Strategies for Parallel and Numerical Scalability of CFD Codes

Ralf Winkelmann[*], *Jochem Häuser*[†]
Department of Transportation, University of Applied Sciences
and
Center of Logistics and Expertsystems GmbH
`http://www.cle.de`
Karl-Scharfenberg Straße 55-57
38229 Salzgitter, Germany

Roy D. Williams
Center of Advanced Computing Research
California Institute of Technology
Pasadena, USA

May 18, 1998

[*]This paper contains a major part of the PhD work of the first author.
[†]Paper to be published in special issue of Computer Methods in Applied Mechanics and Engineering on Parallel Computational Methods in Flow Simulation and Modeling, North-Holland, ed. Tayfun E. Tezduyar.

Contents

ABSTRACT

In this article we discuss a strategy for speeding up the solution of the Navier-Stokes equations on highly complex solution domains such as complete aircraft, spacecraft, or turbomachinery equipment. We have used a finite-volume code for the (non-turbulent) Navier-Stokes equations as a testbed for implementation of linked numerical and parallel processing techniques. Speedup is achieved by the **Tangled Web** of advanced grid topology generation, adaptive coupling, and sophisticated parallel computing techniques.

An optimized grid topology is used to generate an optimized grid: on the block level such a grid is unstructured whereas within a block a structured mesh is constructed, thus retaining the geometrical flexibility of the finite element method while maintaining the numerical efficiency of the finite difference technique. To achieve a steady state solution, we use grid-sequencing: proceeding from coarse to finer grids, where the scheme is explicit in time. Adaptive coupling is derived from the observation that numerical schemes have differing efficiency during the solution process. Coupling strength between grid points is increased by using an implicit scheme at the sub-block level, then at the block level, ultimately fully implicit across the whole computational domain. Other techniques include switching numerical schemes and the physics model during the solution, and dynamic deactivation of blocks. Because the computational work per block is very variable with adaptive coupling, especially for very complex flows, we have implemented parallel dynamic load-balancing to dynamically transfer blocks between processors. Several 2D and 3D examples illustrate the functioning of the Tangled Web approach on different parallel architectures.

1 *Introduction*

1.1 *Motivation*

Computational Fluid Dynamics is becoming increasingly sophisticated. Grids define highly complex geometries, and flows are solved involving very different length and time scales. The number of grid points, and thereby the number of degrees of freedom, is increasing as memory of supercomputers is growing.

During the last few years new developments in computer hardware and architectures have led to significant advances in parallel computing and multiprocessing. It is believed that parallel computing is the most important means of reducing turn around time and computational cost of large scale applications. Furthermore, massively parallel computing is considered to be the key technology in tackling the grand challenges facing CFD, such as multidisciplinary analysis and optimization.

One of the main issues in parallel CFD is the flow simulation past very complex configurations and the design of numerical algorithms which efficiently exploit the capabilities of the parallel hardware. Especially in the case of distributed memory machines comprising several hundred or even thousands of powerful processors, this is a non-trivial task. The important aspects in designing parallel algorithms for these architectures are partitioning of data (domain decomposition), communication across internal boundaries, as well as dynamic load balancing (see Section 3.4) and minimizing overhead caused by both communication and computation.

1.2 *Solution Methods*

Regarding the solution algorithm for the Navier-Stokes equations, an explicit algorithm is easy to implement, but is numerically less efficient than relaxation schemes in calculating a steady state solution. Often relaxation schemes are used, but it should be remembered that even these methods may not converge for highly stretched grids with large cell aspect ratios of e.g. 10^5 or even 10^6, as needed in many viscous flows.

More generally, it can be demonstrated that a single numerical scheme has varying numerical efficiency during the course of the solution process. The novel feature presented in this article is to apply a sequence of numerical strategies, called the **Tangled Web** approach (see Section 4). The objective is to use the optimal scheme at each stage of the solution process, switching (automatically) when certain criteria are met.

These numerical methods include grid sequencing, domain decomposition, adaptive coupling (moving from an explicit scheme, to block-implicit, fully implicit, to Newton's method), use of iterative solvers such as CG-GMRES, along with line-searching and backtracking for root polishing.

The utilization of the outlined strategy attempts to achieve both **parallel**

and numerical scalability. While the former concept concerns good scaling of solution time with the number of processors, the latter concerns good scaling with the problem size. This is, however, generally not encountered in practice, for example the inversion of a matrix of size N elements, needs $O(N^3)$ floating point operations. Obviously, no parallel architecture could keep pace with this computational demand – when problem size scales, we need more sophisticated algorithms to provide the numerical scalability that we seek.

It turns out that the combination of parallel computing and advanced solving strategies is essential for the development of efficient CFD codes that will serve as design tools for aerodynamic shape optimization and multidisciplinary analysis – not just a single "magic bullet" strategy, but a collection of algorithms, each applied where appropriate. In this article, we show this approach is effective for a number of test-cases, manually switching between numerical strategies. In the future we will report on criteria for automatically switching over to a different numerical scheme as soon as the current scheme becomes numerically inefficient.

2 *The Navier–Stokes Equations*

The equations to be solved on highly complex geometries are the *Navier–Stokes* equations that read in integral form

$$\frac{\partial}{\partial t}\int_V \mathbf{U}\; dV + \oint_{A(V)} \mathbf{F}\bullet\, d\mathbf{A} = 0, \tag{1}$$

where $\mathbf{U}$ is the vector of flow variables and the tensor $\mathbf{F}$ denotes fluxes without reference to a particular coordinate system. A flux can be considered as a vector of 3 quantities that each comprise a vector of 5 variables, the so called flux components. In a Cartesian coordinate system, the vector of the flow variables and the flux vector are denoted as

$$\hat{\mathbf{U}} = (\rho, \rho u, \rho v, \rho w, \rho e)^T \tag{2}$$

$$\hat{\mathbf{F}} := \left(\hat{\mathbf{F}}, \hat{\mathbf{G}}, \hat{\mathbf{H}}\right) \tag{3}$$

and the symbol $\hat{}$ indicates Cartesian coordinates, while $\hat{\mathbf{U}}$ is known as the vector of conservative variables where ρ denotes density, u, v, w are the

Cartesian components of the velocity vector, and e denotes the internal energy. The fluxes are written as

$$\hat{\mathbf{F}} = \begin{pmatrix} \hat{F}_1 \\ \hat{F}_2 \\ \hat{F}_3 \\ \hat{F}_4 \\ \hat{F}_5 \end{pmatrix} \;;\; \hat{\mathbf{G}} = \begin{pmatrix} \hat{G}_1 \\ \hat{G}_2 \\ \hat{G}_3 \\ \hat{G}_4 \\ \hat{G}_5 \end{pmatrix} \;;\; \hat{\mathbf{H}} = \begin{pmatrix} \hat{H}_1 \\ \hat{H}_2 \\ \hat{H}_3 \\ \hat{H}_4 \\ \hat{H}_5 \end{pmatrix} \tag{4}$$

Inviscid fluxes are of the form

$$\hat{\mathbf{F}}_I = \begin{pmatrix} \rho u \\ \rho u u + p \\ \rho v u + p \\ \rho w u + p \\ \rho u H \end{pmatrix} \;;\; \hat{\mathbf{G}}_I = \begin{pmatrix} \rho v \\ \rho u v + p \\ \rho v v + p \\ \rho w v + p \\ \rho v H \end{pmatrix} \;;\; \hat{\mathbf{H}}_I = \begin{pmatrix} \rho w \\ \rho u w + p \\ \rho v w + p \\ \rho w w + p \\ \rho w H \end{pmatrix} \tag{5}$$

where enthalphy $H = p + \rho e$ and p denotes pressure.
The Cartesian components of the flux tensor are of the form

$$\hat{\mathbf{F}}_V = \begin{pmatrix} 0 \\ \sigma_{xx} \\ \sigma_{xy} \\ \sigma_{xz} \\ \sigma_{xx} u + \sigma_{xy} v + \sigma_{xz} w + q_x \end{pmatrix} \tag{6}$$

$$\hat{\mathbf{G}}_V = \begin{pmatrix} 0 \\ \sigma_{yx} \\ \sigma_{yy} \\ \sigma_{yz} \\ \sigma_{yy} v + \sigma_{yx} u + \sigma_{yz} w + q_y \end{pmatrix} \tag{7}$$

$$\hat{\mathbf{H}}_V = \begin{pmatrix} 0 \\ \sigma_{zx} \\ \sigma_{zy} \\ \sigma_{zz} \\ \sigma_{zz} w + \sigma_{zx} u + \sigma_{zy} v + q_z \end{pmatrix} . \tag{8}$$

To close the system, the equation of state for a calorically perfect gas is used

$$p = (\gamma - 1)\rho(e - \frac{1}{2}|\mathbf{v}^2|) \tag{9}$$

where $\gamma = c_p/c_v$ is the specific heat ratio.

The stress tensor, σ, is proportional to the strain, i.e. the velocity gradients. The Cartesian components of the stress tensor are

$$\sigma_{ij} = \mu \left(\frac{\partial v_i}{\partial x_j} + \frac{\partial v_j}{\partial x_i} - \frac{2}{3}\delta_{ij}\frac{\partial v_l}{\partial x_l} \right) + (\lambda + \frac{2}{3}\mu)\delta_{ij}\frac{\partial v_l}{\partial x_l} \tag{10}$$

where the summation convention was used and x_i, v_i denote Cartesian coordinates and velocity components. The coefficients μ and λ are called shear viscosity and volume viscosity, respectively. For Newtonian fluids like air, the Cartesian components of the viscous force are of the form

$$\frac{\partial}{\partial x_j}\sigma_{ij} = (\lambda + \mu)\frac{\partial}{\partial x_j}(\nabla \bullet \mathbf{v}) + \mu \Delta v_i. \tag{11}$$

The following relation holds

$$\lambda + \frac{2}{3}\mu \geq 0. \tag{12}$$

Using Stokes' assumption, the equal sign applies in Eq.12.

As can be seen from Eqs. 1-5, the *Navier–Stokes* equations are nonlinear, and give rise to contact discontinuities and (in the limit of low viscosity) shocks, requiring special numerical algorithms.

3 *Parallel Scalability for Large Scale CFD Applications*

In the following we make the assumption that the *Navier–Stokes* equations have been successfully solved on a single processor architecture. The next question then is, can the same problem be solved on a parallel architecture, resulting in a quasi linear speedup, that is, can parallel scalability be achieved. For the time being, we do not consider numerical scalability, since parallel and numerical scalability may be conflicting issues.

Parallel scalability is the ideal condition where, for a given problem, the product of execution time and number of processors is constant. There are basically three different requirements that have to be fulfilled to achieve parallel scalability, namely: (1) the code does not contain a sequential part, (2) communication is at zero cost (no overhead time required), and (3) each processor has the same workload. A discussion of issues in achieving parallel scalability can be found in [1] and [2].

In general, (1) cannot be achieved for flexible and user friendly codes because of program startup time, global communciation operations such as calculation of residuals and stopping criteria, and sequential I/O operations. Obviously, time is spent on sending messages between processors. Even if computation can be overlapped with communication, a certain amount of work for setting up the message transfer remains. Therefore, a certain amount of parallel inefficiency is invariably connected with (1) and (2). In general, assigning equal workload to processors can be achieved by redistributing work during the course of the simulation. However, it has to be realized that load balancing by itself requires computational resources.

It should be noted that with an increasing number of processors the amount of communication increases, which might cause a nonlinear response of the communication channels. Also, the utilization of advanced solving strategies (see Section 4) for complicated physical phenomena in conjunction with highly complex geometries may show a dynamic behavior with regard to computing time per grid point. Therefore, achieving the same workload for each processor during the computation can only be ensured by **dynamic load balancing** (see Section 3.4). In addition, the numerical algorithm might cause a nonlinear increase in communication demand.

