Parallel Computing for Applications in Aeronautical CFD

by

Anders Ytterström

Report No. 2001-15

Doctoral Thesis
Stockholm, 2001
Parallel Computing for Applications in Aeronautical CFD

by

Anders Ytterström

Report No. 2001-15
ISSN 0280-4646

Akademisk avhandling
som med tillstånd av Kungliga Tekniska Högskolan i Stockholm framläggas till
offentlig granskning för avläggande av teknisk doktorsexamen i Flygteknik.
Abstract

A pre-processor tool, MB-Split, has successfully been developed in order to achieve good load balance on parallel computers for the parallelized CFD-solver NSMB. MB-Split was later coupled to NSMB to get dynamic load balance possibilities. An efficient scheme for time accurate calculations was implemented in NSMB, the so called dual time stepping. NSMB was also adopted and extended to be able to run internal flow cases.

A number of demanding flow cases has been run, in order to test how the above tools and implementations work for real-life industrial problems. The solutions are analyzed with regard to accuracy and computational efficiency. Four turbulence models and three different numerical schemes have been used, on four different parallel computers.

A comparison in calculation time for the dual timestepping scheme and a classical explicit Runge-Kutta scheme for the realistic 3D, inviscid, JAS case, showed that the dual timestepping scheme was more than 3300 times faster than the Runge-Kutta scheme.

The largest problem for the development of a correct load balancing algorithm is the difficulty to correctly predict the performance on different blocks. Computational time for a block is not always proportional to the number of elements in a block, because of cache-misses on cache-based computers, different amount of work performed for different boundary conditions, dummy ghost cells, etc. MB-Split gives a fairly good estimate in most cases, but large deviations from predictions have been observed. A dynamic tool, where the actual computational time is continuously measured, was needed in order to get a better prediction for all possible cases.

The dynamical load balance implemented by close coupling MB-Split to NSMB measures actual calculation time by doing a single time step for each load balancing suggestion. A number of load balancing suggestions can be produced by varying one or several parameters in the load balancing algorithm, or by using the time achieved in the previous test calculation.

The close-coupled MB-Split/NSMB is, as a spin-off benefit, now much easier to run: all is done automatically by just starting NSMB with the number of processors as a command line argument.
Preface

This thesis consists of five papers and a summary of the work presented. The five papers are:

**Paper I**

*A Tool for Partitioning Structured Multiblock Meshes for Parallel Computational Mechanics*

*Anders Ytterström, 1996*


**Paper II**

*Navier-Stokes Calculations on a Full Configuration Aircraft using Parallel Computers*

*Anders Ytterström, Carlos Weber, Arthur Rizzi and Jan Vos, 1997*


**Paper III**

*Analysis of Static Distortion in the SAAB 105 Inlet, using Different Turbulence Models within the Navier-Stokes Solver, NSMB*

*Anders Ytterström, 1997*


**Paper IV**

*Hammershock Calculations in the Air Intake of JAS 39 GRIPEN, using Dual Timestepping*

*Anders Ytterström and Emil Axelson, 1999*


**Paper V**

*Dynamic Load Balancing for CFD-solvers running on Parallel Computers*

*Anders Ytterström, 2001*

Technical report, Department of Aeronautics, KTH.
Division of Work Between Authors

Paper II
Ytterström wrote most of the paper, performed the calculations and did some implementation in NSMB in order for the test case to work, mainly parallelization aspects.

Paper IV
Ytterström wrote the paper, performed most of the time accurate calculations and did all implementations in NSMB and MB-Split.
Contents

Abstract i

Preface iii

Division of Work Between Authors iv

1 Introduction 1
   1.1 Parallel Computers ........................................ 1
   1.2 Parallelization ........................................... 2
      1.2.1 Parallel CFD-solvers ................................ 2
   1.3 Load balancing .......................................... 3
      1.3.1 Dynamic Load Balance ................................ 3
   1.4 Time Accurate Calculations ............................... 4
   1.5 Structure of Thesis ....................................... 5

2 NSMB 6
   2.1 Numerical Scheme ......................................... 6
      2.1.1 Dual Timestepping ................................... 6
   2.2 Program Structure ......................................... 10
   2.3 Parallelization Strategy .................................. 10
   2.4 RISC Optimization ........................................ 11

3 MB-Split 11
   3.1 Static Load Balancing ..................................... 12
      3.1.1 Recursive edge bisection for homogeneous machines 12
      3.1.2 Greedy Load Balancing ................................ 13
      3.1.3 GreedyXtra Load Balancing ............................ 15
      3.1.4 Remarks ................................................ 17

4 Dynamic Load Balance - Coupling NSMB with MB-Split 18
   4.1 Outline of program structure .............................. 19
   4.2 Dynamic load balancing algorithm ........................ 21
   4.3 Implementation issues .................................... 22
      4.3.1 Compiler issues ....................................... 22

5 Test cases and Results 23
   5.1 AS28G ..................................................... 23
   5.2 SAAB 105 .................................................. 26
   5.3 Hammershock calculation in Air Intake of JAS 39 Gripen 27
      5.3.1 Mesh and Flowcase Description ...................... 28
      5.3.2 Steady-state Calculation ............................. 28
      5.3.3 Time-Accurate Hammershock Calculation ............. 30
5.3.4 Results ........................................... 34
5.4 Dynamic Load Balance ............................ 34
5.5 A-airfoil ........................................... 35

6 Summary and Conclusion .......................... 35

Acknowledgment ...................................... 37

References ........................................... 39

Abstracts of Papers .................................. 43

Paper I: A Tool for Partitioning Structural Multiblock Meshes for Parallel Computational Mechanics

Paper II: Navier-Stokes Calculations on a Full Configuration Aircraft using Parallel Computers

Paper III: Analysis of Static Distortion in the SAAB 105 Inlet, using Different Turbulence Models within the Navier-Stokes Solver, NSMB

Paper IV: Hammershock Calculations in the Air Intake of JAS 39 GRIPEN, using Dual Timestepping

Paper V: Dynamic Load Balancing for CFD-solvers running on Parallel Computers
Parallel Computing for Aerodynamic Applications

Anders Ytterström

1 Introduction

A growing number of industrial flow problems can today be solved and understood by numerical computations. Numerical calculations are often made as a complement to experiments and theoretical analysis, but can also be the only method suitable for a certain problem, for economical or technical reasons. One reason for the increasing use of numerical calculations is the development of robust and efficient numerical methods, another is the rapid drop of price/performance for high performance computers. This thesis is about efficient tools for Computational Fluid Dynamics (CFD) on modern parallel computers of both vector and RISC type.

Major contributions to the recent trend towards parallel CFD computations are the maturing hardware, the development of useful and stable software, and standardization and portability of message passing libraries (MPI, PVM, etc.). This has encouraged the development of parallel implementations of commercial CFD-solvers as well as solvers on a research level, for unstructured- as well as block-structured meshes.

1.1 Parallel Computers

Parallel computers such as SGI Origin, IBM SP2 and Cray T3E can not only be found at universities and research institutes, but also in industry. In recent years, clusters of cheap-of-the-shelf, PCs has been quite common, using Linux as their operating system and networks that range from standard 100 mbit fast Ethernet to more expensive systems, for example Myrinet. The driving force is the better performance and price/performance for parallel computers, compared to traditional single processor vector computers.

A parallel computer can be implemented in different ways, but one common architecture is to have separate processors with distributed memory, like clusters of workstations, IBM's SP2, and similar solutions. Another way is to make the memory accessible from all processors, like SGI's Origin and PCs with SMP-mounted processors, or part of the memory jointly accessible from some processors and the rest of the memory distributed as IBM SP's Nighthawk-nodes.

A processor on a 16-processors Nighthawk-node can access all memory on the node it is situated at, but not on other nodes, while an Origin processor can access all memory
on the whole machine, but with different efficiency. The best memory bandwidth on an Origin-server comes from the memory directly connected to the processor, then comes memory connected to other processors in the same cabin and slowest access comes from memory placed in other cabins.

1.2 Parallelization

Parallelization of programs can be done in different ways for all these architectures. It is easiest on the machines where all memory is accessible from all processors. Fine-grained parallelization of the loops, can be done automatically by the compiler, or a little more efficiently, with compiler-directives in the source code. This approach is not so demanding regarding the time spent in parallelizing the solver, and it seldom gives a good result when using many processors. It will not work at all for parallel computers with distributed memory like PC-clusters, where you don’t have direct access to all memory.

A much more useful, efficient and portable way of parallelization is to use message-passing, where each processor only accesses memory with the highest bandwidth and then messages are sent between the processors. The most common message-passing library is MPI(Message Passing Interface)[1]. MPI is an industry standard and used in NSMB[2, 3], the main solver used in this thesis. PVM(Parallel Virtual Machine) is another common message-passing library, but it is not an official standard and its popularity is degrading. Other possibilities exist, like using HPF(High Performance Fortran), but these approaches are usually less efficient.

1.2.1 Parallel CFD-solvers

The parallelization approach used for NSMB is domain decomposition, where the computational domain is divided into a number of subdomains, or grid-blocks and each processor takes care of one or more subdomains/blocks. Then the internal boundaries between the blocks are updated by sending messages using MPI.

