Flow simulation with an adaptive finite element method on massively parallel systems

Frank Lohmeyer, Oliver Vornberger
University of Osnabrück, D-49069 Osnabrück, Germany
lohmey@informatik.uni-osnabrueck.de

Abstract. 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. To improve the quality of the calculated solutions an adaptive grid refinement procedure was included. This extension leads to the necessity of a dynamic load balancing for the parallel version. An effective strategy for this task is presented and results for up to 1024 processors show the general suitability of our approach for massively parallel systems.

1 Introduction

The introduction of the computer into engineering techniques has resulted in the growth of a completely new field termed *computational fluid dynamics* (CFD). 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 it seems that some standards could be established in the near future. As our algorithm is designed for message passing environments, this standard might be MPI (Message Passing Interface) [6].

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 developed 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. At the time the algorithm was developed, there was no MPI-environment availiable for our transputer systems, so here we will present a version using Parix (Parallel extensions to Unix, a parallel runtime system for Parsytec machines) that needs only a small number of parallel routines, which are common in most message passing environments. Most of these routines are hidden inside a few communication procedures, so that they can be replaced easily, when changing the parallel environment.

The following two sections give a brief overview about the physical and mathematical foundations of the used numerical methods (for a detailed description see [1, 2]) and an outline of the general parallelization strategy, including a comparison with other approaches (see also [3, 4]). 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. The special subjects of adaptive refinements and dynamic load balancing are discussed in a separate section. Then some results will be presented, while the last section closes with a conclusion and suggestions for further research.

2 Foundations

This section gives a brief description of the equations which are necessary for the parallel flow calculations.

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 both cases 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. 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.

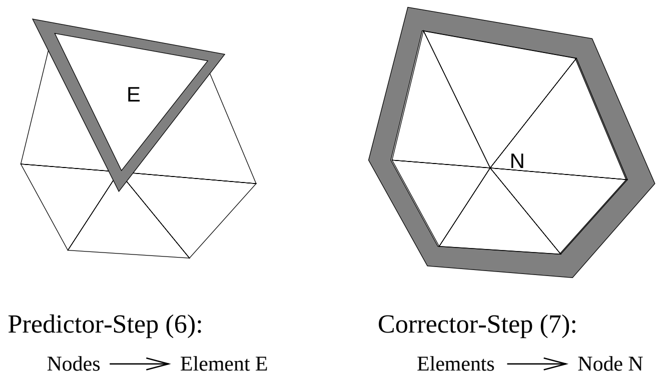

Figure 1: Balance areas

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).

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 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.

The main advantage of unstructured grids is their ability to adapt a given flow. To get a high resolution of the details of a flow, the density of the grids must only be increased in the interesting parts of the domain. This leads to a very efficient use of a given number of elements. One problem arising in this context is the fact that in most cases the details of a flow are the subject of investigations, so that it is impossible to predict the exact regions, where the density of the grid has to be increased. A solution of this problem is a so called adaptive grid refinement, where the calculations start with a grid with no or little refinements. As the calculations proceed, it is now possible to detect regions, where the density of the grid is not sufficient. These parts of the grid will then be refined and the calculations proceed with the refined grid. These refinement step is repeated until the quality of the solution is sufficient.

3 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.

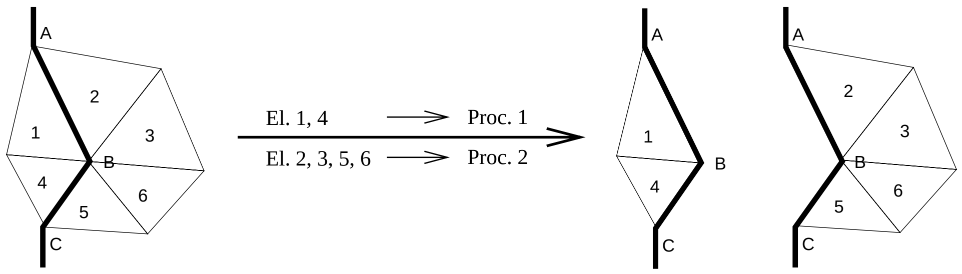

Figure 2: Grid division

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, this 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.