3.1 *Parallelization Strategies for CFD Codes*

There are basically three ways of parallelizing a code. First, a simple and straightforward approach is to **parallelize the do loops** in the code. Many so called automatic parallelizers analyze do loops and suggest a parallelization strategy based on this analysis. This concept, however, is not scalable to hundreds or thousands of processors, and results in very limited speedup [3].

Most applications in science and engineering can be described by a set of equations in some kind of solution space. A second approach is therefore to parallelize the numerical solution process for these equations. For example, if a matrix-vector multiplication occurs, this multiplication could be distributed on the various processors and performed in parallel. Again, scalability to a large number of processors cannot be obtained. Moreover, this technique would work only for large regular matrices. If a problem were represented by a large number of smaller matrices (often the case in practice (see Fig. 17)), parallelization would be impossible.

The third approach is denoted as domain decomposition, sometimes also referred to as grid partitioning. The idea is simple. The solution domain is subdivided into a set of subdomains (blocks) that exchange information to update each other during the solution process. The numerical solution takes place within each domain, and thus is independent of the other domains. The solution space can be the actual space-time continuum, or it can be some abstract space. For the simulation process, this space is discretized and thus is described by a set of points. Domain decomposition is the most general and versatile approach. It also leads to the best parallel efficiency, since the number of points per subdomain can be freely varied as well as the number of subdomains per processor. A large number of codes in science and engineering use finite elements, finite differences, or finite volumes on either unstructured or structured grids. Very often, the governing physical equations are converted into a set of linear equations. The process of parallelizing this kind of problem is to decompose the physical solution domain. Software is available to efficiently perform this process both for unstructured and structured grids [4]. Applying this strategy results in a fully portable code, and allows the user to switch over to new parallel hardware as soon as it becomes available.

3.2 *Domain Decomposition*

As pointed out in section 3.1, domain decomposition is the most general and versatile approach for parallelizing a CFD code. In the following the importance of domain decomposition for achieving parallel scalability is discussed in more detail.

Today, hardware vendors are not able to produce a shared memory system with a large number of processors (more than 128) that has good parallel scalability if the shared memory programming model is used. Therefore, as pointed out in 3.1, simple loop transformations and parallelization of the numerical solution process are not appropriate for today's grand challenge problems.

In contrast, the domain decomposition approach exhibits a high degree of parallelism. The general strategy in the solution procedure of multiblock flow solvers is to construct a halo of cells which surround each block and that contain information from corresponding cells of neighboring blocks. This halo of cells, updated at proper times during the numerical procedure, allows the flow solution inside each subdomain to proceed independently. In updating halo cells parallelism can be achieved because updates are performed locally between pairs of processors. No global operations are necessary.

There is, however, an important aspect of this parallelization approach, namely the geometrical complexity of the solution domain (see Fig. 2). In the following, a brief discussion on geometrical complexity is given and how it affects parallelization. If the solution domain comprises a large rectangle or box, domain decomposition is relatively straightforward. For instance, the rectangle can be decomposed into a set of parallel stripes, and a box can be partitioned into a set of planes. This leads to a one-dimensional communication scheme where messages are sent to left and right neighbors only. However, more realistic simulations in science and engineering require a completely different behavior. For example, the calculation past an entire aircraft (see Fig. 1) or spacecraft (see Fig. 2) leads to a partitioning of

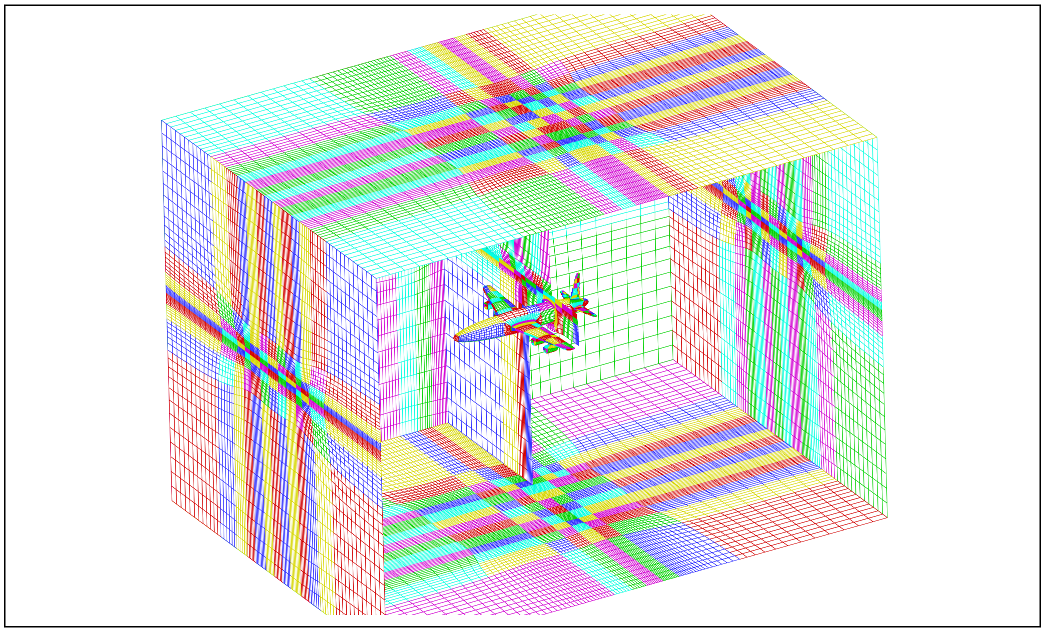


Figure 1: Generic aircraft configuration generated by GridPro [5].

the solution domain that results in a large number of subdomains of widely

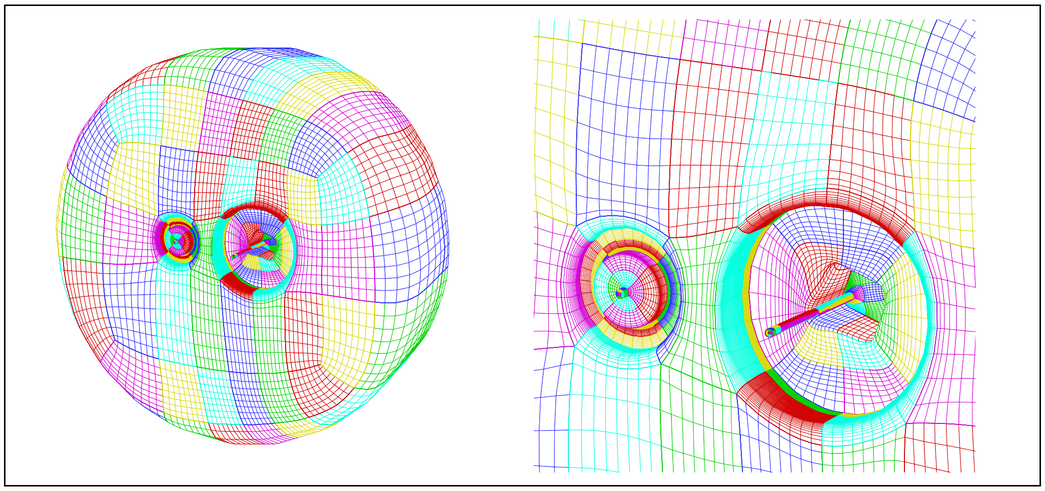


Figure 2: The surface grid for the ESA/NASA Huygens space probe. The grid comprises 561,654 gridpoints in 462 blocks. The large number of blocks is needed because of the high degree of geometrical complexity, modeling the probe's instruments to measure the composition of Titan's atmosphere in 2004 [6].

different size, because of a certain grid topology, optimized for particular flow phenomena, i.e. the number of grid points in the various blocks, may be different. As a consequence, it is unrealistic to assume that a solution domain can be partitioned into a number of equally sized subdomains [7]. On the contrary, the set of subdomains is unordered (unstructured) on the subdomain level, leading to random communication among subdomains. In other words, the communication distance cannot be limited to nearest neighbors, but any distance on the processor topology is possible (processor topology describes how the processors are connected, for instance as a 2D mesh, as a torus or as a hypercube etc.). Hence, the efficiency of the parallel algorithm must not depend on nearest neighbor communication. Therefore, the parallelization of solution domains for complex geometries requires a more complex communication pattern to ensure a loadbalanced application. It also requires more sophisticated message passing among neighboring blocks, which may reside on the same, on a neighboring, or on a distant processor (see Fig. 3). The basic parallelization concept for this kind of problem is the introduction of a new type of boundary condition, namely the inter-domain boundary condition that is updated in the solution process by neighboring subdomains via message passing. Parallelization then is simply achieved by the introduction of a **new type of boundary condition**. Thus, paralleliza-tion of a large class of complex problems has been logically reduced to the well known problem of specifying boundary conditions.

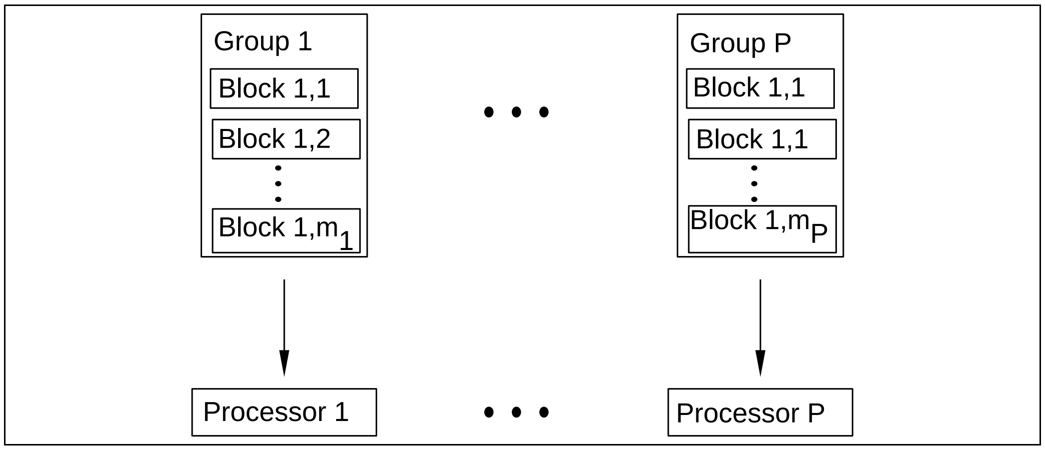


Figure 3: Mapping of blocks to processors for general multiblock topologies. Blocks are assembled into a set of groups, where each group contains the same number of grid points. Each group then is assigned to a particular processor. Communication then has to discern between communication of blocks residing on the same processor and between blocks that reside on a different processor.

3.3 *Message Passing Methods*

The *ParNSS* code [6], written in ANSI-C, used to solve the *Navier–Stokes* equations (Eq.1), uses message passing for updating halo cells. Messages are exchanged after every iteration by using the industry standard MPI (Message Passing Interface) library or the PVM (Parallel Virtual Machine) library. For viscous flow an overlap of two cells is used. However, no messages are sent across diagonals of a block. Instead, only faces are updated and finite difference formulas are used to approximate diagonal terms, i.e. mixed derivatives.

The update of halo cells can be done with either of two message passing strategies (see Fig. 4):

- **BSNR** (Blocking Send, Non-blocking Receive). First, the code sets up non-blocking receives for all incoming messages. Then, looping over all blocks for each of the six faces of a block, a message is sent to each face of a neighboring block, except wall faces.

- **NSBR** (Non-blocking Send, Blocking Receive). Here a single loop over all blocks is used and messages are sent as soon as information is ready. After that, for each face of a block, again wall faces are excepted, a blocking receive is posted. There can be no deadlock, since the mapping between faces of neighboring blocks is one-to-one.

```
BSNR

#initialization loop
loop over all blocks{
    exchange halo cells
}
loop over iterations{
    loop over blocks{
        compute
    }
    loop over blocks{
        exchange halo cells
    }
}
```

```
NSBR

#initialization loop
loop over all blocks{
    send halo cells
}
loop over iterations{
    loop over blocks{
        receive halo cells (blocking)
        compute
        send halo cells (non-blocking)
    }
}
loop over blocks{
    receive halo cells
}
```

Figure 4: Two different message passing strategies for updating halo cells are shown.