This is a very common way of parallelizing a CFD-solver using MPI and it is also fairly simple, depending on the internal structure of the solver. Solvers using unstructured grids, have the computational cells and/or points in a list with pointers or addresses to neighbours. It is therefore easy to create subdomains of arbitrary cell assemblies. It is usually a little more complicated with solvers using structured grids, as the cells/points are stored in arrays and neighbours are addressed by a certain index, which is not known by the individual cell by pointers or addresses. It can also be more restricted on where it is possible to split the blocks, due to physical constraints in the flow and numerical considerations.
1.3 Load balancing

One of the most important aspects in domain decomposition, is to distribute the blocks on the parallel machine in order to have a good load balance between the different processors. The complexity of the block-splitting book-keeping demands an automatic load balancing and block-splitting tool.

It can be quite complicated to achieve a good load balancing of structured grids as, for example, it is not possible to cut away individual cells, the best index-direction to split the blocks has to be considered, etc., etc., in order to get as optimal load balance as possible, and hence the shortest computational time.

Many load balancing algorithms, like Reverse Cuthill McKee (RCM)[4], Recursive Spectral Bisection (RSB)[5], and automatic domain decomposition tools, e.g. TOP/DOMDEC[6], Chaco[7], MeTis[8], etc., have been developed for unstructured meshes. The work on structured multiblock meshes has mainly focused on distribution of the original blocks, without actually splitting any blocks, or splitting them by hand. A demand for an efficient automatic tool inspired the development of the MB-Split[9, 10, 11] load balancing tool for multiblock meshes.

When NSMB first was parallelized, in 1994, it was decided to make the load balancing and block splitting in a pre-processor tool. One of the reasons was the ability to use a programming language more suited for the task, but also due to project considerations as the actual parallelization of NSMB were performed on a number of different places in Europe, but the load balancing part of the project was performed only at KTH. A pre-processing tool would make the updating of NSMB, which is written in Fortran 77, much easier, as the load balancing part could be very large.

The load balancing tool, “MB-Split”, is written in C++. As earlier suspected it has become quite large, around 80 000 lines of code at the moment (April 2001).

When MB-Split is used as a pre-processor it works as a stand-alone database processor and is only coupled to the CFD-solver, through the specific IO-format of the grid and solution for each solver. Three different load balancing algorithms are implemented: Recursive Edge Bisection, Greedy load balancing and GreedyXtra load balancing. The main functionality of MB-Split and how its static load balance algorithms works are described in Paper I, A Tool for Partitioning Structured Multiblock Meshes for Parallel Computational Mechanics, and in section 3.

1.3.1 Dynamic Load Balance

When a pre-processing tool is used, only computer- and mesh-data can be used for the load balancing, which is sometimes not enough. It is sometimes very difficult to consider all things that affect the computation time and it is also impractical to de-
mand that the user should give all data necessary. One example is the computational
time for different types of boundary conditions, another the effect of cache-misses
and vector-lengths. Cache-misses can have a very severe effect on the computational
time, for some cases, up to 5 times slower computational time were measured on an
SP2 due to cache-misses.

Another reason for dynamic load balance is for cases where the flow solution effects
the computational time in different blocks and hence affects the load balance on the
parallel computer.
Due to the above reasons and also due to a demand for easier running of the whole
solution process, a dynamic load balance has been implemented by coupling MB-Split
to NSMB and make all necessary changes in NSMB and MB-Split. The dynamic load
balance takes the actual computational time into account.

1.4 Time Accurate Calculations

Time-accurate calculations are of great importance, as many flow problems are of a
timedependent nature. The largest problem with time-accurate calculations are the
very large computation time for realistic cases. Another problem is moving grids and
for adaptive grids, regeneration of the grid due to a changing flow field.

Local timesteps has often been used to increase the convergence rate for steady-state
calculations. This and some of the other methods used for steady-state calculations,
like pre-conditioning, some time-stepping schemes, etc., could not be used in the
time-accurate case. Explicit Runge-Kutta time-stepping, using global timesteps, has
traditionally been the most used method for time-accurate calculations. The largest
disadvantage with this method is the fact that the global timestep usually becomes
very small, due to stability restrictions, as the global timestep is chosen as the small-
est local timestep in the total mesh. Accuracy restrictions are often much larger in a
realistic flowcase. Especially for viscous calculations with a large Reynold’s number
and, hence, very stretched grids, but also for inviscid calculations with complicated
geometries.

One way to be able to use a larger timestep is to use an implicit timestepping scheme,
which has a much larger stability region, compared to the explicit Runge-Kutta
scheme. However, solving the system of equations resulting from a truly implicit
scheme is usually very demanding regarding both cpu time and memory require-
ments. Another aspect is the amount of work to implement such a scheme.

A method to come around at least the memory requirement and implementation prob-
lems has been suggested by Jameson[12], which involves dual timesteps. This method
uses existing algorithms in Navier-Stokes solvers, including convergence rate accelera-
tion methods like local timesteps, multigrid, etc.. Instead of solving the Navier-Stokes equations these solvers are slightly modified to solve the system of equations resulting from an implicit formulation of the Navier-Stokes equations.

This thesis and especially Paper IV Hammershock Calculations in the Air Intake of JAS 39 GRIPEN, using Dual Timestepping, describes the theory of the, so called, dual timestepping approach, as well as implementation notes. There are also some examples comparing the traditional Runge-Kutta timestepping approach, using global timesteps, to the dual timestepping method. One of the examples involves an inviscid 3D hammershock calculation in the air intake of JAS 39 Gripen.

1.5 Structure of Thesis

Paper I: A Tool for Partitioning Structured Multiblock Meshes for Parallel Computational Mechanics, describes how NSMB is parallelized, a more in depth description of MB-Split and the whole solution process from the original database to the solution, via an intermediate database and all input files.

NSMB’s capability to solve large flow problems with the approach described in Paper I, on parallel computers is demonstrated and analyzed in Papers II and III, Navier-Stokes Calculations on a Full Configuration Aircraft using Parallel Computers and Analysis of Static Distortion in the SAAB 105 Inlet, using Different Turbulence Models within the Navier-Stokes Solver, NSMB. Both parallel efficiency and solution accuracy are investigated. Paper III handles also some issues concerning calculations on air intakes, and necessary adaption done to NSMB to be able to handle internal calculations.

Paper IV Hammershock Calculations in the Air Intake of JAS 39 GRIPEN, using Dual Timestepping is mainly about the, so called, dual timestepping approach and its implementation in NSMB. It also describes how the dual timestepping works on a couple of testcases and why this scheme was necessary for the large time accurate calculation performed on the air intake of JAS 39 Gripen.

In order to make the parallel efficiency even better and also to make the solution process a lot simpler, a dynamical load balancing algorithm were implemented, which is described in Paper V: Dynamic Load Balancing for CFD-solvers running on Parallel Computers.

This summary of the five papers does not go through the different papers one by one. Instead it describes NSMB and mainly the parts of NSMB that I have been involved in and MB-Split, which has been totally developed by me. Some of the testcases from the five papers are also described.
2 NSMB

2.1 Numerical Scheme

NSMB solves the Reynolds average Navier-Stokes equations on multiblock structured grids. The Navier Stokes equations are discretized using a cell centered finite volume scheme. Both a central scheme and several types of upwind schemes are implemented, including different types of TVD\textsuperscript{1}-limiters. Time discretization is made with either an explicit Runge-Kutta timestepping, an implicit LU-SGS\cite{13,14} scheme, or, in the time accurate case, the dual timestepping approach. Runge-Kutta timestepping with global timesteps can also be used for time accurate calculations. Otherwise local timesteps are used, for convergence acceleration.

Various turbulence models of different complexity are available in NSMB which range from algebraic models to one- and two-equation models. Four different turbulence models have been used within this thesis: the algebraic Baldwin-Lomax\cite{15} and Granville\cite{16} models, the one-equation model by Spalart-Allmaras\cite{17} and the two-equation k-\(\varepsilon\) model by Chien\cite{18}.

2.1.1 Dual Timestepping

The idea behind the dual time-stepping technique, is to have an outer time-stepping loop for a real time-accurate timestep using a fully implicit scheme, and an inner time-stepping loop, with a fictitious timestep, to solve the system at each real timestep. Local timesteps, multigrid and other techniques for acceleration of convergence can be used in the inner loop to converge the solution to a “steady-state” within each real timestep. This implementation follows the technique presented by Jameson\cite{12}.

The Navier-Stokes equations are given in the following discrete form as a set of ordinary differential equations:

\[
\frac{d}{dt} (V W) + R(W) = 0
\]

(1)

or:

\[
\frac{d}{dt} (V^{n+1} W^{n+1}) + R(W^{n+1}) = 0
\]

(2)

where \(V\) = cell volume and \(W\) = state vector.