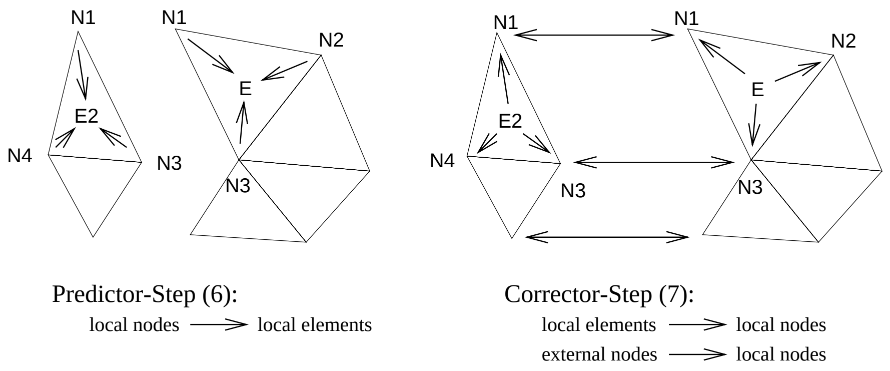
N1
N1
N2
E
E2
N4
N3
N3
N1
N1
N2
E
E2
N4
N3
N3
Predictor-Step (6):
local nodes → local elements
Corrector-Step (7):
local elements → local nodes
external nodes → local nodes

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 usually described with the key word *data decomposition*. Other *domain decomposition* approaches have to deal with numerically different calculations in different parallelel cases, and have to pay special attention to numerical stability. In the case of implicit algorithms it is common to make a division of the grid nodes, due to the structure of the resulting system of linear equations, which have to be solved in parallel. The main advantage of the explicit algorithm used here is the totally local communication structure, which results in a higher parallel efficiency, specially for large numbers of processors.

This structure implies a MIMD architecture and the locality of data is exploited best with a distributed memory system together with a message passing environment. This special algorithm has a very high communication demand, because in every time step for every element loop an additional communication step occurs. An alternative approach in this context is an overlapping distribution, where the subgrids have common elements around the borders (see [5]). This decreases the number of necessary communications but leads to redundant numerical calculations. We decided to use non-overlapping divisions for two reasons: First they are more efficient for large numbers of subgrids (and are therefore better suited for massively parallel systems), and the other reason is that we want to use adaptive grids. The required dynamic load balancing would be a much more difficult task for overlapping subgrids. The only drawback of our approach is that to obtain high efficiencies a parallel system with high communication performance is required, so it will not work e.g. on workstation clusters. 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.

4 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. 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.

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 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