For the *BSNR* method, the communication is separated from the computation; all the receives are posted, then all the messages are sent. On the other hand, for the *NSBR* method, each message is sent as soon as it is ready, spreading out the communication traffic in time. Thus we could expect that the impact of a slow communication fabric is felt to the greatest extent for the *BSNR* method.

Even though the *NSBR* method has the advantage of spreading out the message traffic, it has a potential overhead in that messages are sent as soon as they are available, meaning that they may be received before the receiving process is ready to use them. Thus the message passing library implementation must manage buffers to hold the messages until the receiving process is ready.

These two different message passing strategies have been tested on two different supercomputers at the Center of Advanced Computing Research, California Institute of Technology, namely, the 256 processor HP Exemplar and a 'Beowulf' system at Caltech.

The HP Exemplar system has (September 1997), 256 HP PA8000 processors running at 180 MHz. Each processor has 256 MBytes memory and 4 Gbyte disk. The system is packaged as sixteen 'hypernodes', where each of these nodes consists of 16 processors in a box with 4 Gbytes shared memory and 64 Gbytes disk. Processors within a hypernode communicate with each other at high bandwidth and with low latency via a crossbar switch. Processors on hypernodes communicate via CTI (Coherent Toroidal Interface) in a ring, at a somewhat lower bandwidth, with somewhat higher latency. This model of communication between processors is called NUMA (Non-Uniform Memory Access).

Each processor of the Beowulf machine has (September 1997), 58 Intel Pentium Pro compute processors, running at 200 MHz. Each processor has 128 MBytes memory and 3.1 Gbytes disk. The Beowulf is packaged as a conventional PC box, with motherboard, power supply, disk, floppy drive, ports and so on. There is a host processor that is connected to the Internet through which users can log in, compile, and launch jobs. In addition, each PC is supplied with fast (100 Mbit/sec) ethernet for interprocessor communication, which is routed by four crossbar switches.

The graphs in Fig. 5 show the impact of the message passing strategies *BSNR* and *NSBR* on the parallel efficiency of the computation. As a testcase, the inviscid flow past the Single-Stage to Orbit (X-33) configuration at Mach 9.8 was chosen. A grid comprising 274 blocks and a total of 342,361 grid points was used (Section 5.4).

3.4 *Strategy for Dynamic Load Balancing*

For some applications the workload per processor can be estimated or even analytically determined in a preprocessing step before the simulation is started. If the workload does not change during the course of the computation static load balancing is sufficient. However, the majority of applications in computational science and engineering show a more complex runtime behavior, necessitating some kind of dynamic load balancing to provide the same workload on each processor at all times.

The goal of dynamic load balancing can be stated as follows:
Given a collection of tasks performing a computation and a set of processors on which these tasks are to be executed, find the mapping of tasks to processors that minimizes the run time.

As will be stated in Section 4 the numerical solution strategy is based

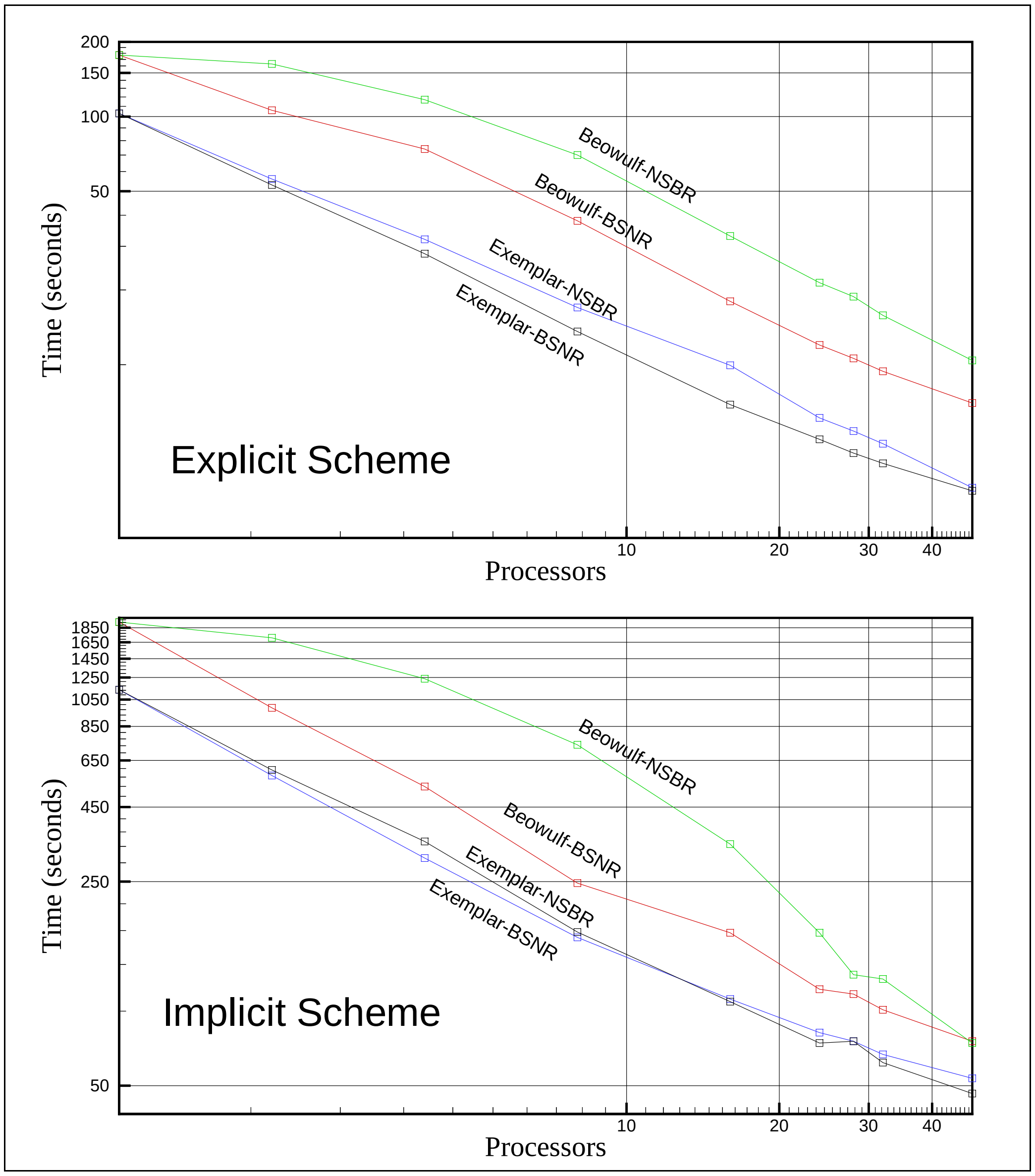


Figure 5: Run times for 10 iterations of the ParNSS code, (top) Explicit Scheme and (bottom) Implicit Scheme, solving the Euler equations at Mach 9.8 for the X-33 model. Each panel shows computing times for both Beowulf and Exemplar for the two message passing schemes. In both cases, the Exemplar is roughly two times faster than the Beowulf machine. We note that the timestep for the implicit scheme may be much larger than for the explicit, also that the implicit step may decrease the residual by much more.

on the idea of a varying coupling strength among grid points during the course of the solution in order to achieve numerical scalability. However, this approach results in an algorithm that is not parallel scalable, because of the load imbalance that is caused by the dynamic behavior of the implicit *GMRES* algorithm. Therefore, in order to achieve both numerical and parallel scalability, it is necessary to implement dynamic load balancing in the Navier-Stokes flow solver.

```
typedef struct {
    char BlockName[32];
    int Block;
    int I,J,K; /* number of points in each direction */
    int NodeId;/* processor ID on which block resides*/
    int NumOfBc;
    int intbc,*ibc;
    bcond *bc;
    coord ***C;/* pointer to grid points */
    Qvector ***S, ***D;/* Flux vector Q and Delta_Q */
    double timestep; /* local timestep */

    double comptime;/*number of seconds needed for last
                    iteration */
    int active; /* block is active(1) or non-active(0)*/
    int moveable;/*block is movable to other
                  processor(1) or non-movable*/
} block;
```

Figure 6: The topology of the solution domain is described by the data structures `block` and `face`.

Dynamic load balancing in the *ParNSS* code has been achieved through the following five stages [8].

1. **Load Evaluation:** Estimates of the processor's workload must be provided to determine the current load imbalance. In the *ParNSS* code the runtime behavior of each block is monitored during an iteration.

2. **Profitability:** Once runtimes for each block have been collected via global communication (see Fig. 8), the load imbalance is computed. If the cost for a load balancing step is lower than the current imbalance, load balancing actions are performed.

```
typedef struct {
    int I,Istart,J,Jstart;
    int IU,IstartU,JU,JstartU;
    int MyFace,MyPart,MyBlock,MyType,MyInd,
    nnode,nblock,nface,npart,op,nind;
    char NBlockName[32];
    char XYZchar;
    int scut1,scut2,sgcut,rcut1,rcut2,rgcut;
    int stype,rtype;
    double *inbuf; /* message buffer */
    double *outbuf; /* message buffer */
#ifdef MPI2
    MPI_Request send_request;
    MPI_Request receive_request;
    MPI_Status receive_status;
#endif
} face;
```

Figure 7: The topology of the solution domain is described by the data structures `block` and `face`.

3. **Selection of Moveable Tasks:** Flags are attached to each block that constrain the task selection process. First, blocks are marked as moveable or non-moveable. Second, blocks are automatically deactivated, if the local residual of this block compared to the global residual is below a certain limit. However, a deactivated block is still updated via message passing, and if a solution change above a specified threshold propagates into a deactivated block, this block is re-activated. Only those blocks which are both moveable and active are *distributable*.

4. **Task Mapping Decision:** We now compute which processor should be responsible for each distributable block; the details are in section 3.4.1.

5. **Task Migration:** In a loosely synchronous communication, the distributable blocks are moved to their new homes, and the face structures updated so that each block knows the correct processor for its neighbors; the details are in section 3.4.2.

By subdividing the load balancing process into single stages, a high degree of flexibility is obtained for analyzing different strategies in each stage.

3.4.1 *Task Mapping Decision*

Currently we use a simple bin-packing algorithm, designed to produce a reasonable load-balance only, but neglecting the communication cost. This is because much of the numerical operations take place in a block-implicit scheme, where the ratio of communication to computation cost is very small [7].

The computation of the mapping takes place serially, since it is easier that way, and it takes such a small time. The workload resulting from the non-distributable blocks is computed for each processor, providing a workload number for each processor. The distributable blocks are sorted in decreasing order of their workload to form a list. The largest block is removed from this list and assigned to the processor with the smallest workload, thereby increasing the workload. This is repeated until there are no more blocks in the list.

When the new mapping of blocks to processors is complete, the parallel efficiency of the new map is estimated; only if this is sufficiently better than the previous efficiency does the expensive task migration actually take place.

3.4.2 *Task Migration*

This section describes how blocks are sent to their new processor, updating the communication topology to ensure the correct block boundary exchange for the next iteration.

Each block to be moved is serialized, including grid point coordinates, numerical solution, message buffers, and flags, and all its memory is freed on the source processor. The set of messages are sent to the respective destination processors, and finally we need blocks that share a face to know each others processor ID; this is stored in the data structure `face`.

This is done using the idea of the Voxel Database [9], which is, in essence, a way to associate data with geometric points. A Voxel Database allows a distributed application to share data, using a key based on geometric position. If one processor writes data 'at coordinates (x, y, z)', then another processor can inquire if there is any data at that position. Processors can also write data into these shared, geometically-positioned memories in a weakly-coherent fashion.