The \(\frac{d}{dt}\) operator is approximated by an implicit Backward Difference Formula (BDF) operator of \(k\)th order accuracy:

\[
\frac{d}{dt} = \frac{1}{\Delta t} \sum_{q=1}^{k} \frac{1}{q} [\Delta^{-}]^{q}
\]

(3)

\textsuperscript{1}TVD = Total Variation Diminishing
, where $\Delta^{-} w^n = w^n - w^{n-1}$

So the second order scheme ($k = 2$ in Eq(2) ) becomes:

$$\frac{1}{\Delta t} \Delta^{-} \left( V^{n+1} W^{n+1} \right) + \frac{1}{2 \Delta t} \left( \Delta^{-} \right)^2 \left( V^{n+1} W^{n+1} \right) + R\left( W^{n+1} \right) =$$

$$\frac{1}{\Delta t} \left( V^{n+1} W^{n+1} - V^n W^n \right) + \frac{1}{2 \Delta t} \left( V^{n+1} W^{n+1} - V^n W^n \right) - $$

$$\frac{1}{2 \Delta t} \left( V^n W^n - V^{n-1} W^{n-1} \right) + R\left( W^{n+1} \right) = 0$$

$$\Rightarrow$$

$$\frac{3}{2 \Delta t} \left[ V^{n+1} W^{n+1} \right] - \frac{2}{\Delta t} \left[ V^n W^n \right] +$$

$$\frac{1}{2 \Delta t} \left[ V^{n-1} W^{n-1} \right] + R\left( W^{n+1} \right) = 0 \quad (4)$$

A modified residual, $R^* (W)$, is defined using the above discretization in time, as

$$R^* (W) = \frac{3}{2 \Delta t} \left[ V^{n+1} W \right] - \frac{2}{\Delta t} \left[ V^n W^n \right] +$$

$$\frac{1}{2 \Delta t} \left[ V^{n-1} W^n \right] + R\left( W \right) \quad (5)$$

and the system to solve is $R^* (W) = 0$.

A new system of ordinary differential equations, similar to Eq(1), is formulated using the modified residual and in a fictitious time, $t^*$:

$$\frac{d}{dt^*} (V W) + R^* (W) = 0 \quad (6)$$

Equation (6) is marched to steady-state in the fictitious time $t^*$, using any convergence acceleration method implemented in the flow solver, with the slightly modified residual in Eq(5).

The implementation in NSMB follows an alternative formulation to (3) and (4), for an arbitrary two-step scheme:

$$\frac{\alpha V^{n+1} W^{n+1} + \beta V^n W^n + \gamma V^{n-1} W^{n-1}}{\Delta t} +$$

$$\lambda_1 R\left( W^{n+1} \right) + \lambda_2 R\left( W^n \right) + \lambda_3 R\left( W^{n-1} \right) = 0 \quad (7)$$
Setting $\alpha = \frac{3}{2}$, $\beta = -2$, $\gamma = \frac{1}{2}$, $\lambda_1 = 1$, $\lambda_2 = \lambda_3 = 0$ gives the second order BDF scheme, $\alpha = 1$, $\beta = -1$, $\gamma = 0$, $\lambda_1 = \frac{1}{2}$, $\lambda_2 = \frac{1}{2}$, $\lambda_3 = 0$ gives the second order trapezoidal scheme, $\alpha = 1$, $\beta = -1$, $\gamma = 0$, $\lambda_1 = 1$, $\lambda_2 = \lambda_3 = 0$ gives the first order implicit backward Euler scheme, etc.. This makes it very easy to change between different schemes.

If an automatic timestep control is required, the parameters in (7) can’t be constant. Following Beam and Warming [19] a general two-step method with varying timesteps can be formulated as:

$$
(1+\xi) \frac{V^{n+1}W^{n+1} - V^n W^n}{\Delta t^n} - \xi \frac{V^n W^n - V^{n-1} W^{n-1}}{\Delta t^{n-1}} + \\
\theta R^{n+1} + (1 - \theta + \phi) R^n + \phi R^{n-1} = 0
$$

(8)

where $\Delta t^n$ is the timestep between $t^n$ and $t^{n+1}$.

The terms $R^n$ and $R^{n-1}$ are not used for the second order BDF method, thus $\phi = 0$ and $\theta = 1$ for this scheme.

In order to check if (8) is of second order accuracy a Taylor expansion is performed at $t^{n+1}$, with $\Delta t^n = \Delta t$ and $\Delta t^{n-1} = \alpha \Delta t^n = \alpha \Delta t$:

$$
W^n = W^{n+1} - \Delta t W_t^{n+1} + \frac{1}{2} \Delta t^2 W_{tt}^{n+1} + O \left( \Delta t^3 \right)
$$

(9)

$$
W^{n-1} = W^{n+1} - (1 + \alpha) \Delta t W_t^{n+1} + \\
\frac{1}{2} (1 + \alpha)^2 \Delta t^2 W_{tt}^{n+1} + O \left( \Delta t^3 \right)
$$

(10)

Substituting (9) and (10) into (8), together with $R^{n+1} = -(V^{n+1} W^{n+1})$, (from (2)) and $V^{n+1} = V^n = V^{n-1} = V$, gives the residual:

$$
\left( 1 + \xi + \frac{\xi}{\alpha} - \frac{\xi}{\alpha} - 1 \right) W_t^{n+1} + \\
\frac{1}{2} \left( -1 - \xi - \frac{\xi}{\alpha} + \frac{2\xi}{\alpha} + \frac{\xi^2}{\alpha} \right) \Delta t W_{tt}^{n+1} + O \left( \Delta t^2 \right)
$$

$$
= \frac{1}{2} [\xi (1 + \alpha) - 1] \Delta t W_{tt}^{n+1} + O \left( \Delta t^2 \right)
$$

(11)

This gives the following condition on $\xi$ for second order accuracy:

$$
\xi = \frac{1}{1 + \alpha}
$$

(12)

The second order BDF method is A-stable [20, 21] for constant timesteps, but not for arbitrary changes in timesteps [22, 23, 24], why a conservative approach should
be considered. Sand [25] found for example that BDF-2 is zero-stable for a fixed timestep-ratio, $\gamma$, if and only if $\gamma < 1 + \sqrt{2}$.

One way to ensure stability is to only increase the timestep when it can be doubled and make use of an extra saved time level, in order to avoid extrapolation. When the timestep needs to be decreased it should always be set to half the previous timestep. In this way there are a need for more timesteps and extra memory for one more time level, but there are no need for extrapolation when the timestep is increased, so the loss is about 30\% [26]. However, the gain in stability could be considerable, which can allow a much larger timestep and definitely less crashes due to divergence. Another way to avoid instabilities is to change the timestep no more often than every second step.

**Starting procedure**

A starting procedure has to be defined when using the second order BDF method, as $W^{n-1}$ is not defined at the very first step. This could be done in a number of ways:

- Set $W^{n-1} = W^n$ and use BDF as usual. However, this does not give a scheme with global second order accuracy.

- Use the explicit Runge-Kutta scheme for the first step and then use BDF.

- Apply the second order trapezoidal scheme, or any other implicit second order scheme that only involves $W^{n+1}$ and $W^n$, for the first step.

- Use the first order backward Euler scheme for the first step.

The first order backward Euler scheme is used in NSMB. It is easy to implement, and has better damping properties than the second order trapezoidal scheme and it still results in a method that is globally second order accurate. As the dual time-stepping approach implemented in NSMB does not use automatic timestep control at the moment, the explicit Runge-Kutta is not an alternative.

**Timestep restrictions**

The term $\frac{3}{2\Delta t}[V^{n+1}W]$ in Eq(5) moves the eigenvalues for the discretized Eq(6) in the direction of the negative real axis, which could be a problem for the Runge-Kutta timestep, when using small $\Delta t$, especially in large cells where $\Delta t^n$ could be large. This is because the stability region for the explicit Runge-Kutta scheme of 3rd, 4th and 5th order is limited in the direction of the negative real axis. One way to limit the problem in large cells is to introduce a limiter for $\Delta t^n$ [27]. Another method is to treat the $\frac{3}{2\Delta t}[V^{n+1}W]$ term implicitly [28]. Neither of these methods are implemented in NSMB.
2.2 Program Structure

NSMB uses the MEM-COM [29] object oriented data management system for memory and memory-to-disk data handling. The MEM-COM library includes a Dynamic Memory Manager (DMM), which offers the possibility to allocate at run time the necessary storage of the arrays in NSMB. When running on distributed memory computers, the DMM allocates the memory on each node of the computer.

Although it is possible to allocate memory dynamically as in a C-program, the implementation in NSMB is still of standard Fortran-type in the way that the main part of the memory is allocated in a few routines, even the temporary memory used in different routines. This saves time for allocation but complicates things as pointers to this memory have to be carried around to all subroutines that use this memory.

There is hardly any difference between a single block or multi block calculation for the explicit Runge-Kutta scheme used in NSMB. It is mainly on the level of the incorporation of the boundary conditions that there appears a difference. The implicit LU-SGS scheme has been implemented using explicit coupling between blocks which means that the implicit time stepping procedure is performed independently block by block. The explicit coupling does not introduce any additional computational costs but the global nature of the implicit scheme is reduced, which could decrease the convergence rate if the blocks gets too many and too small.

Fictitious ghost cells on each side of the computational block are used to transfer information from the boundary conditions (including the block connectivity boundary condition) to the algorithm used to update the interior points.

2.3 Parallelization Strategy

Two design choices were first made in the development of parallel NSMB. First the domain partitioning is executed as a pre-processor, before the execution of parallel NSMB, and secondly, the parallel implementation in NSMB is based on a master/slave paradigm. This latter choice permits an easy porting of NSMB to different parallel platforms, especially those with a front-end node that performs I/O and handles the other nodes. This approach was later changed in favor of a SPMD\textsuperscript{2} implementation, because current parallel machines have no front-end. SPMD is generally more efficient on some computers. Software management becomes easier with only one source code to keep track of. But there is still only one node that performs the I/O and also keep track of convergence, etc.

The data sent between the processors is the statevector and a few other variables for each ghost cell needed on another node, in order for the boundary condition to

\textsuperscript{2}SPMD = Single Program Multiple Data
be updated. This data is sent after each timestep by MPI. After the calculation has finished, the whole solution has to be sent to the node that performs I/O.