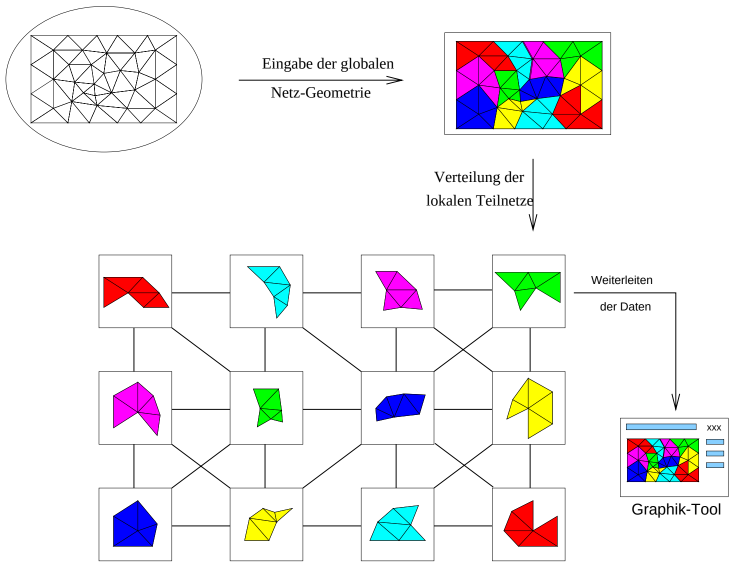

Figure 4: Decomposition algorithm

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 time 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 periodic borders. In this case nodes on opposite sides of the calculation domain are strongly coupled and this fact should be reflected in the given virtual coordinates.

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.

5 Adaptive refinement and dynamic optimization

The parallelization approach described in the last two sections is well suited for fixed grids, which remain constant through all calculation steps. We will now introduce a simple, but effective method for an adaptive grid refinement and an improvement of the parallel algorithm which takes into account that the work load for each processor has changed after every refinement step. Before we can describe the algorithms, some questions have to be answered: what does refinement mean exactly, which parts should be refined, and how can we construct the new, refined grid.

- Refinement in our case means the splitting of elements into smaller elements, which replace the original elements.
- The question, which parts should be refined now turns into the selection of elements that should be refined. Therefore we choose for a given flow problem a characteristic function, e.g. the pressure field. We then look for elements, where the gradient of this function exceeds a given bound.

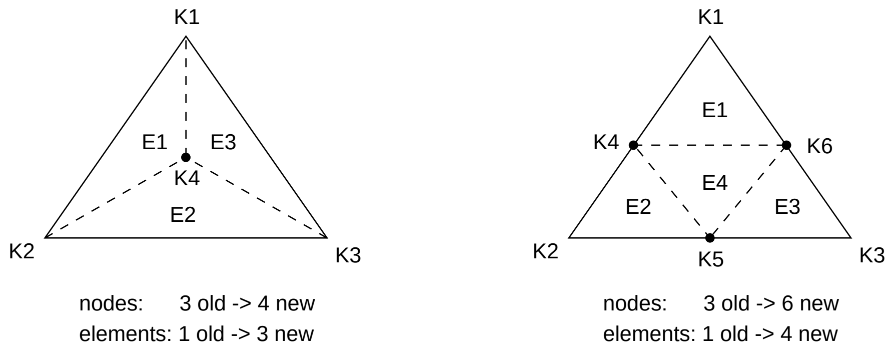

Figure 5: Splitting of an element

- How should these elements be split into smaller ones? In figure 5 two possibilities are shown: Using the left alternative leads to numerical problems, caused by the shape of the resulting triangles. Especially if an element is refined several times, the new grid nodes will be placed near to the remaining sides leading to very flat triangles, which should be avoided. So we must use the right alternative, which leads to the problem that the new nodes are placed on the edges of a triangle. This would result in an inconsistent grid, because all nodes must be corners of elements. The solution of this problem is an additional splitting of elements with such edges. Elements with two or three refined edges must be split into four elements (as if they where originally selected for refinement). Elements with only one refined edge must be split into two elements in a canonical way. This additional splitting of elements can lead to new elements with nodes on their edges, so the process has to be repeated until no more splitting is necessary.

- How can we construct the new grid? Every split element will be replaced by one of the new elements, the remaining new elements will be added to the element list. All new nodes are added to the node list and all new border nodes and elements are added to the appropriate lists. A new local time step for every element must be calculated and in the case of an instationary solution a new global time step must be calculated from the local time steps. After this some derived values like element sizes have to be reinitialized and then the calculations can continue.