In this case, we use a Voxel Database to store lists of processor ID's, which are positioned at the centers of the faces of the blocks. We first loop

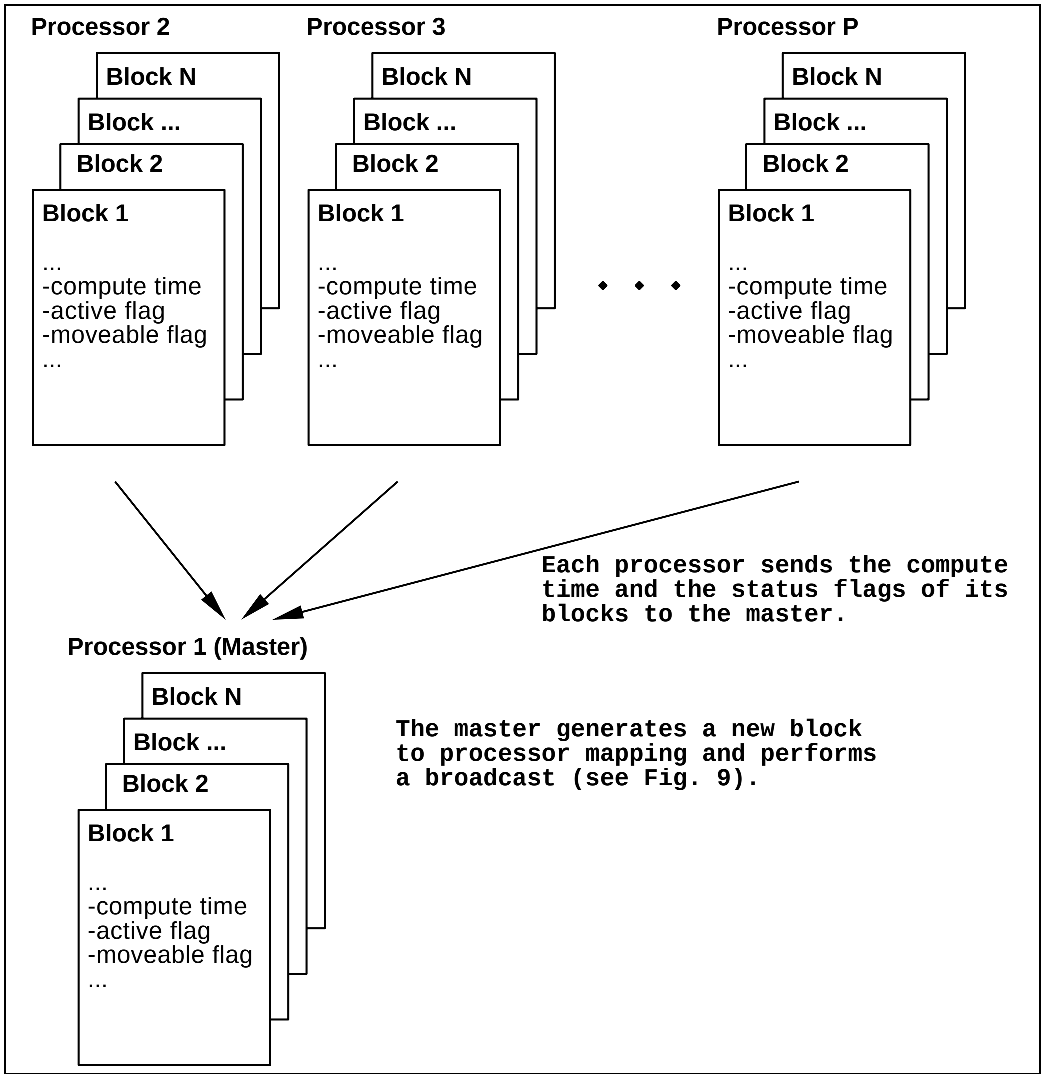

Processor 2
Processor 3
Processor P
Block N
Block ...
Block 2
Block 1
...
-compute time
-active flag
-moveable flag
...
Each processor sends the compute time and the status flags of its blocks to the master.
Processor 1 (Master)
The master generates a new block to processor mapping and performs a broadcast (see Fig. 9).

Figure 8: The topology of the multiblock grid is described by the set of objects of type `block` (see Fig. 6). In addition, for each block there is a set of up to 6 faces described by object type `face` (see Fig.7). Each processor holds the entire grid topology information.

```
All processors: Send runtime and status flags of own blocks
                to master.

Master:         Determine load imbalance.

Master:         Compute new mapping of blocks to processors.

Master:         Broadcast new mapping to all processors.

All processors: Loop over all blocks of SD n=1,...,N
                 if (n is mapped to a new processor)
                  action on the source processor:
                    -send block data to destination processor
                    -free memory used by this block
                    -set new processor id
                  action on the destination processor:
                    -allocate memory for arriving block
                    -receive block
                    -set new processor id

All processors: For each face of each block determine processor id
                of neighboring block using global combine function.
```

Figure 9: The sequence of commands illustrates how dynamic load balancing is implemented in the *ParNSS* code.

over all such faces, putting the processor ID in the list associated with the position of the center, then synchronize the database. When we can look at the list associated with a given face, its length is one for a boundary face, and two for a face that is shared with another block, i.e., and interior face. In the latter case, one of the processor ID's in the list is the processor ID of the block that shares the face, which is what we wanted to find. A more complete description of this process is given in [17].

The Voxel Database can be used in a more general way to dispense with connectivity files for multiblock grids. Given only the geometric coordinates of the vertices of the blocks, we can synthesize the connectivity of faces, edges, and vertices, similarly to the above.

4 *Numerical Scalability for Large Scale CFD Applications*

In this section we describe how numerical scalability can be achieved. It is believed that the successful solution of the large scale parallel N-S equations can only be performed by combining **grid generation**, **domain decomposition**, and **numerical solution scheme**. Each of the three elements has its own unique contribution in the numerical solution process. However, in the past, these topics were considered mainly separately and their close interrelationship has not been fully recognized.

In this chapter **domain decomposition** along with the **numerical solution scheme** will be discussed. Grid generation will not be discussed here (see instead [10], [11]); but it should be noted that grid generation has a major influence on the overall accuracy of the solution and the convergence properties of numerical methods.

In the following, strategies are considered for efficiently solving the system of linear equations that arises from the discretized *Navier–Stokes* Equations.

Ideally, numerical scalability would mean that computing time required to solve a problem is linearly related to problem size. Provided parallel scalability is achieved, the solution time for a problem would remain constant if both problem size and the number of processors increase at a fixed ratio. For example, if a system of linear equations of size N is to be solved and a direct method like LU-decomposition is used, the number of floating point operations is $O(N^3)$, which is far from being numerically scalable. There are also many iterative solution procedures for which the convergence speed is of order $O(h^2)$, h being a measure of the so called grid spacing, simply denoting the distance between two neighboring grid points. The convergence speed is a measure of how fast the residual in the solution is reduced. For small h, many of the iterative solution procedures stall [12]. For an equidistant mesh in 3D, $h = N^{-1/3}$. The number of floating point operations to reach a certain convergence level is then of order $O(N^{5/3})$ for $O(N)$ denoting the number of operations per iteration. Therefore, in order to obtain a numerically scalable algorithm additional measures have to be taken.

Numerical experience has shown that a single numerical scheme has varying numerical efficiency during the course of the solution process. The novel feature presented in this article is to apply a sequence of numerical strategies, and to establish criteria for switching over to a different numerical scheme as soon as the present scheme becomes numerically inefficient.

The Tangled Web approach is a combination of the following techniques that will be briefly discussed below:

- **Grid sequencing:** this was mentioned in Section 4.1, and consists of using a set of nested grids, analogous to multigrid methods. There is a sequence of grids, each with 8 times as many points as the last, and we loop through these from coarsest to finest, interpolating the final solution on one grid as the initial solution on the next finer grid. At the same time coarsening is used to compress the Eigenvalue spectrum (*GMRES* technique).

- **Domain decomposition:** This several benefits. First, it allows the construction of completely flexible topologies which, in turn, allow for grid point clustering and grid optimization. Second, the inversion of a set of small matrices, arising from the implicit solution of each block, is faster than the inversion of a single large matrix. Third, for each block the implicit solution is obtained by the so called Dynamic *GMRES* technique that might exhibit a different numerical behavior because the Krylov basis may be of different size. This may lead to load imbalance and limit parallel efficiency, requiring dynamic load balancing (Section 3.4). Fourth, the sparse linear system to be solved with *GMRES* requires an efficient and effective preconditioner. Domain decomposition is one technique to reduce the condition number [13].

- **Adaptive coupling strength:** The dynamic coupling strength approach is shown in Figs. 10 and 11 and accounts for the fact that coupling of grid points should increase during the computation.

- **Dynamic load balancing:** As noted above, there are several reasons for load imbalance, including the unequal number of iterations of the GMRES linear solver in the block-implicit phase of the solution. There are other reasons for load imbalance, the most important being the flow physics and the local grid density. It may also be necessary to utilize a different set of physical equations within different blocks, depending upon the prevailing flow features.

- **Block activation/deactivation:** Subdomain or block activation or deactivation is where the flow is converged in some parts of the computational domain, but not in others, so it is natural to concentrate resources only where necessary. The activation and deactivation is steered by both the change in the numerical solution within the block and by the amount of change received by message passing.

The important feature is that these acceleration techniques are applied in combination for the sake of synergistic effects.

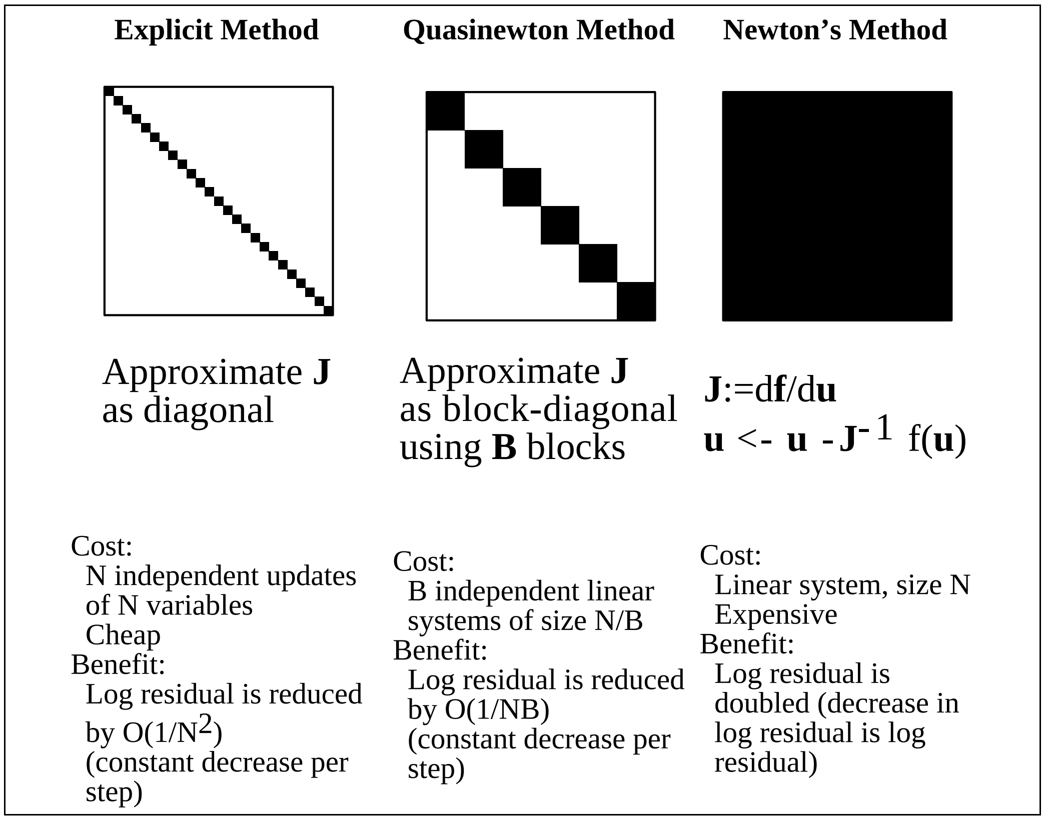

Explicit Method
Quasinewton Method
Newton's Method
Approximate J as diagonal
Approximate J as block-diagonal using B blocks
J:=df/du
u <- u - J^-1 f(u)
Cost: N independent updates of N variables Cheap
Benefit: Log residual is reduced by O(1/N^2) (constant decrease per step)
Cost: B independent linear systems of size N/B
Benefit: Log residual is reduced by O(1/NB) (constant decrease per step)
Cost: Linear system, size N Expensive
Benefit: Log residual is doubled (decrease in log residual is log residual)

Figure 10: The three plots depict the coupling strength (black area) for solving a linear system of equations, described by matrix **J** . Coupling strength is dynamically increased during the course of the computation. N denotes the size of the system, B is the number of blocks.