Even if NSMB is of SPMD type, there are only one of the nodes that performs all accesses to the MEM-COM data base and input files. This node is called “root-node” and handles also all updating of the convergence-curves, when to stop, etc., except for all I/O.

At the beginning of a Runge-Kutta stage, each processor sends the data for the block connectivity boundary condition to processors holding the neighboring blocks. Blocks on the same processor exchange data without using the network. The calculation of the next Runge-Kutta stage in a block is started as soon as all the data from the neighboring blocks is received. The same procedure applies in the LU-SGS scheme. All data exchange is performed before solving the linear systems.

2.4 RISC Optimization

NSMB was primarily designed for running on vector computers, and the most time consuming routines were written to have vectorizable do-loops over all cells in a block, including ghost cells. The vector performance is hence very good, yielding over 170 Mflops on an old Cray YMP, which is 50% of the peak performance. However, the initial version of NSMB performed rather poorly on computers of RISC type, due to memory and cache contention problems. Permuting array indices of some arrays and other techniques gave a large improvement, with only small modifications to the code. When permuting the array indices, the work array is e.g. changed from: work(1:npoint,1:nwork) to: work(1:nwork,1:npoint). It is now possible to choose between 4 different versions of NSMB, depending on the type of computer to run on. There exists only one version of the source code and the choice of optimization is made at compile-time, as the choice between a parallel or serial version.

A more detailed presentation of NSMB is found in Paper III: Analysis of Static Distortion in the SAAB 105 Inlet, using Different Turbulence Models within the Navier-Stokes Solver, NSMB and Paper IV: Hammersock Calculations in the Air Intake of JAS 39 GRIPEN, using Dual Timestepping.

3 MB-Split

MB-Split can be used in both a user-friendly GUT³-mode and in batch-mode, using an input file. The interactive GUI is implemented using Tcl/Tk[30, 31, 32], which is much easier than directly using X-libraries.

³GUI = Graphical User Interface
The pre-processor approach means that MB-Split reads a mesh from a database, including boundary conditions, block topology and possibly solution, performs load balancing by domain partitioning and finally writes the new mesh into a new database. It is also possible to merge the mesh, with solution, back to the original number of blocks. This is useful for running a flow-case on different computers. It is also useful if calculations started on a coarse grid are continued on a finer grid, with a different number of processors and for post-processing purposes.

The parallel computer environment (e.g. no of processors, memory, performance, etc.), some properties of the parallel solver, and properties of the problem to be computed are described in the text-file, LBCASE, with information about processor assignments being passed on to the parallel solver in another. This other text-file and the output mesh database, are the only communication necessary between MB-Split and the parallel solver, in this case NSMB.

Because of the complexity of updating the block topology, MB-Split is implemented in an object oriented fashion in C++. An object oriented approach makes it, e.g. easy to change and add to a program, without having to change everywhere. The motivation to use C++, before other object oriented languages is that it allows links with Fortran and C and that it is the most common object oriented programming language for UNIX-workstations and supercomputers, which makes it reasonable portable.

### 3.1 Static Load Balancing

Three different load balancing algorithms have been implemented:

- Recursive Edge Bisection
- Greedy Load Balancing
- GreedyXtra Load Balancing

It is also possible to split a block at the desired splitpoint and direction interactively, or using an input file. There is no explicit consideration of communication time in the three load balancing algorithms implemented, only computational time on the different processors. This has been found appropriate on computers with a fast network and a moderate number of blocks and processors.

#### 3.1.1 Recursive edge bisection for homogeneous machines

This algorithm does not take machine or solver characteristics into account. It is very simple and suitable only for homogeneous machines, where the number of processors is larger than the original number of mesh-blocks:
1. Take the largest block (i.e. the greatest number of grid-points) and split it along its longest edge into two blocks of the same size. See figure 1.

2. This goes on until the no. of blocks equals the desired no. of blocks (i.e. the no. of nodes/processors to run on). It is also possible to choose a larger number of blocks in order to enhance the load balance. This could however give an increased amount of work due to the increased number of blocks and hence ghost cells.

![Splitpoint](image)

**Figure 1: Load balancing according to Recursive Edge Bisection**

### 3.1.2 Greedy Load Balancing

This algorithm applies to heterogeneous systems, as well as homogeneous. It is also possible to start with more blocks than processors. So far, it does not explicitly take communication into account, just computation time. Information about the host computer system is a necessary input.

**Performance model**

Processors must synchronize after each step, and the communication time is neglected, so the algorithm tries to minimize $\max_j (t_j)$, where

$$ t_j = \frac{1}{P_j} \sum_{k \in G_j} N_k \times W $$

(13)

$t_j =$ Computation time on processor $j$.

$N_k =$ Number of internal grid-cells for block $k$.

$W =$ Number of float operations per gridcell.
$G_j$ = The set of blocks allocated to processor $j$.
$P_j$ = Float operations per second for processor $j$.

The Greedy Load Balancing Algorithm

1. Compute an “ideal” uniprocessor time, $T_{uni}$.
   
   
   $T_{uni} = \frac{\sum_k N_k \times W}{\sum_j P_j}$ (14)

2. Set the group of unassigned blocks, $G_u$, to contain all blocks. The number of blocks in $N_u = M$, where $M$ is the total number of blocks in the mesh.

3. Start all processors with a very small fictitious block to get a start $t_j = \frac{\delta \times W}{P_j}$, where $\delta \ll 1$ is the size of the very small fictitious block.
   
   This is used to find the fastest processor in a heterogeneous system, $t_j$ could otherwise be set to 0 in the homogeneous case.

4. Main loop:
   
   While $(N_u > 0)$ do
   
   first := $\arg \min_j (t_j)$
   
   $\mathrm{large} := \arg \max_{k \in G_u} (N_k)$
   
   $\tau := \frac{1}{P_{\mathrm{first}}} \left[ \left( \sum_{k \in G_{\mathrm{first}}} N_k \times W \right) + N_{\mathrm{large}} \times W \right]$ ($= t_{\mathrm{first}} + \frac{N_{\mathrm{large}} \times W}{P_{\mathrm{first}}}$)

   if [$\tau > T_{uni} \times (1 + \varepsilon)$] then
   
   rest := $S(\mathrm{large}, \mathrm{first}, T_{uni})$
   
   $N_{\mathrm{large}} := N_{\mathrm{large}} - N_{\mathrm{rest}}$
   
   $G_u := G_u \cup rest$

   $N_u := N_u + 1$

   end if

   $G_u := G_u \setminus \mathrm{large}$

   $N_u := N_u - 1$

   $G_{\mathrm{first}} := G_{\mathrm{first}} \cup \mathrm{large}$

   End of while-loop

Definition of variables and functions:

$G_u$ = The set of unassigned blocks
$N_u$ = Number of unassigned blocks
first = Processor that finishes first
large = Largest unassigned block.
$\varepsilon$ = Small value to avoid creation of thin blocks, typically $\varepsilon = 0.05$

$S(k, j, t)$ = The part of block $k$ that does not fit on processor $j$, if $j$ must finish at $t$, i.e. $t_j = t$.  

14
The only consideration of communication time is the choice to cut across the longest side and a small $\varepsilon > 0$, to avoid creating small blocks.

### 3.1.3 GreedyXtra Load Balancing

The GreedyXtra Load Balancing algorithm is an iterative algorithm which takes extra work done outside the internal grid-cells into account, otherwise it is similar to the Greedy Load Balancing described above. Extra work means work done in the ghost cells and in the handling of blocks. The float operations performed in the ghost cells is given as proportional to the number of float operations in the internal cells, and it is possible to give different parameters for the different directions, $i, j$ and $k$.

The other parameter that is possible to put into the algorithm is the extra work done in the solver proportional to the handling of blocks instead of number of cells.

An advantage with the GreedyXtra Load Balancing algorithm is the possibility to refine the estimates, e.g. work done for handling different boundary conditions, without changing the basic algorithm.

**Performance model**

$$t_j = \frac{1}{P_j} \sum_{k \in G_j} [(N_k + N_{g_k}) \times W + WB]$$

(15)

$N_k = \text{Number of internal grid-cells for block } k$.