Now we can formulate the refinement step:

```
calculate reference function F
Delta = (Max(F) - Min(F)) * refine_rate

for (El in element-list)
   dF = local gradient of F in El

   if (dF > Delta)
      mark El with red
   else
      mark El with white
   end-if

end-for

mark all elements with nodes on edges:
   with yellow for full refinement
   with green for half refinement

refine all elements full or half according to their colour
construct new node- and element-lists
reinitialize all dependent variables
```

All steps can be implemented straightforward with one exception: the colouring of the elements that must be refined to get a consistent grid. This can be done efficiently with the following recursive algorithm:

```
for (El in element-list)

   if (El is marked red)
      mark_neighbours(El)
   end-if

end-for
```

```
mark_neighbours(El):

   for (E in neighbour-elements(El))
      if (E is marked white)
         mark E with green
      else if (E is marked green)
         mark E with yellow
         mark_neighbours(E)
      end-if
   end-for

end-mark_neighbours
```

To insert this refinement step in the parallel algorithm, we have to analyse the different parts for parallelism:

```
calculate reference function F
    local data, fully parallel, no communication
Delta = (Max(F) - Min(F)) * refine_rate
    local data, global min/max, mostly parallel, global communication
mark elements with red or white
    local data, fully parallel, no communication
mark additional elements with yellow or green
    global data, sequential, global communication (collection)
refinement of elements
    global data, sequential, global communication (broadcast)
construct new node- and element-lists
    local data, fully parallel, no communication
reinitialize all dependent variables
    local data, fully parallel, local communications
```

We can see that most of the parts can be performed in parallel with no or little communication. The only exception is the additional colouring of elements and the construction of new elements and nodes. These parts operate on global data structures, so that a parallel version of them must lead to a very high degree of global communication. As these parts need only very little of the time spent on the complete refinement step, we decided to keep this parts sequential. All necessary data is collected from one processor, the two steps are processed, and the resulting data is broadcasted to the appropriate processors.

For a typical flow calculation up to five refinement steps are sufficient in most cases, so that the lack of parallelism in the refinement step decribed above is not very problematic. Much more important is the fact that after a refinement step the work load of every processor has changed. As the refinement takes place in only small regions there are a few processors with a load that is much higher than the load of most of the other processors. This would not only slow down the calculations, but can lead to memory problems if further refinements in the same region take place. The solution of this problems is a dynamic load balancing, where parts of the sub grids are exchanged between processors until equal work load and nearly equal memory consumption is reached.

The load is obtained by simply measuring the CPU-time needed for one time step including communication times but excluding idle times. The items that could be exchanged between processors are single elements including their nodes and all the related data. To achieve this efficiently, we had to use dynamic data structures for all element and node data. There is one data block for every node and every element. These blocks are linked together in many different dynamic lists. The exchange of one element between two processors is therefore a complex operation: the element has to be removed from all lists on one processor and must be included in all lists on the other processor. Since nodes can be used on more than one processor, the nodes belonging to that element must not be exchanged in every case. It must be checked, whether they are already on the target processor and if other elements on the sending processor will need them, too. Nodes not availiable on the target processor must be sent to it and nodes which are no longer needed on the sending processor must be deleted there.