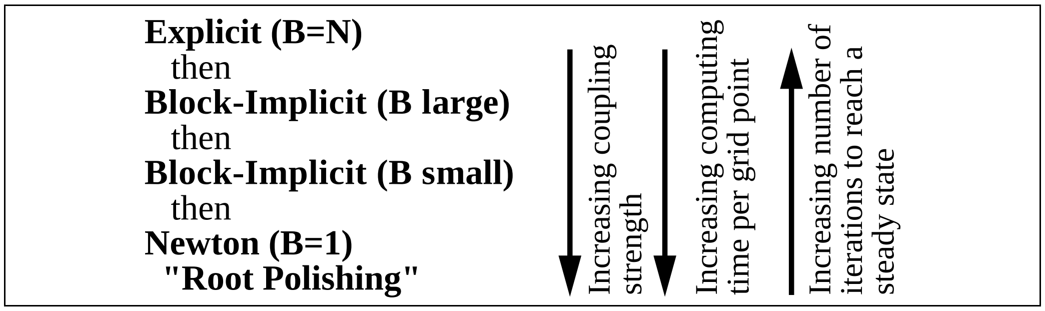

Explicit (B=N)
then
Block-Implicit (B large)
then
Block-Implicit (B small)
then
Newton (B=1)
"Root Polishing"
Increasing coupling strength
Increasing computing time per grid point
Increasing number of iterations to reach a steady state

Figure 11: The relations between coupling strength, computing time per grid point, and iterations needed to reach a steady state is presented. B is the number of blocks and N denotes the problem size, e.g. the number of grid points.

4.1 ***The*** *Navier–Stokes* ***Equations as a Set of ODEs***

We distinguish space discretization from time discretization. The space discretization produces a set of Ordinary Differential Equations:

$$\frac{d\mathbf{U}}{dt} = f(\mathbf{U}), \tag{13}$$

and we assume the existence of a steady state $\mathbf{U}^*$ such that $f(\mathbf{U}^*) = 0$. Discretizing time, we approach $\mathbf{U}^*$ by a sequence of explicit or implicit steps, repeatedly transforming an initial state $\mathbf{U}^0$ into a final state $\mathbf{U}^*$.

The **explicit step** is, for example, the two-stage Runge-Kutta

$$\mathbf{U}^{n+1} = \mathbf{U}^n + f(\mathbf{U}^n + f'(\mathbf{U}^n)\Delta t/2)\Delta t. \tag{14}$$

As the **implicit time step** a backward Euler is used

$$\mathbf{U}^{n+1} = \mathbf{U}^n + f(\mathbf{U}^{n+1})\Delta t. \tag{15}$$

The final step, i.e. attaining the steady state directly via Newton [14] [15], can be thought of as an implicit step with infinite Δt: Solve $f(\mathbf{U}) = 0$. There is also a weaker version of the implicit step, which we might call the linearized implicit step, that is actually just the first Newton iteration of the fully nonlinear implicit step:

$$\mathbf{U}^{n+1} = \mathbf{U}^n + [1 - df/d\mathbf{U}\ \Delta t]^{-1} f(\mathbf{U}^n)\Delta t. \tag{16}$$

The resulting linear system is solved by the *GMRES* method [16].

4.2 ***Domain Decomposition for Eigenvalue Spectrum Compression and Preconditioning***

A major question arises in how the decomposition process affects the convergence rate of the implicit scheme. First, it should be noted that the N-S equations are not elliptic, unless the time derivative is omitted and inertia terms are neglected (Stokes equations). This only occurs in the boundary layer when a steady state has been reached or has almost been reached. However, in this case the Newton method will converge quadratically, since the initial solution is close to the final solution. The update process via overlap boundaries should therefore be sufficient. In all other cases, the N-S equations are dominated by hyperbolic phenomena. Hence, a full coupling of all points in the solution domain would be nonphysical, because of the finite propagation speed, and is therefore neither desirable nor necessary.

Continuing the discussion of convergence speed, it should be remembered that for steady state computations implicit techniques converge in fewer steps than fully explicit schemes, but each step takes more computing. The former are generally more computationally efficient, in particular for meshes with large variations in grid spacing. However, since a full coupling is not required by the physics, decomposing the solution domain should result in a convergence speed up. This is due to the inversion of a set of small matrices being faster than the inversion of the single large matrix, although boundary values are dynamically updated. In the preconditioning process used for the Conjugate-Gradient technique, domain decomposition is used to decrease the condition number (ratio of largest to smallest Eigenvalues) of the matrix forming the left hand side, derived from the discretized N-S equations. In other words, the Eigenvalue spectrum is compressed because the resulting matrices are smaller. It is shown in [16] that this ratio is a measure of the convergence speed for generalized conjugate residual algorithms. Having smaller matrices, the condition number should not increase; based on physical reasoning it is to be expected that, in general, the condition number should decrease. On the other hand, if the decomposition leads to a blocksize of 1 point per block, the scheme is fully explicit and hence computationally less efficient than the fully implicit scheme. Therefore, an optimal decomposition topology must be selected and most likely depends on the flow physics and the type of implicit solution process. However, a number of numerical experiments has been performed with the *ParNSS* code, clearly demonstrating the convergence speed up. Block numbers have been varied from 2 to 1024 in 2D (see Tab. 2) and from 6 to 384 in 3D (see Tab. 5), using an otherwise identical grid.

5 *Results Using the Tangled Web Approach*

In this section we present results attained by using the *Tangled Web* Approach on four different examples:

- A Mach 1.7 Euler flow over the NACA0012 airfoil,
- A Navier-Stokes computation for the NACA0012 airfoil. The viscous laminar flow was computed for an angle of attack of 7 degrees at Mach

1.7 for a Reynolds number of 5×10^6,

- Flow around the ESA/NASA Huygens space probe which is scheduled to enter Titan's atmosphere in 2004,
- A modified X-33 vehicle, serving as a prototype for the new generation SSTO spaceplane. The Euler flow at Ma 9.8 was computed at an angle of attack of 40°.

For all testcases the parallel numerical strategy of section 4 was applied. Computations were performed on a 10 processor Silicon Graphics Power Challenge XL at Center for Logistics and Expert Systems in Germany, as well as on the HP Exemplar and a Beowulf machine at the Center of Advanced Computing Research, California Institute of Technology.

5.1 *NACA Airfoil Inviscid Flow*

The Euler simulation was carried out on a 2 block grid with a total of 48,000 grid points. The effects of both domain decomposition and grid sequencing were investigated and are reported in the following two sections. The original grid was split into 2, 32, 120, 128, 256, 480, and 1024 blocks. In addition, three coarse grid-levels were extracted from this 2 block grid.

5.1.1 *Acceleration by Grid Sequencing*

In Table 1 we show the speedup obtained by grid sequencing using the two block grid. The last row of Table 1 has the scaled computing time of 1. Adding up the computing time on all grid levels results in a speedup value of 1.68.

5.1.2 *Acceleration by Domain Decomposition*

A similar analysis was done for the NACA0012 airfoil investigating the effect of domain decomposition (see Table 2). The same grid with 48,000 points was used. The Euler solution was computed by the implicit *GMRES* algorithm for a Mach number of 1.7. The computation ended after the residual dropped by 10 orders of magnitude. The optimal speedup was obtained for 480 blocks (see Sec. 4.2).

Grid level	**Grid points**	**Scaled computing time**
3	832	0.0026
2	3162	0.0150
1	12,322	0.1200
0	48,000	0.4600
0 (no grid sequencing)	48,000	1.0000

Table 1: Results for solution acceleration by using multilevel grids for a 48,000 point NACA0012 airfoil. The Euler solution was computed by implicit *GMRES* for a Mach number of 1.7. The algorithm switched to the next finer grid after the residual dropped to 10^{-12} on the coarse grid level. A speedup of 1.68 is obtained by this technique.

Number of blocks	**Number of points per block**	**Number of iterations**	**Speedup**
2	24000	253	1.00
32	1560	305	1.55
120	435	317	2.33
256	213	333	2.73
480	119	349	2.96
1024	61	380	2.92

Table 2: Results for solution acceleration by variation of block number for a 48,000 point NACA0012 airfoil. The Euler solution is computed by the implicit *GMRES* algorithm for a Mach number of 1.7. The computation stops after the residual drops to 10^{-12}. The optimal speedup is obtained for 480 blocks, resulting in a speedup of 2.96 as compared to the 2 block grid.

5.1.3 ***Acceleration by Adaptive Coupling***

A combination of the explicit Runge-Kutta scheme and the implicit *GMRES* scheme, termed adaptive coupling, was used for the Euler computation of the NACA0012 airfoil (see Fig. 12). A set of three grids, comprising 8, 32, and 128 blocks was used.

In stage 1, (see Fig. 12), four different computations are performed. An explicit scheme is used on an 8 block grid, and the implicit *GMRES* scheme is used on 8, 32, and 128 blocks. In each computation the residual is reduced by two orders of magnitude to 0.1. As can be seen from Fig. 12, the explicit scheme is fastest. In stage 2, we try to reduce the residual from 0.1 to 10^{-3}. Now the situation has changed completely, because the explicit scheme stalled. The fastest solution is delivered by *GMRES* on the 128 block grid. In stage 3, the residual is to be further decreased to 10^{-5}. Again, the same phenomenon as in stage 2 is observed, namely, increasing the coupling strength results in solution speedup. In other words, the *GMRES* solution on 8 blocks is fastest. Obviously, adaptively increasing the coupling strength at each stage leads to a decrease in overall computing time.

5.2 ***NACA Airfoil Viscous Flow***

The Navier-Stokes computation was performed on a sequence of grids and a varying number of blocks. The original grid was a 26 block (27,120 cells) grid that was further split into a 40, 68, and 272 blocks. These grids had exactly the same number of grid cells and the same grid point coordinates. Furthermore, two coarse grid-levels were extracted from the 26 block grid comprising 6,780 and 1,712 grid cells, repectively. The left picture in Fig. 13 shows the 26 block fine grid. The multiblock topology has been chosen to automatically cluster grid points at the airfoil surface. The right plot in Figure 13 depicts the Mach number contour plot of the steady state solution.

5.2.1 ***Acceleration by Grid Sequencing***

The computation was started on the coarsest grid. The residual was reduced by about 6 orders of magnitude by applying first the explicit scheme, followed by the implicit *GMRES* scheme. The solution on this grid was then used as initial solution to the next finer grid. The same strategy was used on each grid level, until the steady state solution on the fine grid was obtained. The grid sequencing approach delivered a speedup of 1.25 in comparison to a computation using the fine grid only. Table 3 summarizes the results of the simulation.