$N_{g_k} = \text{Number of ghost cells for block } k \text{ where work is performed. This number is calculated from the parameters given by the user.}$

$WB = \text{Amount of extra work done, as number of extra float operations per block.}$

The optimum uni-processor time becomes:

$$T_{uni} = \frac{\sum_k [(N_k + N_{g_k}) \times W + WB]}{\sum_j P_j}$$

(16)

The GreedyXtra Load Balancing Structure

$T_{uni}$ is here updated after each split, because the number of ghost cells is increased when a block is split, hence the algorithm needs to be iterative. If the new updated $T_{uni}$ is larger than a previous maximum, the algorithm is restarted from the original set of blocks, with the new $T_{uni}$ as maximum. This constitutes the middle loop. A problem with the always increasing $T_{uni}$ is that it could be so large that a block-split is not necessary, and therefore $T_{uni}$ becomes too large for the present number of blocks. This necessitates continued looping in an outer loop. The outer loop restarts with $T_{uni}$ equal to the actual $T_{uni}$ of the present block splitting.
Main structure of the *GreedyXtru* algorithm:

\[
T^\text{new}_{\text{uni}} = \frac{\sum_i \left[ (N_i + N_{g_k}) \times W + WB \right]}{\sum_j P_j}
\]

\[N_u = M\]

\[T_{\text{uni}} = T^\text{new}_{\text{uni}} \times (1 + 2 \times \varepsilon_2)\]

\[i = 0\]

While \([T^\text{new}_{\text{uni}} \times (1 + 2 \times \varepsilon_2) < T_{\text{uni}}] \text{ AND } (i < \text{Maxi})\] do

\[i := i + 1\]

While \((N_u > 0)\) do

\[T_{\text{uni}} := T^\text{new}_{\text{uni}}\]

\[G_u := \text{mesh}_\text{orig}\]

\[N_u := M_{\text{orig}}\]

\[t_j := \delta_{\text{cycle}}\]

While \([N_u > 0] \text{ AND } (T^\text{new}_{\text{uni}} < T_{\text{uni}} \times (1 + \varepsilon_2))\] do

\[\text{first} := \arg \min_j (t_j)\]

\[\text{large} := \arg \max_{k \in G_u} (N_k)\]

\[\tau := \frac{1}{T^-_{\text{first}}} \left[ \sum_{k \in G_{\text{first}}} \left( (N_k + N_{g_k}) \times W + WB \right) + (N_{\text{large}} + N_{g_{\text{large}}}) \times W + WB \right]\]

if \([\tau > T_{\text{uni}} \times (1 + \varepsilon_2)]\) then

\[\text{rest} := S(\text{large}, \text{first}, T_{\text{uni}})\]

\[N_{\text{large}} := N_{\text{large}} - N_{\text{rest}}\]

\[G_u := G_u \cup \text{rest}\]

\[N_u := N_u + 1\]

# Remark: mesh has now changed after the split-operation in function S

\[T^\text{new}_{\text{uni}} := \frac{\sum_i \left[ (N_i + N_{g_k}) \times W + WB \right]}{\sum_j P_j}\]

end if

if \([T^\text{new}_{\text{uni}} < T_{\text{uni}} \times (1 + \varepsilon_2)]\) then

\[G_u := G_u \setminus \text{large}\]

\[N_u := N_u - 1\]

\[G_{\text{first}} := G_{\text{first}} \cup \text{large}\]

end if

End of while-loop (inner)

End of while-loop (middle)

# Remark: \(T_{\text{uni}}\) could now be to large, if the last while-loop gave fewer splits than

# the previous did

\[T^\text{new}_{\text{uni}} := \frac{\sum_i \left[ (N_i + N_{g_k}) \times W + WB \right]}{\sum_j P_j}\]

End of while-loop (outer)

Definition of variables and functions:

\(T^\text{new}_{\text{uni}} = \) “Ideal” uniprocessor time for the current mesh.

\(M = \) Total number of blocks in the current mesh.
\( M_{\text{orig}} \) = Total number of blocks in the original mesh.
\( \varepsilon_2 \) = A small value, typically set to \( \frac{\varepsilon_1}{2} \).
\( i \) = Iteration counter for the outer loop.
\( \text{Maxi} \) = Maximum number of iterations in the outer loop, typically set to 100.
\( \text{mesh}_{\text{orig}} \) = The original mesh, before the block-splitting.
\( \text{mesh} \) = Current mesh, changes after each block-split.

All other variables and functions are the same as for the Greedy Load Balance algorithm (see section 3.1.2), except that equations (15) and (16) are used instead of (13) and (14), for \( t_j/\tau \) and \( T_{\text{uni}}/T_{\text{new}}^{\text{uni}} \) respectively, and that function \( S \) takes extra work in ghost cells, etc., into account, when it splits the largest block, large.

3.1.4 Remarks

Generally the Greedy Load Balance gives a better load balance than the Recursive Edge Bisection. There are some exceptions, but they give so small differences that they do not have to be considered.

As mentioned before there is no consideration of communication time in any of the 3 algorithms, but that does not seem to be a problem on machines with a fast network and solvers making many float operations relative to the amount of data exchanged between processors.

The largest problem is the difficulty to predict the computation time for the different blocks. The performance model in the GreedyXtra algorithm (Eq. 15) works very well for some cases, but for others not. It could always be extended to include more parameters, like the amount of work for the different type of boundary conditions that is used, inner or outer layer for the turbulence models, etc., but it would be very difficult for the user to give a correct estimation of all these parameters. It would not be so convenient either, especially not in an engineering environment where many different cases are calculated.

Tests has shown that it could be very difficult to correctly predict all parameters involved in a performance model. One aspect is the continuously changing software, where new models and numerical methods are introduced which could change the performance model in a way that could be difficult for the user to foresee. An example is changes in performance due to problems with cache or vector-registers.

Calculations with an early version of NSMB on the IBM SP2 gave a drop in performance of about 5 times for certain block-sizes. This was explained by the fact that the SP2-processors are 4-way associative, which means that each byte in memory could go to 4 different places in the cache. There are 1024 cache-lines in the SP2-processors, with a length of 64 byte ("thin" node), in each line, which means that
variables having a distance of 64 kbyte between them will try to go to the same places in the cache. This works fine if there are only 4 of these variables involved, but a fifth variable has to wait until one of the other four has been used. The problem comes in the next iteration, when the cacheline has to be filled again. Cachelines are read one at the time, which means that bytes surrounding the wanted byte are also placed in the same cacheline and could therefore be used in e.g. the next iteration. This advantage is not possible to use in the above example, because each variable has to be read again. Reading a variable from cache takes about 1 clock-cycle, while reading from memory takes at least 10 clock-cycles, i.e. reading from memory every time a variable is used costs 10 clock cycles, while reading from cache whenever possible takes $1 + 10/8 = 2.25$ clock cycles in average, which is almost 5 times faster ($10/8$ comes from reading to cache every 8th word).

Arrays in the vector-version of NSMB has the number of gridcells as first running index and variable-type as second running index, which gave cache-failure problems for blocks where the number of gridcells were a multiple of 1024, as there is more than 4 variables in many of the calculated expressions. This problem is not so big after flipping the array-indices in the RISC-version.

4 Dynamic Load Balance - Coupling NSMB with MB-Split

Many users of NSMB have experienced some initial difficulties using NSMB according to the solution process described above, with a load balancing pre-processor, it takes also some extra time and can give a lot of extra databases to keep track of. So that was one of the reasons that it was decided that MB-Split should be directly coupled to NSMB, so that parallel NSMB could be started directly, without running any pre-processor for load balancing. The only files necessary to run the parallel NSMB would be the database with the mesh and, possibly, solutions, the NSMB run-file and the LBCASE-file containing information about the parallel computer to run on. Some systems may demand additional files to be able to run parallel jobs, like a hostfile, etc., but that is not directly related to NSMB.

The LBCASE-file is not necessary in the actual implementation of the NSMB-MB-Split coupling, for parallel computers with homogeneous nodes and no consideration of communication bandwidth.

NSMB can instead be started with the number of nodes to run on as a command line argument: `>nsmb -p8`, to run on 8 nodes/processors.

However, the main reason for the dynamical load balancing implementation is to increase parallel efficiently by using actual measured computational time when per-
forming load balancing. Dynamical load balancing is also necessary when the computational case in use changes with time for different reasons. One reason could be that some of the nodes on the parallel machine are no longer available, which demands a redistribution of the blocks on the new number of nodes available. Proper check-pointing is of course necessary for this to work.

Another reason could be the use of different numerical algorithms for different parts of the flow, hence the computational work for different blocks becomes dependent on the flow-solution and therefore a new load balance needs to be performed within a certain interval.

### 4.1 Outline of program structure

Below is a flow chart that describes how the dynamic load balance works, when NSMB has been coupled to MB-Split.

**Initial work, done before load balancing loop**

All information about the flow case, including what is in the runfile (Mach-number, Re-number, turbulence-model, etc.) and what is in the database (boundary conditions, mesh-sizes, etc.), except arrays (grid-coordinates, solution vector, etc.) is read in the beginning of the run, described in the flow chart as “Read mesh-info from DB”.

NSMB calls MB-Split with information about block-sizes, boundary-conditions, etc., to create an initial mesh similar to the mesh saved in the database. The arrays are not saved due to memory-constraints. They are not necessary at this time, anyway. This mesh template is later used for the different load balancing cases to start from.

Among other things the “mesh” contains a list of the different load balancing cases tested, including information about how the meshes were split and their different run-times.

**Load balancing loop**

NSMB calls MB-Split to perform a load balance. The result is returned in the form of changed information about block-size, boundary conditions, and block-processor assignments. MB-Split reads information about the parallel computer from the “LBCASE”-file, if NSMB is not started with a command line argument demanding a certain number of homogeneous processors. Run time for the former run is saved in the list of load balancing cases, if it is not the first test in the load balancing loop.

Next, the arrays containing grid coordinates are read from the database, one by one, and sent to MB-Split. If that particular array belongs to a block that has been split in
this load balancing case, the array is split accordingly and returned as several arrays.
These arrays are then sent to their processors.

Next the solution arrays (if existing) are processed in the same way.

One timestep is performed, in order to get a timing for this case. If the load balancing loop is not finished, everything is deleted, except for the calculation time and the load balancing loop starts from the beginning again.

**Load balancing loop finished; last iteration**

Starts in the same way as usual, but this time the load balancing case with the lowest test-time is selected from the list of tested cases.

Now the calculation loop continues to the end of the calculation. It is also possible to stop before the end of the calculation and perform a new load balance, in case one of the nodes has disappeared, or if the load has changed due to a change in the flow, which in turn has changed numerical scheme, turbulence model, etc., for part of the mesh.