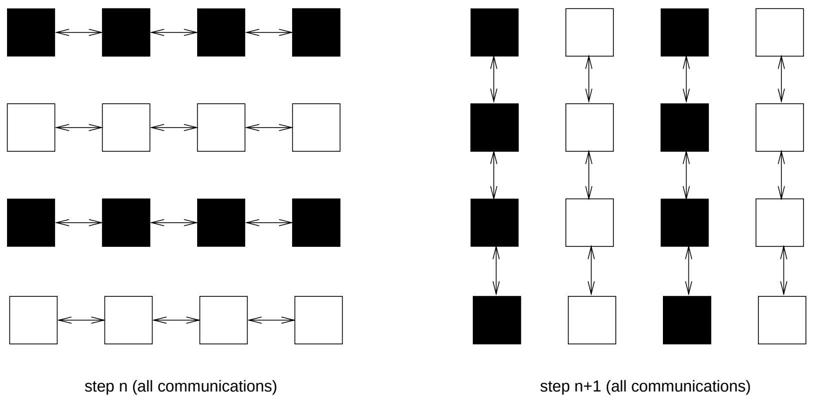

Figure 6: Areas for dynamic load balancing

One remaining problem is to find an efficient strategy for the exchange of elements. A good strategy should deliver an even load balance after only a few steps and every step should be finished in a short time. As elements can only be exchanged between direct neighbours, a first approach was a local exchange between pairs of processors. This results in fast exchange steps, but shows a bad convergence behaviour. A global exchange would converge very fast, but at the cost of the lack of any parallelism. We will present here a semi-global strategy, where the balancing is carried out along the rows and columns of the processor grid. As shown in figure 6, the balancing areas are alternating all rows and all columns, where every row (column) is treated independent from all other rows (columns). A single row (column) is interpreted as a tree with the middle processor as the root and then we can use a modification of a tree balancing algorithm developed for combinatorical optimization problems [7].

This algorithm uses two steps: in a first step information about the local loads of the tree is moving up to the root and the computed optimal load value is propagated down the tree. In a second step the actual exchange is done according to the optimal loads found in the first step. The first step has the following structure, where `num_procs` is the number of processors in the tree (= row or column) and `load_move_`*direction* is the load that has to be moved in that direction in the second step:

```
if (root)
   receive load_sub_l (load of left subtree)
   receive load_sub_r (load of right subtree)

   global_load = local_load + loads of subtrees
   load_opt = global_load / num_procs

   send load_opt to subtrees

   load_move_l = load_opt * num_sub_l - load_sub_l
   load_move_r = load_opt * num_sub_r - load_sub_r

else
   receive load_sub from subtree (if not leaf)
   load_sub_n_me += load_sub
   send load_sub_n_me to parent node

   receive load_opt from parent node
   send load_opt to subtree (if not leaf)

   load_move_sub = load_opt * num_sub - load_sub
   load_move_top = load_sub_n_me - (num_sub + 1) * load_opt
end-if
```

The second step is very simple: all processors translate the loads they have to move into the appropriate number of elements and exchange these elements. For the decision, which element to send in a specific direction, the virtual coordinates of this element are looked up and the element with the greatest value is choosen.

6 Results

The algorithms described in the previous chapters 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 (P1). Other problems used with the adaptive refinement procedure are stationary turbine flows with grids of different sizes, all of them using periodic borders.

All measurements were made with a 1024 processor system located at the $(PC)^2$ of 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.

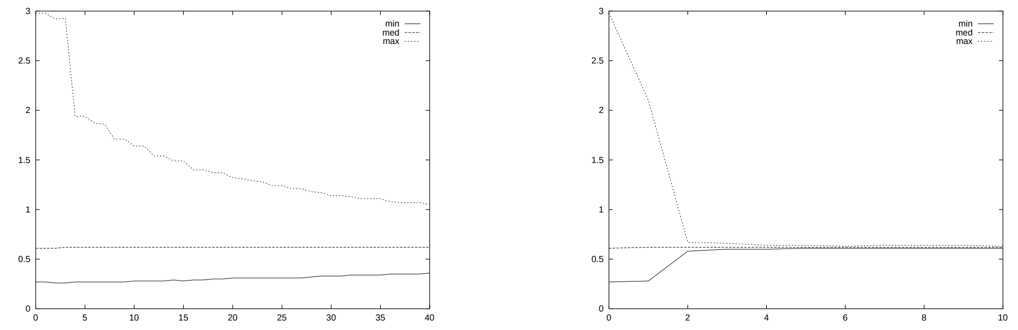

Figure 7: Different load balancing strategies

First we will present some results for the dynamic load balancing. Figure 7 shows the different convergence behaviour of two different strategies. To investigate this, we used a start division of our reference problem with an extremely bad load balancing. The left picture shows the results of a local strategy, where the balance improves in the first steps, but stays away from the optimum for a large number of optimization steps. The right picture shows the effects of the described semi-global balancing, where after two steps the balance is nearly optimal.