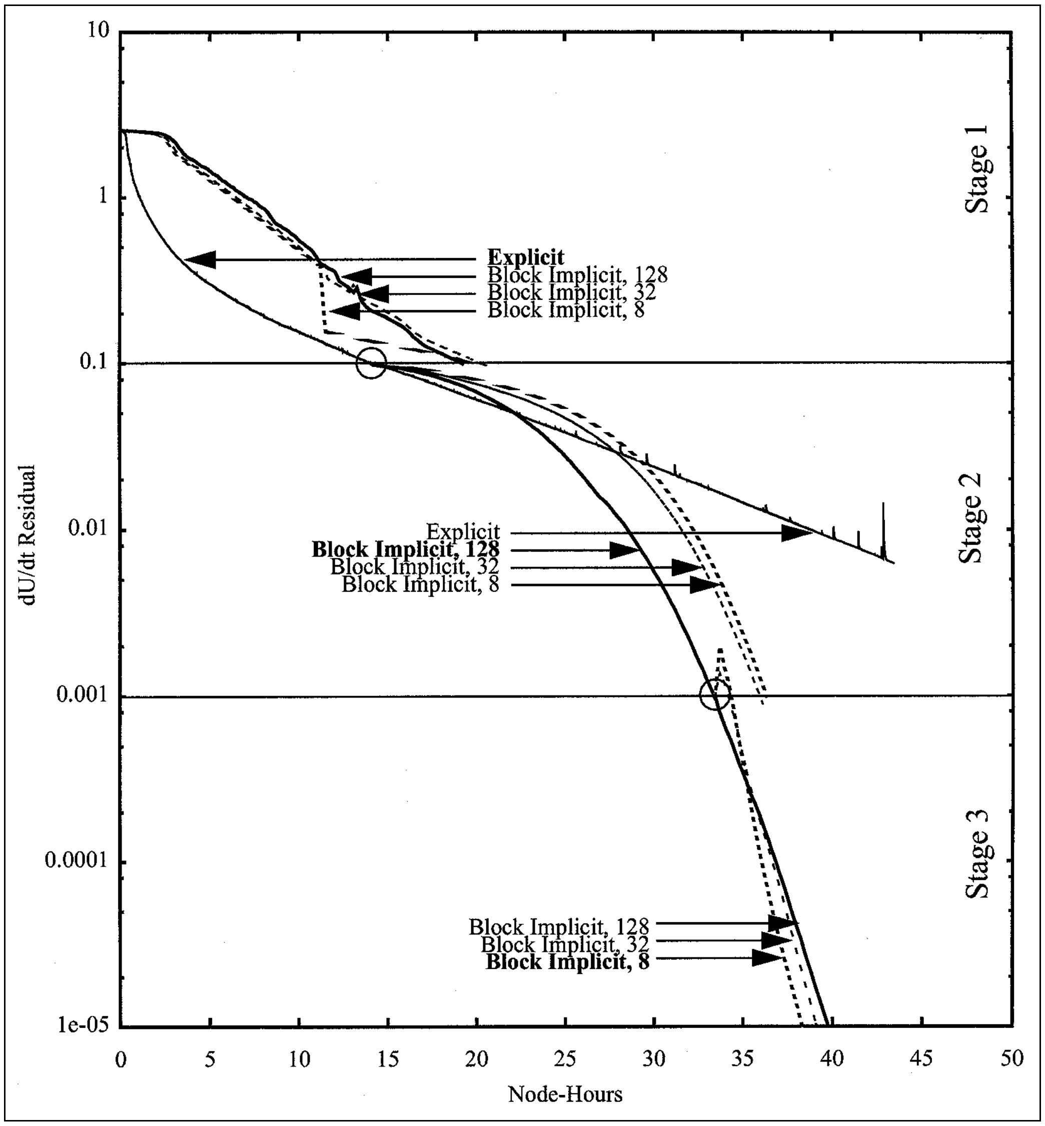


Figure 12: Solution acceleration using adaptive coupling. The computation was done for a NACA0012 airfoil.

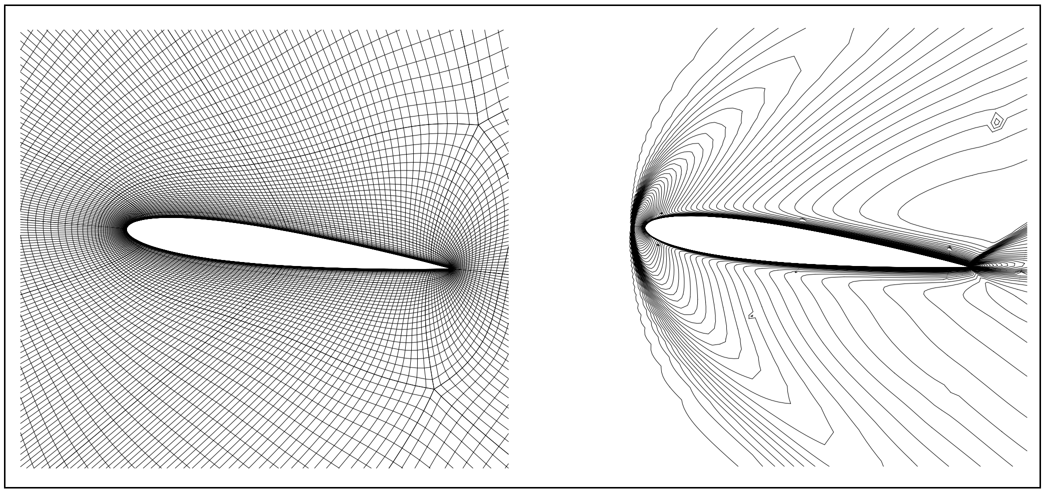


Figure 13: NACA0012 airfoil. Navier-Stokes grid and Mach number contour plot. The fine grid consists of 26 blocks and some 27,000 cells.

Grid level	**Grid points**	**Computing time in s**
2	2156	55
1	7638	335
0	28810	2222
0 (no grid sequencing)	28810	3269

Table 3: Results for solution acceleration using multilevel grids for a 27,000 cell NACA0012 airfoil.

5.2.2 ***Acceleration by Domain Decomposition***

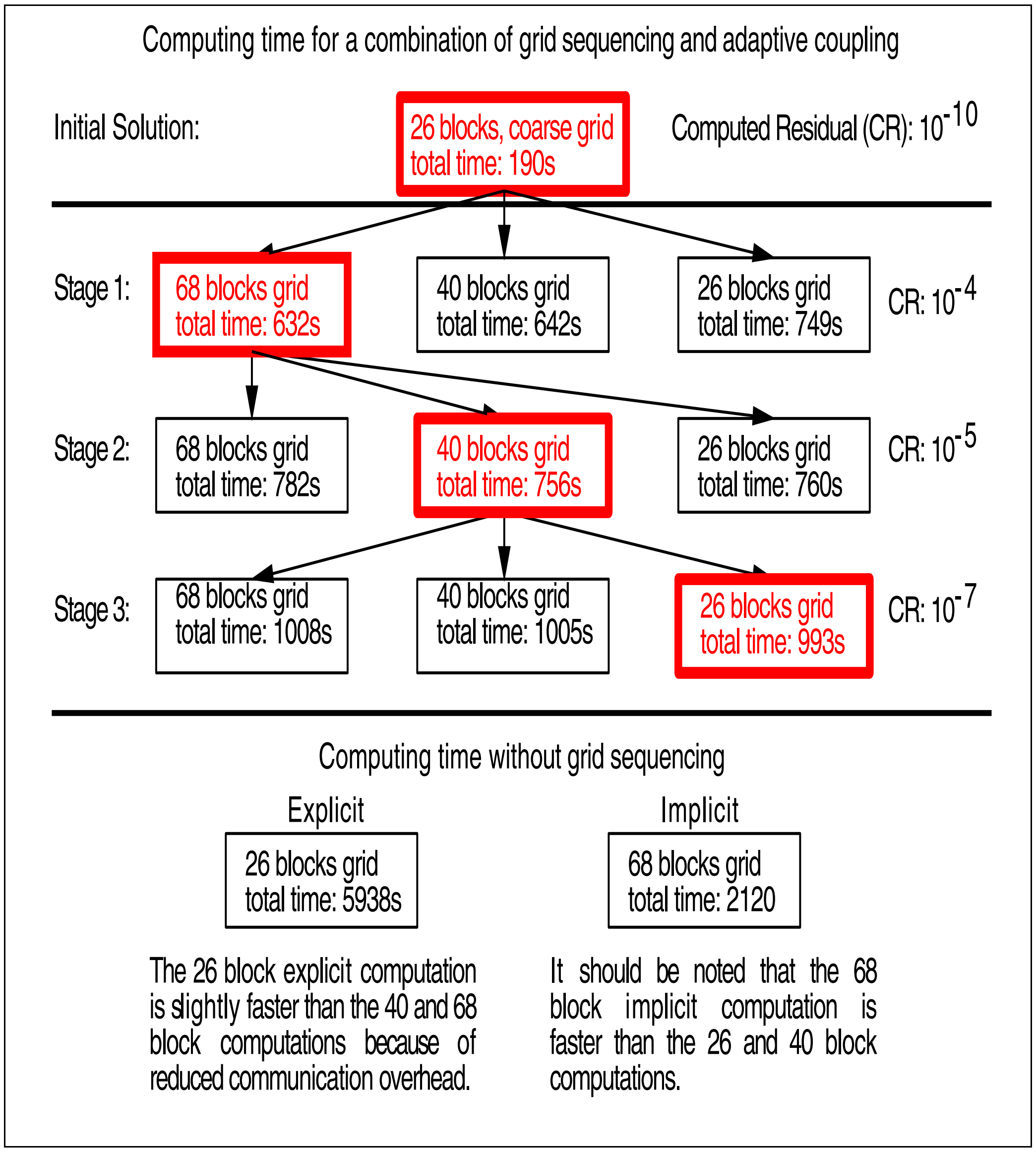
The influence of domain decomposition as a preconditioner for the implicit *GMRES* solver is illustrated by the results in Table 4. Here, computations on 26, 40, 68, and 272 block grids were done. All runs were based on the same parallel numerical strategy, namely, using first the explicit scheme followed by the implicit *GMRES* scheme. Again, the residual is reduced by about 6 orders of magnitude. The decoupling of grid points by splitting the solution domain into smaller blocks reduced the time for each *GMRES* iteration, but required a larger number of iterations. For this configuration the optimal speedup was obtained for 68 blocks.

Number of blocks	**Number of iterations**	**Implicit time**	**Speedup**
26	148	2623	1.00
40	153	2202	1.19
68	161	2066	1.27
272	243	2201	1.19

Table 4: Results for solution acceleration by variation of block number for a 27,000 cell NACA0012 airfoil. The Navier-Stokes solution is computed by using both the explicit and the implicit *GMRES* scheme.

5.2.3 *Acceleration by Adaptive Coupling*

Grid sequencing, in combination with adaptive coupling, was used in the Navier-Stokes computation of the NACA0012 airfoil (see Fig. 14). First, a converged solution was computed on the 26 block coarse grid which served as the initial condition for the computations in stages 1 to 3. It turns out that the combination of grid sequencing and adaptive coupling results in substantial speedups in comparison with fully explicit or block implicit solutions on the fine grid only. The speedups are 5.91 and 2.11, respectively. It should be mentioned that a further reduction in the total execution time will be possible using dynamic load balancing.


Computing time for a combination of grid sequencing and adaptive coupling
Initial Solution:
26 blocks, coarse grid total time: 190s
Computed Residual (CR): 10^-10
Stage 1:
68 blocks grid total time: 632s
40 blocks grid total time: 642s
26 blocks grid total time: 749s
CR: 10^-4
Stage 2:
68 blocks grid total time: 782s
40 blocks grid total time: 756s
26 blocks grid total time: 760s
CR: 10^-5
Stage 3:
68 blocks grid total time: 1008s
40 blocks grid total time: 1005s
26 blocks grid total time: 993s
CR: 10^-7
Computing time without grid sequencing
Explicit
26 blocks grid total time: 5938s
Implicit
68 blocks grid total time: 2120

The 26 block explicit computation is slightly faster than the 40 and 68 block computations because of reduced communication overhead.