After the calculation has finished, MB-Split is called to merge the necessary arrays back to the original blocks, one by one, and save the solution after each merge. This way was chosen to reduce memory requirements on the single node that performs I/O.
4.2 Dynamic load balancing algorithm

Currently, two dynamic load balancing algorithms are implemented, which can easily be extended in the future. It is possible to select algorithm by giving a command line argument. One that uses the Greedy Load Balancing algorithm as a base and one that uses the GreedyXtra Load Balancing algorithm. They are both described in ref([11]) and in section 3.1.

The little $\varepsilon$ used in both the Greedy and GreedyXtra algorithms are varied from 0.03 to 0.11 in steps of 0.02. So the load balancing loop is iterated 5 times before the final calculation starts. The optimal imbalance, $1 + \varepsilon$, is case depended, but usually within the interval 1.03 to 1.11.
4.3 Implementation issues

NSMB was not at all suited for the structure necessary for the dynamic load balancing algorithm. The main problem was the fact that reading from the database and from the runfile was done in several places and not in the beginning of the program. This is convenient as different initializations are done in conjunction with reading from the database. For the dynamic load balance to work all information, except arrays, needs to be read in the beginning, before the load balancing loop, but most initializations are done after calling MB-Split and inside the load balancing loop.

Due to the I/O and initialization structure and some problems with the memory management, a radical rewrite of NSMB was necessary, involving program logics and data structures. But no part of the numerical calculations had to be changed.

4.3.1 Compiler issues

NSMB is implemented in standard Fortran 77 calls MB-Split, which is implemented in C++. The solution to this problem is compiler dependent and has been tested on the standard compilers for IBM RS6000 and SGIs using IRIX. Both the new compiler for 64-bit OS and the old with so called o32-object-files has been used on the SGI. Functions in MB-Split called by NSMB were implemented as pure C-functions, in order to have proper names in the object-files. C++ unfortunately uses names that are generated by the compiler and not so easy to predict. The linker needs to be given all necessary paths to all object-files and libraries that are used by both C++ and Fortran.

A C++-program can also contain parts that could not fully be compiled before linking, as the type can for example be unknown at compile time. This could be a problem for a linker that expects pure Fortran-files, so an extra program for post-compiling can become necessary.

Apart from the above, there are also the usual problems with naming of Fortran routines in object-files, which is done differently for different compilers. Some add an extra "." after the name and some use only capital letters, and so on.
5 Testcases and Results

5.1 AS28G

Paper II [33] analyzes turbulent calculations made on a full aircraft configuration, named AS28G. The fine mesh for this calculation consists of around 3.5 million grid-points and 62 blocks. A large number of blocks were due to the complicated geometry, especially around the nacelle and pylon.

![Residuals AS28G-Fine calculation, using B-L and LU-SGS or ADI on IBM SP2](image)

Figure 2: L2 residual based on the density, for the LU-SGS scheme and the Diagonal-ADI scheme. Both using Baldwin-Lomax and 32 SP2 processors.

Two different timestepping methods were used in the calculations, the Diagonal-ADI and the fully implicit LU-SGS scheme. It was not possible to get a converged solution using an explicit 5-stage Runge-Kutta scheme for the finest mesh and the calculation using ADI was stopped after 60 000 iterations due to the slow convergence rate on the fine mesh. Calculations using the LU-SGS scheme were very successful after initial problems with correct parameters for the Baldwin-Lomax turbulence model. The CFL-number was increased from 1 to $10^9$ in 700 iterations, giving a converged solution after 5000 iterations, starting from a freestream initialization. Figure 2 shows the difference in convergence rates between the ADI scheme and the LU-SGS scheme. A converged solution could be reached within about 2 days using either 32 processors on an IBM SP2, 64 processors on a Cray T3D, or a single processor on either a Fujitsu VX or a NEC SX4.

The solution is quite sensitive to parameter values in the turbulence model, giving a solution with lift ($C_L$) and drag ($C_D$) coefficients, that were about 2% for $C_L$ and 4% for $C_D$ from wind tunnel measurements. Pressure distribution on the wings is good close to the tip (see fig. 3.a), but not so good regarding shock positioning in the area around the engine (see fig. 3.b).
Parallel performance was not perfect on either of the two parallel machines. On the T3D it was mainly due to problems in estimating processor performance using MB-Split, while the problem for the SP2 was due to a slow network using public domain PVM combined with a large amount of communication for turbulent wall information.

The estimated imbalance for the load balancing that were performed with the GreedyX-tra algorithm was less than 2% for 32 SP2 processors (fig. 4), but the real value measured by NSMB showed a load balance that were 37% from optimum using ADI (fig. 5) and 10% for the LU-SGS scheme (fig. 6). The difference between theory and reality is due to difficulties in predicting the computational timings for the different blocks. A redistribution of the blocks using the real timings were made with MB-Split, together with a split of some of the most time-consuming blocks, which gave a load imbalance of 3% for ADI and 9% for LU-SGS. The total time was not decreased, due to the creation of more ghost cells when the extra blocks were split.

Figure 3: Cp distribution on the wing of AS28G, at different stations.

Figure 4: Theoretical loadbalance on 32 IBM SP2 processors, fine mesh AS28G.
Figure 5: Loadbalance based on real timings on 32 IBM SP2 processors, using ADI and Baldwin-Lomax, fine mesh AS28G.

Figure 6: Loadbalance based on real timings on 32 IBM SP2 processors, using LU-SGS and Baldwin-Lomax, fine mesh AS28G.
5.2 SAAB 105

A calculation for the turbulent flow in the air intake of the two-engine jet-trainer/light attack, SAAB 105, is presented in Paper III [34]. The mesh has 285 696 gridcells in 25 blocks and only the intake duct together with an upstream streamtube has been used in the calculations. Four different turbulence models and two different time-stepping schemes were compared, regarding both efficiency and accuracy. Comparisons with wind tunnel results were made as total pressure patterns and distortion indices at AIP, just in front of the engine, together with the values of some characteristic flow parameters. There were also a comparison made with solutions achieved using two other flow-solvers, both with a two-equation k-ε model.

The best NSMB solutions used either the algebraic Granville model or the two-equation model by Chien, both with a 5-stage Runge-Kutta time-stepping scheme. The Granville model was also used together with the implicit LU-SGS scheme, which gave a completely different solution. Figure 7.a shows the wind tunnel measurements, while figure 7.b is for the k-ε calculation, figure 8.a for the Granville calculation with Runge-Kutta and figure 8.b for Granville using LU-SGS, of the total pressure distribution at AIP. The low pressure region at six o’clock is due to flow separation just behind the lips, at the duct entrance, and the low pressure region at one o’clock is due to a secondary flow phenomena in the curved duct, bringing low energy flow from the boundary layer into the engine. All models could predict the basic flow characteristics, but not the correct level and exact position.

![Distortion Pattern SAAB 105 Intake](image)

Figure 7: Total pressure map with characteristic results.

The Granville model seemed to be quite sensitive to parameter values, like the Baldwin-Lomax model, even if the solution presented in figure 8.a predicts the flow

---

4 AIP = Aerodynamic Interface Plane
well. The mass flow, W1R, capture area ratio, CA, and pressure recovery, RE- COV, are exactly the same as for the measurements. Circumferential distortion, DC60, is however under-predicted, while the radial distortion index, IDRmax, is over-predicted. A small difference in separation point at the duct entrance could give large differences at AIP, which is one of the difficulties with these calculations.

Parallel performance is good for this case, especially for the laminar case, with a speedup of more than 57 on 64 processors and close to 30 on 32 processors. The Baldwin-Lomax calculation scaled worse, due to the increased amount of communication, for the turbulence wall information. The load balance achieved using the GreedyXtra algorithm in MB-Split, was almost perfect. Differences between theoretical estimate and measured values for the load balance are given figure 9. Figure 9.a shows an estimated imbalance of less than 1%, compared to the measured imbalance of 1.5% shown in figure 9.b, which is very good.

5.3 Hammershock calculation in Air Intake of JAS 39 Gripen

A dual time-stepping algorithm for time accurate calculations has been implemented in Paper IV [35] and tested on two testcases. Below is a description of one of the testcases, an inviscid hammershock calculation in the air intake of JAS 39 Gripen.

Hammershock is a phenomena occurring in air intakes when the engine is stalling, giving a sudden pressure rise at the engine entrance resulting in a shock moving in the upstream direction through the intake duct. The pressure rise over this shock could be as high as 3 times the static pressure in front of the shock, which is often the design load in an air intake duct. It could also be a large pressure rise at other parts of the aircraft immediately in front of the intake when the shock comes out
of the duct. This phenomena resembles the shocktube case described above and is mainly inviscid, but viscous effects like thick boundary layers, separation, etc., could of course have an impact in some cases.

### 5.3.1 Mesh and Flowcase Description

This calculation is time accurate and inviscid. Viscosity is not considered to have so much impact and is far to expensive for a time accurate calculation of a complex 3D geometry as the JAS 39 GRIPEN. There are 83 blocks in the mesh with a total of 1707938 cells. The complexity of the mesh and geometry, which could be seen in figure 10, gives very small cells in some regions, which will result in very small $\Delta t$ for the Runge-Kutta time-stepping.

Firstly, a steady-state solution is obtained using a constant static pressure as an outflow condition out of the intake duct, into the engine. Then, a time-accurate calculation is performed using a timedependent static pressure, at the intake outflow, simulating an engine stall.