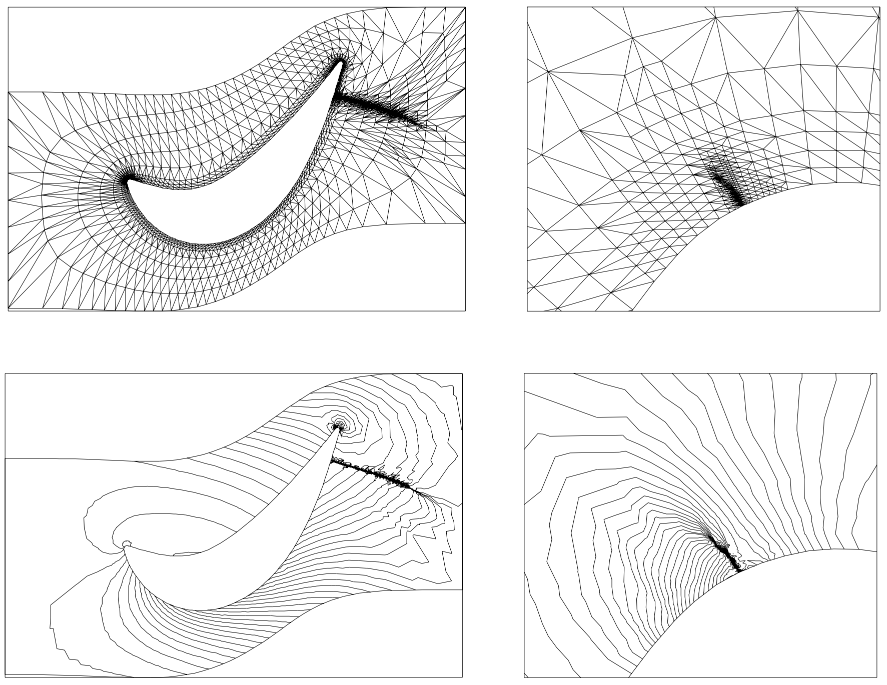

Figure 8: Grid and isobars after five refinement steps (complete and zoomed)

An example for the adaptive refinement procedure is shown in figure 8, where the grid and the pressure field around a turbine is shown after five refinement steps. One can see the high resolution of the two shocks made visible by the adaptive refinements. Without refinement one of these effects can only be guessed, the other is missing completely.

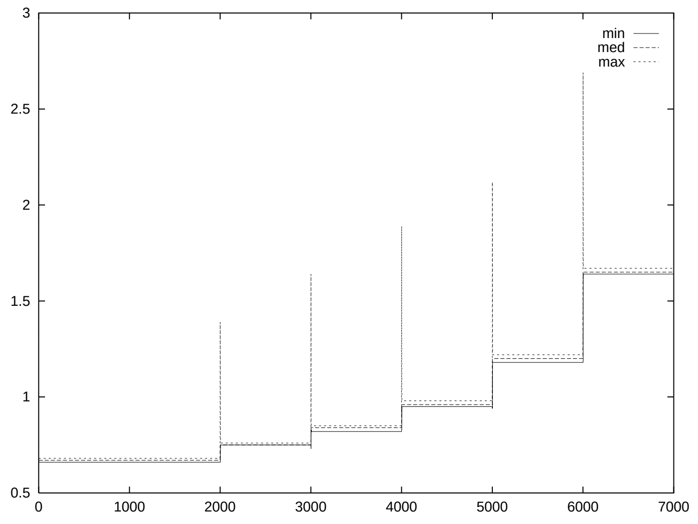

Figure 9: Load balancing for adaptive refinements

The development of the corresponding loads can be seen in figure 9. After each refinement step four balancing steps were carried out, using the semi-global strategy. The picture shows the efficient and fast balancing of this method.