It should be noted that the 68 block implicit computation is faster than the 26 and 40 block computations.

Figure 14: Grid sequencing, in combination with adaptive coupling was used in the Navier-Stokes computation of the NACA0012 airfoil. The thick-lined boxes show the optimum sequence: in stage 1, the 68 block computation is fastest. However, in stage 2 the 40 block computation is fastest while in stage 3 the 26 block grid gives the lowest computing time. This clearly shows the speedup achievable by the adaptive coupling strength strategy.

5.2.4 *Acceleration by Dynamic Loadbalancing*

The 68 block grid of the NACA0012 airfoil configuration was used to show the influence of dynamic loadbalancing on total run time for the Navier-Stokes computation on 9 processors of the Silicon Graphics Power Challenge XL. The two curves in Fig. 15 show the comparison between runs with and without dynamic loadbalancing. Both simulations were performed using the same numerical strategy. A reduction of 7% in computing time was achieved. While the two strategies require the same resources for the explicit part of the computation, dynamic loadbalancing reduces the cost of the implicit part by 10% in this example. A higher percentage for 3D computations is expected.

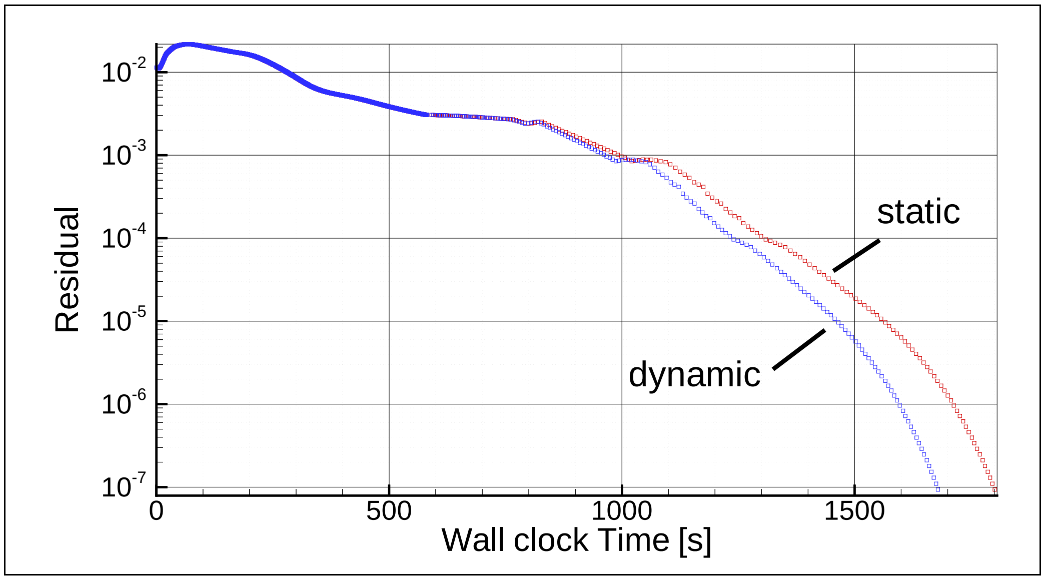


Figure 15: The graph shows the influence of dynamic load balancing on the total computing time for a Navier-Stokes computation of the 68 block NACA0012 airfoil. A total reduction of 7% in computing time was achieved.

5.3 *Huygens Space Probe Flow Simulation*

In Fig. 16 an Euler flow solution for the ESA/NASA Huygens space probe is shown. This probe was launched in 1997 and will enter Titan's atmosphere in 2004 measuring its composition. The instruments on the windward side have been modeled to simulate microaerodynamics effects during the entry phase.

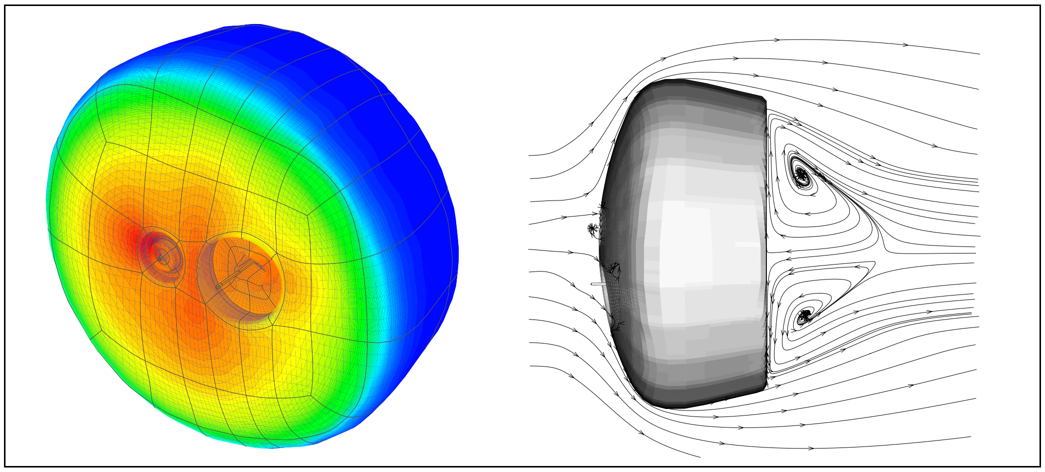


Figure 16: The Mach 3.1 Euler flow was computed for the Huygens Space Probe. A 462 block grid comprising 561,654 grid points was used. The modeling of the instruments on the windward side of the probe should be noted.

5.3.1 *Acceleration by Domain Decomposition*

Number of blocks	Number of points per block	Number of iterations	Scaled computing time
6	12167	350	1.00
48	1728	351	0.53
384	343	420	0.59

Table 5: Convergence behavior for 3D Huygens Space Probe (see also Fig. 2). Speedup resulting from employing block decomposition as a preconditioner. The results clearly demonstrate that this strategy is successful for large 3D grids.

5.4 *Simulation for an SSTO Configuration*

The SSTO testcase is similar to the X–33 Lockheed Martin configuration (Fig. 17). The Euler equations are solved for a Mach number of 9.8 at angle of attack of 40°. The highly complex 3D geometry is modeled by 274 blocks using 256,268 grid cells. The large number of blocks is caused by the high degree of geometric complexity. The surface grid together with parts of the symmetry plane and the outer boundary are shown in the left plot of Fig. 17. The grid is mirrored about the symmetry plane for visualization purposes. For the computation only half of the vehicle is simulated. The right picture in Fig. 17 shows the Mach number contour plot in the symmetry plane. The surface of the vehicle is shaded.

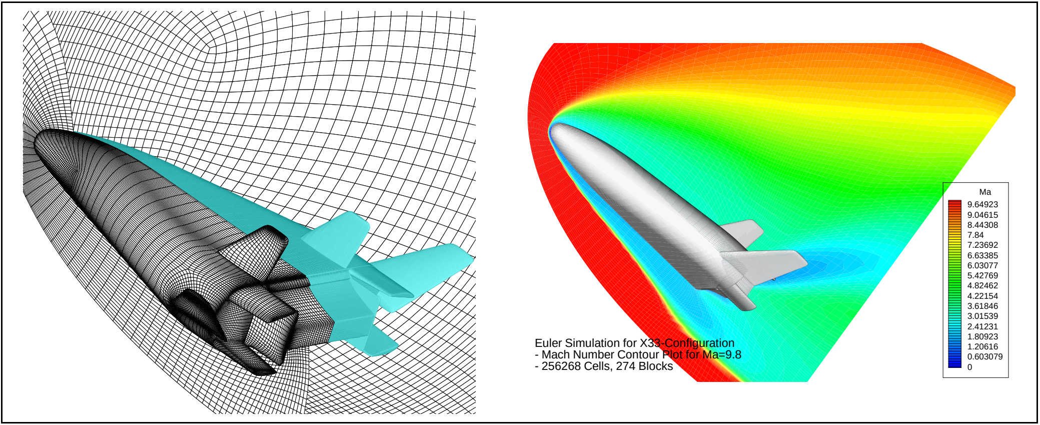


Figure 17: The Tangled Web approach was used to compute the inviscid flow at Mach 9.8 for this highly complex SSTO configuration. The grid comprises 274 blocks (subdomains) of widely different size with a total of 342,361 grid points. The grid is almost orthogonal everywhere and grid point clustering does not extend into the far field. This is a result of the topology chosen.

5.4.1 *Acceleration by Grid Sequencing*

The influence of grid sequencing on the total computing time is given in Table 6. The Euler solution is computed by the explicit scheme. If grid sequencing is applied, a total of $605s + 5,027s = 5,632s$ is needed to obtain the steady state solution that has to be compared with the $7,166$ seconds needed for a computation on the fine grid only. Thus, a speedup of 1.43 is achieved. For this computation the residual is reduced by 6 orders of magnitude on both the coarse and the fine grids. Only two grid levels are used, because the next coarser grid would comprise about 4,600 cells and thereby would no longer be sufficient to represent the surface geometry of the vehicle.

Grid level	**Cells**	**Computing time in s**
1	34196	605
0	256268	5027
0 (no grid sequencing)	256268	7166

Table 6: Results for solution acceleration by using multilevel grids for the modified X-33 SSTO configuration.

5.4.2 *Acceleration by Domain Decomposition*

A 331 block grid was generated from the 274 block grid by splitting larger blocks into sub-blocks. It should be noted that both grids have exactly the same number of cells as well as identical coordinates. However, the number of halo cells is larger for the 331 block grid. Hence, more communication occurs on the network which makes the update of the halo cells somewhat more costly. Two numerical experiments were carried out to illustrate the influence of varying coupling strength for this configuration. First, the Euler flow was computed for the coarse 274 and 331 block grids using the explicit scheme to provide a good starting solution for the implicit *GMRES* algorithm. The steady state solution was then obtained by the implicit *GMRES* algorithm in combination with dynamic load balancing. Table 7 gives the results for this comparison. Only minor speedup was observed following the increase of the number of blocks from 274 to 331.

5.4.3 *Parallel Efficiency*

Results for parallel efficiency are given in Fig. 18. An almost ideal speedup for the 274 block grid of the X-33 configuration is obtained on the 10 Processor Power Challenge XL. The explicit scheme was used to measure the performance of the message passing. It should be noted that the ratio of computation to communication is lower for the explicit than for the implicit

Number of blocks	Number of cells	Computing time in s	Speedup
274	34196	1223	1.00
331	34196	1264	0.97

Table 7: Results for solution acceleration by variation of block number for the modified X-33 SSTO configuration.

scheme, resulting in a better parallel scalability for the implicit scheme. The load balancing algorithm explicitly takes into account that a mixture of 75 Mhz and 90 Mhz R8000 processors is used. The coarse mesh has a total number of 256,268 cells. The fine mesh was generated by doubling the number of cells in each direction. Hence the total number of cells is 2,050,144.

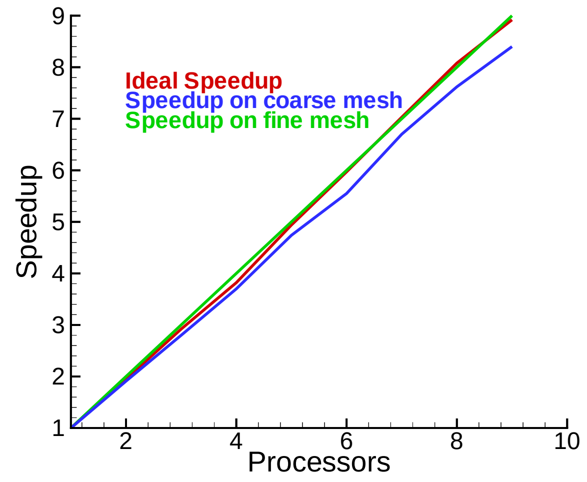


Figure 18: Parallel Efficiency of explicit X-33 computation on 10 processor Silicon Graphics Power Challenge XL. The speedup is almost linear.