### 5.3.2 Steady-state Calculation

The steady-state calculation was performed using the explicit Runge-Kutta scheme for time-stepping and the second order central difference scheme with artificial dissipation as spatial discretization. A three decade convergence was achieved in 14000 iterations (see figure 11), which took about 180 hours on 8 R8000 processors on a SGI Power Challenge.
Figure 10: Part of the grid, on the surface of the JAS 39 Gripen, including the air intake.
Figure 11: Residual of the steady-state calculation for the inviscid Hammershock case on 8 R8000 processors on a SGI Power Challenge.

Load balancing were achieved using the GreedyXtra Loadbalancing algorithm in MB-Split[11], which uses domain decomposition to get an even load on all nodes/processors. Theoretical load balance was 0.2% from optimum, but in practice, the load balance was 7.9% from optimum, which is considered as fairly good.

A plot of the Mach-number on the surface of the forebody of the JAS 39 GRIPEN could be seen in figure 12. No absolute levels of the Mach-number is given in this figure, but highest Mach-number is white and then the Mach-number is decreased with shorter wavelength, i.e. red is second fastest and blue is slowest. This calculation is inviscid which explains why the Mach number is not zero at the surface. This picture is not so illustrative in black/white, due to the mapping of the different colours.

5.3.3 Time-Accurate Hammershock Calculation

The steady-state solution is used as starting solution for the time accurate calculation and the outflow boundary condition at AIP\textsuperscript{5} is changed from a constant static pressure to a time dependent static pressure. The pressure pulse shown in figure 13 resembles the behavior of the static pressure when the engine stalls.

It takes approximately 15-20 ms for the pressure to increase enough for a hammershock to form and then it takes at least an additional 15 ms before the hammershock has traveled upstream, through the duct. Then the effect of the outgoing hammershock on the fuselage and the pressure decrease in the duct is studied for an extra

\textsuperscript{5}AIP = Aerodynamic Interface Plane
Figure 12: Mach number distribution on the forebody and intake of JAS 39 GRIPEN.

Figure 13: Time dependent static pressure at AIP for the time accurate calculation.
15 ms. So, the total simulated time is approximately 0.05 s, for this calculation it is 0.047 s.

With the CFL number set to 1.0, the explicit Runge-Kutta scheme gives a timestep of $\Delta t = 10^{-10}$ s, which means that a total simulated time of 0.047 s would take 470 million iterations. On a 8 R8000 processors SGI Power Challenge, this would demand a cpu time of 685 years.

After some initial tests with different $\Delta t$ for the dual time-stepping, $\Delta t = 10^{-5}$ s was considered to give an accurate enough result.

Figures 14 to 16 shows the difference in L2-norm for different variables and in different blocks, using $\Delta t = 5 \times 10^{-7}, 1 \times 10^{-6}, 1 \times 10^{-5}$ and $\times 10^{-5}$ s. The difference in fig 14 between solutions using $\Delta t = 5 \times 10^{-7}$ s and $\Delta t = 1 \times 10^{-6}$ s is very small($10^{-5}$), so $\Delta t = 1 \times 10^{-6}$ is approximated to be the exact solution. The error in fig 16 is about 10 times larger than for fig 15, $10^{-3}$ compared to $10^{-4}$, why $\Delta t = 1 \times 10^{-5}$ was chosen.

An automatic timestep control will be used in the future, but there are still some questions about the error calculation, especially around shocks and other discontinuities.

![Figure 14: Difference in L2-norm between solutions using a timestep of $1 \times 10^{-6}$ s and $5 \times 10^{-7}$ s.](image)

Each outer timestep demanded about 30 internal iterations in average, ranging from 10 to 200, to reach 2 decades of convergence, which means that 0.047 s of simulated time would demand 76 days of cpu time. Even if the cpu time is reduced by more than 3000 times, compared to the classical Runge-Kutta scheme, it still takes more than 2.5 months of cpu time.

Most of the time accurate calculations were performed on 1 processor of a Fujitsu

32
Figure 15: Difference in L2-norm between solutions using a timestep of $1 \times 10^{-6}$ s and $1 \times 10^{-5}$ s.

Figure 16: Difference in L2-norm between solutions using a timestep of $1 \times 10^{-6}$ s and $4 \times 10^{-5}$ s.
VX at PDC\textsuperscript{6}, KTH, which reduced the cpu-time to about two weeks for a simulated time of 0.047 s.

The speedup on 3 Fujitsu processors is very close to 3 for this case, which would mean a total cpu-time of 5 days, unfortunately the machine was very loaded so it was not practical to run on any more than 1 processor.

5.3.4 Results

Figure 17 shows the pressure as a function of time at different points in the intake duct. An overview picture of the point locations can be seen in fig 18. The slope of the pressure rise increases as the hammershock travels towards the duct entrance. Maximum relative increase in static pressure is achieved in point 4 ($l/L=0.28$), where the static pressure is close to 2.5 times higher than the steady state condition for this case.

![Figure 17: Pressure as a function of time in different points in the intake duct of JAS 39 GRIPEN.](image)

5.4 Dynamic Load Balance

Three different flow cases were used in the tests for the dynamic load balance:

- A-airfoil: A one block 2D-airfoil
- F5-wing: A one block 3D-wing
- AS28G: A 62-block mesh around a complete aircraft

Only the first test case, A-airfoil, is presented here. The other cases can be found in Paper V.

\textsuperscript{6}PDC = Center for Parallel Computers, KTH, Stockholm, Sweden
Figure 18: Locations of the points used in previous figure in the intake duct.

All those tests are more “proof of concept” than a real measure of how effective the dynamical load balance is. More tests need to be done when more complicated algorithms has been implemented, but it still gives a hint of the necessity of considering real calculation times when performing load balancing on parallel computers.

5.5 A-airfoil

A one block medium mesh with 32768 cells has been used for this 2D case. The turbulence model was a standard algebraic Baldwin-Lomax model and a 5 stage explicit Runge-Kutta scheme for timestepping, together with a Jameson central scheme.

Table 1 shows calculation times for different number of blocks and different values for $\varepsilon$. This case is not so large, so only 16 processors were used as speedup started to degrade when using more processors. The best value of $\varepsilon$ for most cases seems to be $\varepsilon = 0.05$, except for 8 processors were $\varepsilon = 0.07$ gave a slightly lower calculation time. Max difference is defined as $(\text{max-time} - \text{min-time})/\text{min-time}$.

6 Summary and Conclusion

A pre-processor tool, MB-Split, has successfully been developed in order to achieve good load balance on parallel computers for the parallelized CFD-solver NSMB. MB-Split was later coupled to NSMB to get dynamic load balance possibilities. An efficient scheme for time accurate calculations were implemented in NSMB, the so called dual time stepping. NSMB was also adopted and extended to be able to run internal flow cases.
A number of demanding flow cases has been run, in order to test how the above tools and implementations works for real-life industrial problems.

Turbulent flow around a full aircraft configuration, AS28G, and in an air intake duct was computed within a reasonable time, using the Navier-Stokes solver NSMB and the load balancing tool MB-Split on parallel computers. A correct, converged solution were achieved within 2 days for a full aircraft configuration with a mesh consisting of 3.5 million gridpoints.

The turbulence models are quite sensitive to parameter values for both cases tested, at least the algebraic models. The more advanced k-ε model could only be solved using the explicit Runge-Kutta scheme, with very slow convergence. More testing needs to be done for the k-ε model, before it could be considered reliable, especially for internal flow, but this initial test shows satisfying results, except for the convergence rate. Some of the problems encountered in the calculations could also be due to the mesh, and not entirely related to the turbulence models, which is very complicated in the AS28G case and maybe a little to coarse in the SAAB 105 case.

The k-ε model is much more computational costly than the algebraic models, but a large disadvantage with the algebraic models is the necessity to calculate global max and min values perpendicular to the wall. This increases the amount of communication considerably, which could be devastating for parallel computers with a slow network. The AS28G calculation spends a lot of time communicating the turbulence wall information, when used on the IBM SP2. It is possible to change the updating frequency of the turbulent wall information, but a block could have been split in the boundary layer, which makes it necessary to update this information every iteration, as the gradient could be strong over the block boundary. A more advanced turbulence model could in these cases be more effective and take less time per iteration.

<table>
<thead>
<tr>
<th>No of proc</th>
<th>2</th>
<th>4</th>
<th>8</th>
<th>16</th>
</tr>
</thead>
<tbody>
<tr>
<td>ε = 0.03</td>
<td>0.464</td>
<td>0.293</td>
<td>0.240</td>
<td>0.211</td>
</tr>
<tr>
<td>ε = 0.05</td>
<td><strong>0.364</strong></td>
<td><strong>0.214</strong></td>
<td>0.141</td>
<td><strong>0.106</strong></td>
</tr>
<tr>
<td>ε = 0.07</td>
<td>0.371</td>
<td>0.214</td>
<td><strong>0.141</strong></td>
<td>0.108</td>
</tr>
<tr>
<td>ε = 0.09</td>
<td>0.368</td>
<td>0.219</td>
<td>0.142</td>
<td>0.107</td>
</tr>
<tr>
<td>ε = 0.11</td>
<td>0.371</td>
<td>0.215</td>
<td>0.145</td>
<td><strong>0.106</strong></td>
</tr>
<tr>
<td>Max diff</td>
<td>27%</td>
<td>37%</td>
<td>70%</td>
<td>99%</td>
</tr>
</tbody>
</table>

Table 1: Timings for different values of ε on different number of processors of the SP2 using A-airfoil, medium mesh.
A comparison in calculation time for the dual time-stepping scheme and a classical explicit Runge-Kutta scheme for the realistic 3D, inviscid, JAS case, showed that the dual time-stepping scheme was more than 3300 times faster than the Runge-Kutta scheme. Unfortunately, 3300 times faster still means 76 days of cpu time on 8 R8000 processors on an old SGI Power Challenge to compute a whole hammershock cycle, but on a more modern parallel computer it would perhaps come down to a couple of weeks of CPU-time on a reasonable number of processors.