At last we will present results for large processor numbers. In figure 10 the speedups for some parameter settings of our reference problem are shown. 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 size of the problem (P2), the speedups are closer to the theoretical values, which proofs the scalability of the parallelization approach.

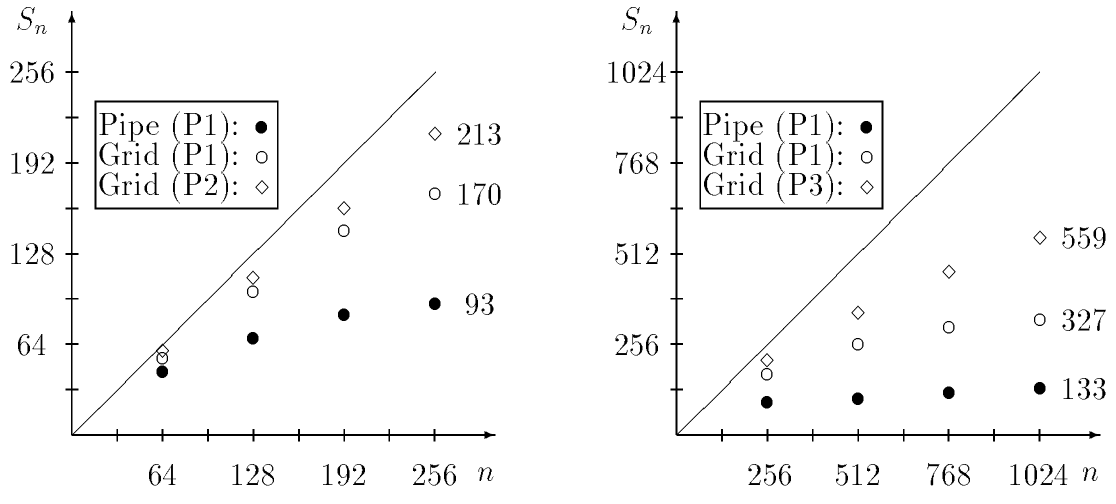

Figure 10: Speedups for different topologies

If we increase the number of processors (right picture) 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. If we want to get the same efficiencies as for 256 processors we have to use grids with approximately 50 000 elements. This was impossible on the used T8-system, because such problems are too large for it. The biggest problem that fits into the 4 Mbyte nodes has about 32 000 elements (P3), so that the speedups for 1024 processors are limited on this machine. Nevertheless, the increase of speedup to 559 shows the scalability of our algorithm again.

7 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-dimensional 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.

We presented an adaptive refinement procedure which is used for the solution of flow problems with a priori unknown local effects. For the parallel version of this procedure we showed the need for a dynamic load balancing. A semi-global strategy for this balancing was described in detail. We presented results for the performance of this strategy and compared it with a local strategy. We showed the excellent convergence behaviour of our strategy and the usefulness of the dynamic load balancing together with the adaptive refinement.

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. To exploit this advantage of our algorithms, they must be implemented as portable as possible. To achieve this our further research will concentrate on porting the current implementation to MPI and on investigations for different hardware platforms.

References

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

[2] 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

[3] 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

[4] F. Lohmeyer, O. Vornberger. *Flow Simulation with FEM on Massively Parallel Systems.* Computational Fluid Dynamics on Parallel Systems, Notes on Numerical Fluid Mechanics, Vol. 50, pp. 147-156, Braunschweig, 1995

[5] S. Lanteri. *Unstructured CFD Computations on M.I.M.D. Systems.* Computational Fluid Dynamics on Parallel Systems, Notes on Numerical Fluid Mechanics, Vol. 50, pp. 112-124, Braunschweig, 1995

[6] Message Passing Interface Forum. *MPI: A message-passing interface standard.* University of Tennessee, Knoxville, TN, 1994

[7] M. Böhm, E. Speckenmeier. *Effiziente Lastausgleichsalgorithmen.* Proceedings of TAT-94, Aachen, 1994