5.4.4 *Dynamic Block Management based on Residual Values*

The rate at which the flow field changes during the course of the computation is different for individual blocks in a multiblock grid. In hypersonic flow computations it is common practice to use freestream conditions as initial solution. This means that during the first few iterations flow variables are changed only in blocks with physical boundary conditions, e.g. wall, inflow, or outflow conditions. For this reason, only these blocks need to be updated. It would not make any difference if all other blocks were switched off. However, these deactivated blocks still need to be involved in the update process via block boundaries. Using the update information they can determine whether or not to activate themselves. On the other hand, active blocks are deactivated if their local residual is much smaller than the global residual and if no sufficient change in the solution is transported across block boundaries.

First investigations have been made to implement this dynamic behavior into the *ParNSS* code. Figs. 19 and 20 illustrate this dynamic behavior for the SSTO vehicle simulation.

The first row of plots in Fig. 20 shows all of the blocks that are active in the beginning of the computation. At this stage, only those blocks that have at least one fixed boundary face are active. During the course of the simulations additional blocks are turned on, because information propagates from active blocks into non-active blocks. Towards the end of the computation (Fig. 19) the residual in some of the blocks indicates that a steady-state solution has been reached, automatically switching off these blocks, while other blocks have not converged. Finally, almost all blocks are switched off and the computation is stopped.

6 *Conclusions and Future Work*

In this article a strategy has been presented to achieve both parallel and numerical scalability for the solution of the Navier-Stokes equations with complex flows in complex geometries. The strategy consists of using numerical and computational accelerators such as grid sequencing, domain decomposition, adaptive coupling, dynamic load balancing, and block deactivation.

The *ParNSS* Navier-Stokes code has been used to demonstrate parallel scalability for these test cases, as well as the scalability of the acceleration strategies.

ParNSS has also been used for numerical scalability studies, by implementing the *Tangled Web* collection of acceleration strategies. We computed Euler and Navier-Stokes airfoil flows, simulations for the Huygens space probe

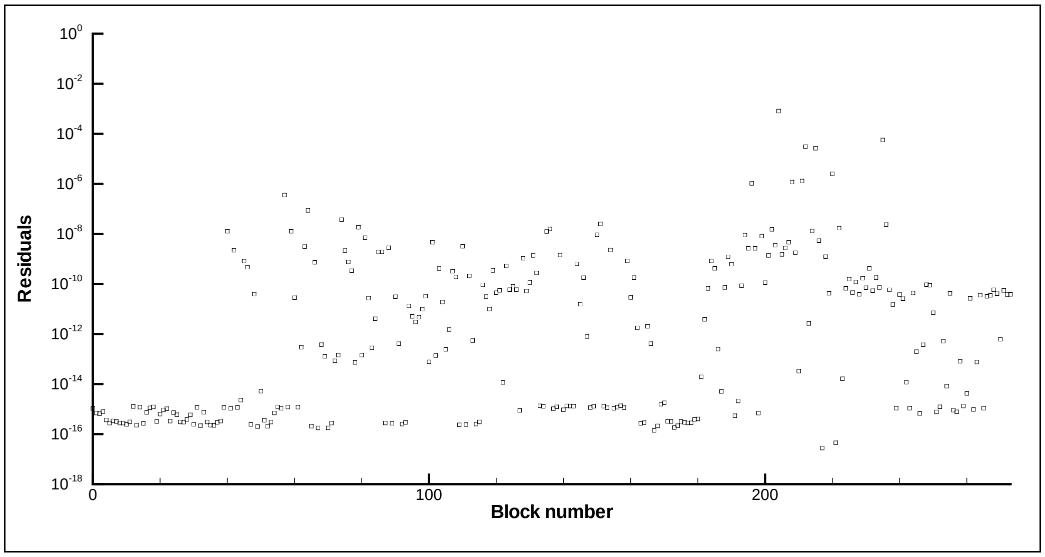


Figure 19: The figure shows the distribution of the residual values (norm of the flux) versus block number for the 274 block X-33 configuration when the solution is close to convergence. The residuals among blocks differ by up to 13 orders of magnitude. By switching off those blocks having a low residual value, computation cost can be reduced substantially. The *ParNSS* code automatically determines those blocks to be switched on/off during the computation. Dynamic load balancing is used to provide an equal workload for each processor.

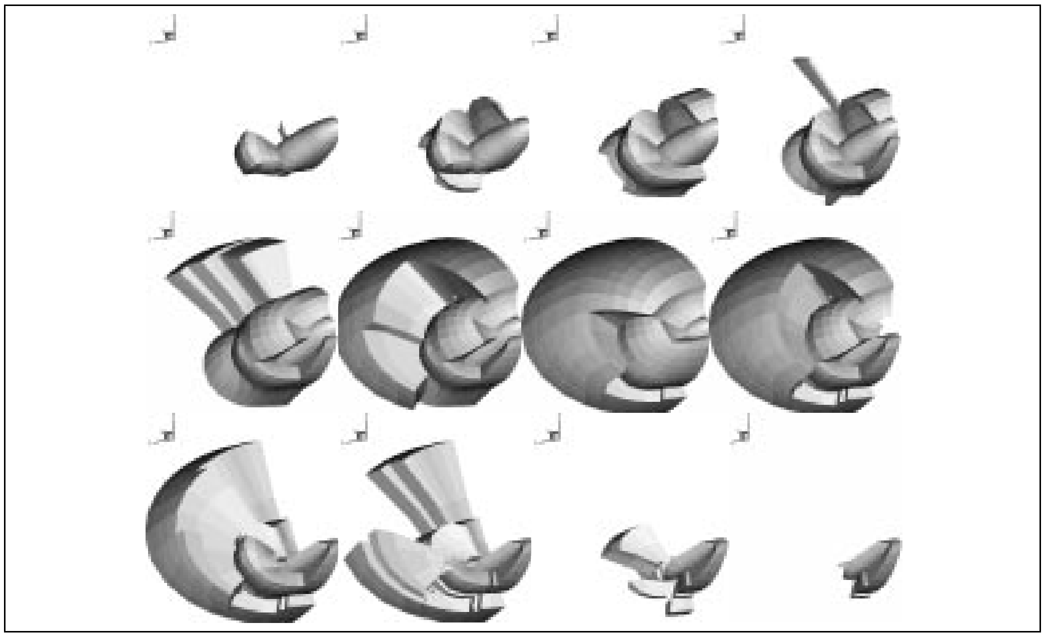


Figure 20: The 12 plots illustrate the dynamic behavior of a *ParNSS* simulation with regard to dynamic activation and deactivation of blocks during a run.

and an SSTO configuration. In each case, speedups of 1.5 to 3 have been obtained, and we have shown that combining these strategies according to the Tangled Web methods multiplies the speedups, as expected.

Heuristic algorithms have been used to increase the adaptive coupling of grid points during the solution process. In the next stage, a more formal approach will be taken to forecast or estimate the condition number to improve the adaptive coupling strategy. These kind of automatic transitions between the numerical strategies form the core of our future research: deciding when to change to a coarser or finer grid, when to split blocks, when to change from explicit to implicit schemes, when to do load balancing, which blocks ahould be active. Further ahead, we hope to dynamically switch between physics modules, with or without turbulence, with or without chemistry or ionization physics.

The dynamic load-balancing strategy was one of the most difficult to implement, both conceptually and technically, yet the speedup was only 1.07: we believe that this is because the test-cases are too simple. Since the cost of doing the dynamic load balance is roughly constant, and the benefit increases with workload imbalance, then the greater the imbalance, the greater the speedup. The load-imbalance is not as sharp as it would be, for example, with strong shocks moving through the fluid, with the great disparity in length scales charactaristic of complex flows, or with chemistry and turbulence models being switched on and off. Dynamic load balancing for three-dimensional testcases will be investigated, too. Additional numerical experiments for large 3D examples will be performed to demonstrate the viability of the *Tangled Web* approach.

We are also working on an environment for distributed computing on a client-server basis as well as an internet based environment for integration of multidisciplinary codes. Investigations are being conducted on the use of Java threads to boost utilization of parallel resources, and on leveraging *ParNSS* to a distributed, collaborative design tool.

Acknowledgement

The authors are grateful to *Prof. Mark Cross*, University of Greenwich, London, for numerous stimulating discussions.

The authors are particularly grateful to *Yang Xia*, CLE for providing the complex grids used in this article.

The continuous support by *Jean Muylaert* and *Martin Spel*, ESTEC, ESA, Noordwijk, The Netherlands is gratefully acknowledged.

This work was partly funded by the *Ministerium für Wissenschaft und Kultur, Niedersachsen, Deutschland*, AGIP Projekt F.A.Nr. 1997.262.

References

[1] Gustavson, J. L., Montry, G. R. and Benner, R. E, *Development of Parallel Methods for a 1024-Processor Hypercube*, SIAM J. Sci. Stat. Comp., 9 (1988) 4.

[2] Fox, G. C., Williams, R. D. and Messina, P. C., *Parallel Computing Works!*, Morgan Kaufmann, 1994.

[3] EUROPORT, *http://www.gmd.de/SCAI/europort-1*, 1996.

[4] Walshaw C., Cross M., and Everett M, *Mesh partitioning and load-balancing for distributed memory parallel systems*, In B. Topping, editor, Proc. Parallel & Distributed Computing for Computational Mechanics, Lochinver, Scotland, 1997, 1997.

[5] Eiseman P.R. et al., 1998: *GridPro v3.1, Users's Guide and Reference Manual*, 111 pp., 2nd edition, Program Development Corporation of Scarsdale, Inc.

[6] Häuser, J. et al., 1994: *ParNSS : An Efficient Parallel Navier-Stokes Solver for Complex Geometries*, AIAA 94-2263, 9pp.

[7] Häuser, J. and R. D. Williams, *Strategies for Parallelizing a Navier-Stokes Code on the Intel Touchstone Machines*, Int. J. Num. Meth. Fluids, 15 (1992) 51-58.

[8] Willebeek-Le Mair, M.,Reeves, *A Strategies for dynamic load balancing on highly parallel computers, IEEE Trans. on Parallel and Distributed Systems 4 1993 979-993*

[9] Williams, R. D., *Voxel Databases: A Paradigm for Parallelism with Spatial Structure*, Concurrency, 4 (1992) 619, 1992.

[10] Häuser, J., J. Muylaert, H.-G. Paap, M. Spel, and P.R. Eiseman, *Grid Generation for Spaceplanes*, 3rd Space Course, University of Stuttgart, Germany, February 20– March 3, 1995, 66pp.

[11] Häuser, J., J. Muylaert, and Y. Xia, 1996: *Grid Generation for the Halis Configuration* in Numerical Grid Generation for Computational Fluid Dynamics, eds. B. Soni et al, MSU Press, USA

[12] Jameson, A., Yoon S.: *Lower Upper Implicit Schemes with Multiple Grids for the Euler Equations*, pp. 929-935, AIAA Journal, Vol. 25, No. 7, 1987

[13] Häuser, J., Williams, R.D., Winkelmann, R., *Parallel Implementation of Large CFD Codes*, European Shortcourse on Strategies and Tools for Parallelising Large Computational Mechanical Codes, 1996, 130 pp., available through F.Barkshire@gre.ac.uk.

[14] Whitfield, David L., 1990: *Newton-Relaxation Schemes for Nonlinear Hyperbolic Systems*, MSSU-EIRS-ASE-90-3, 14 pp.

[15] Whitfield David L.: *Perspective on Applied CFD*, AIAA-95-0349, 33rd Aerospace Sciences Meeting and Exhibit, January 9-12, 1995, Reno, NV, USA

[16] Golub,G., J.M. Ortega, 1992: *Scientific Computing and Differential Equations*, Academic Press.

[17] Winkelmann R., 1998, Strategies for Parallel and Numerical Scalability of CFD Codes, PhD Thesis, University of Greenwich, London, UK.