It is very important to have a good load balance in parallel computations, but the most difficult part of a load balancing algorithm is to have a correct performance model for a complex solver, such as NSMB. Problems like ghost cells, different boundary conditions, cache-failure, etc., makes it almost impossible to get a correct load balance for all cases using a pre-processor like MB-Split. It could give a reasonable good estimate for most cases, but a dynamic load balancing tool, that is coupled to the solver, is necessary in order to get the best possible load balance for an arbitrary case.

The dynamical load balance that were implemented by coupling MB-Split to NSMB measures actual calculation time by doing a single time step as a test after each load balancing suggestion. A number of load balancing suggestions can be produced by varying one or several parameters in the load balancing algorithm, or by using the time achieved in the previous test calculation.

Tests performed in this thesis and in Paper V, used a quite simple, static, load balancing algorithm as its base, and then achieved a dynamical behavior by varying one parameter and then picking the load balancing case which had the lowest calculation time. The difference between different values of this parameter were very large and not so easy to predict, which showed that a dynamical approach is sometimes necessary.

Another big advantage by this approach is that parallel NSMB is now much easier to run: all is done automatically by just starting NSMB with the number of processors as a command line argument.

One of the big problems in the implementation were the different programming languages used, C++ in MB-Split and Fortran 77 in NSMB. Another big problem was the structure of NSMB which was not at all suited for the dynamical load balancing algorithm used in this case.

Acknowledgment

Part of this work were performed within the ESPRIT - Europort1 project Parallel Aero, and the other part were performed within the NFFP(Nationellt Flygforskn
ingsProgram) project *Air Intake Design Optimized Using Computations*, project-number 2.252. The Swedish participation in *Parallel Aero* was financed by NUTEK.

The extensive use of the computers and all the help from the computer support group at the Center for Parallel Computers (PDC), KTH, Sweden is hereby gratefully acknowledged.

I am also grateful for all the help I have got from all the others involved in developing NSMB and involved in the *Parallel Aero* project. I am likewise grateful to Björn Ullbrand, SAAB Military Aircraft, for his help in the *Air Intake Design Optimized Using Computations* project.

I would like to thank my advisor Prof. Arthur Rizzi for sharing his knowledge in CFD with me. I would also like to thank the people that I work with at C2M2, NADA, KTH, for making it a great place to work at. A special thanks goes to Dr. Jesper Oppelstrup, NADA, for providing an excellent research environment plus all his help and support.

Developments of MB-Split and NSMB were also made in the PSCL-project “CFD-metoder för analys av dynamisk och stationär luftintagsströmning”, which was jointly financed by SAAB, KTH and NUTEK. Here I would also like to thank the other participants in this project, who all worked at SAAB.
References


Abstracts of Papers

Paper I: A Tool for Partitioning Structured Multiblock Meshes for Parallel Computational Mechanics
This paper discusses the MB-Split load balancing tool for Structured-grid Multi-Block Navier Stokes solvers running on parallel computers of MIMD-type.

Many load balancing algorithms and tools are available for CFD-solvers using unstructured meshes. Most load balancing algorithms for block-structured meshes only aim at distributing the initial blocks of the mesh over the processors, without splitting any blocks.

A more flexible load balancing technique uses domain decomposition, and splits blocks into smaller blocks. One of the problems with this approach is the complexity of updating block connectivity boundary conditions.

The three load balancing algorithms implemented, and some results for realistic test-cases are presented.

Paper II: Navier-Stokes Calculations on a Full Configuration Aircraft using Parallel Computers
This paper presents and analyze the turbulent flow over a full configuration (body, wing, nacelle and pylon) aircraft, the AS28G, with a block structured Navier-Stokes solver on two parallel computers: IBM SP2 and Cray T3D.

The results are compared with Windtunnel tests and parallel efficiency are compared with two modern vector computers: NEC SX4 and Fujitsu VX.

The experiment shows that a complete aircraft with a large Reynolds number ($\sim 10^7$) can be correctly calculated within 2 days using a modern vector or parallel computer, with an error in lift of around 2% and in drag of about 4%.

Parallel performance suffered from a bad load balance due to difficulties to predict the performances for the different blocks of the fine AS28G mesh. This makes the use of vector computers very competitive for these kind of large, complicated, 3D cases, with a single NEC SX4 processor corresponding to approximately 50 Cray T3D processors and 45 processors on the IBM SP2. The same figures for the Fujitsu VX were 34 and 32 respectively.

Paper III: Analysis of Static Distortion in the SAAB 105 Inlet, using Different Turbulence Models within the Navier-Stokes Solver, NSMB
An investigation has been made in order to determine whether the flow solver NSMB were capable of predicting internal flow in a curved air intake duct correctly and
efficiently. Four different turbulence models have been used: the algebraic Baldwin-Lomax\cite{15} and Granville\cite{16} models, the one-equation model by Spalart-Allmaras\cite{17}, the two-equation $k$-$\varepsilon$ model by Chien\cite{18}, and three different time-stepping schemes: an explicit 5-stage Runge-Kutta scheme, an ADI(Adverse Direction Implicit) scheme and an implicit scheme called LU-SGS. All calculations were made using central differences with second and fourth order artificial dissipation. The air intake of the two-engine SAAB 105 jet-trainer/light attack were used in the calculations, the results were then compared with Windtunnel measurements and calculations using two other flow solvers.

The algebraic Granville model using a 5-stage Runge-Kutta scheme gave the best results, together with the two-equation $k$-$\varepsilon$ model by Chien. However, calculations using the Granville model together with the LU-SGS scheme did not give so good results and the Granville model has shown to be quite sensitive to parameter values for flow cases with large separations, so a two-equation model is recommended for internal flow cases like this, with separation and strong secondary flows. It was very difficult to get a converged solution using the ADI-scheme, why there are no results presented with this scheme in the report. All solutions were able to predict the main structure of the flow at AIP(Aerodynamic Interface Plane), with a low pressure region at one o’clock, due to a secondary flow phenomena transporting boundary layer flow into the engine, and the other low pressure region at six o’clock, caused by separation just behind the lips, at the throat of the intake duct.

Performance were tested on four different platforms for all cases: Fujitsu VX, Cray J932, SGI Onyx2(Origin) and IBM SP2. Parallel performance were tested on 4, 8, 16, 32 and 64 processors on the IBM SP2 at PDC, KTH, Sweden. Load balancing made with the preprocessor tool MB-Split, gave very good load balance for all cases and the parallel speedup were good up to 64 processors for the laminar testcase, but not so good on 64 processors for the turbulent calculations. This was mainly due to the slow network when using the public domain version of the message passing library, PVM. Turbulent calculations are communicating much more data than laminar, due to the need to know max and min values of different flow properties along lines perpendicular to the wall, and in other mesh-blocks than the one containing the wall.

Paper IV: Hammershock Calculations in the Air Intake of JAS 39 GRIPEN, using Dual Timestepping

The dual timestepping scheme has been implemented into the flow solver NSMB, in order to make time-accurate calculations more efficient. This scheme has been tested for the hammershock phenomena occurring in air intakes when the engine stalls.

A comparison was made between the dual time stepping scheme and a more traditional explicit Runge-Kutta scheme on three test cases, two shock tube cases which resemble a hammershock and a full 3D, inviscid, hammershock calculation for the air
intake of JAS 39 GRIPEN. One of the shocktube cases uses an equidistant grid for inviscid calculation while the other shocktube case uses a stretched Navier-Stokes grid.

Dual timestepping is 6 times slower than the explicit Runge-Kutta scheme for the equidistant shocktube case, while it is 40 times faster for the shocktube case which uses a stretched grid. The reason for these results is that the timestep in the Runge-Kutta scheme for the equidistant grid is limited by accuracy and not by stability, which is not the case in the stretched grid were the timestep in the Runge-Kutta scheme is limited by stability, due to some very small cells. The implicit scheme used in the dual timestepping is A-stable and is only limited by accuracy, why the same timestep could be used for both shocktube cases.

The full 3D, inviscid, hammershock calculation was tested on 8 R8000 processors of a SGI Power Challenge, as well as on 1 processor of a Fujitsu VX. A complete calculation of 47 ms simulated time, would require approximately 685 years of cpu time on the SGI with the explicit Runge-Kutta scheme, while the dual timestepping scheme would finish in around 76 days, which is a speedup of more than 3300 times.

Paper V: Dynamic Load Balancing for CFD-solvers running on Parallel Computers

This paper describes a way to handle dynamic load balancing for CFD-calculations on parallel computers, using structured grids and why this is necessary for efficient calculations and ease of use, from a user’s point of view.

An industry-standard CFD-solver, NSMB, were coupled to a pre-processing tool, MB-Split, for load balancing, in order to achieve dynamic load balancing.

Real timings, together with details about the computer in use, are used to get the best load balancing as possible. Tests shows that this approach gives a much better results than just doing a static load balance, as it is very difficult to predict the performance for some flow cases on certain computers.