analysis Using a Finite Element Method on Unstructured Grids.* Institute for Jet Propulsion and Turbomachinery, Technical University of Aachen

[7] R. Peyret, T.D. Taylor. *Computation Methods for Fluid Flow.* Springer Series in Computational Physics, Berlin, 1983