CONCURRENCY: PRACTICE AND EXPERIENCE, VOL. 4(5), 357–376 (AUGUST 1992)

Parallel computing in aerospace using multi-block grids. Part 1: Application to grid generation

J. HÄUSER AND H. WONG
European Space Research and Technology Center, P.O. Box 299, 2200 AG Noordwijk, The Netherlands

H. G. PAAP
Physikalisches Inst., NWII, University of Bayreuth, P.O. Box 3008, 8580 Bayreuth, Germany

W. GENTZSCH
Department of Mathematics, FH Regensburg 8400 Regensburg, Germany

SUMMARY

The present paper describes the implementation of multi-block codes, used to model complex 2-D geometries for applications in computational fluid dynamics on massively parallel architectures. The work starts with a brief description of ongoing and planned major aerospace projects and gives an estimate of the computing power needed. In order to provide this computational speed, one has to resort to massively parallel systems. In the first section the essential features of multi-block grids, along with the grid generation equations are discussed and it is shown that overlapping multi-block grids are inherently parallel by construction. Since the number of blocks is not fixed, but can be matched to a large extent to the number of available processors, there are no principal limitations of this parallelization approach, provided the ratio of computation time to communication time remains large enough, which leads to the discussion of problem scalability. The details of implementation on the Intel iPSC/2 of a general 2-D multi-block mesh-generation code are outlined in sections 2 and 3, together with the listings of the major communication function (Section 4). In section 5 the results for this code are presented, clearly demonstrating that the multi-block concept is a viable tool for massively parallel computers, which can be applied to virtually all problems in science and engineering where computational meshes are used. In section 5.2 an outlook on the parallelization of more complex problems is given, and estimates for speed-up and efficiency, based on the present experiences, are provided. It turns out that, as long as computation dominates communication time, which is usually the case for complex aerospace applications, parallelization will be the tool to provide the additional orders of magnitude of computing power needed to routinely design and analyse future aircraft as well as spacecraft, in particular at high Mach numbers, when chemical reactions become important.

1. PARALLELIZATION OF COMPUTER CODES FOR AEROSPACE APPLICATIONS

For the next two decades the operation of several space vehicles for hypersonic missions is planned. All these projects have in common that the vehicles are entering or flying in an atmosphere with high speed, resulting in high thermal loads and stresses. The high velocity at which the vehicles negotiate the atmosphere makes it necessary to consider chemical reactions, e.g. dissociation of molecules, and for temperatures larger than 10,000°C ionization occurs. In any case, the flow is compressible and a bow shock develops in front of the vehicle, leading to a jump of temperature, T, velocity components u, v, w (Cartesian co-ordinate system) and pressure p, as well as density ρ. If the temperature is high enough for chemical reactions to occur, the conservation for each

1040–3108/92/050357–20$15.00

Received 20 November 1990
Revised 21 November 1991

chemical species has to be considered (the simplest model of air involves N_2, O_2, N, O, and NO). Depending on the ratio of characteristic flow time and thermal relaxation time, a decoupling of the vibrational energy modes can occur, resulting in additional equations for the thermal energies of the molecules (i.e. for N_2, O_2 and NO if the model mentioned above is used). For the design of a vehicle the prediction of thermal loads and thermal stresses is needed as well as the aerodynamic performance (i.e. pressure load, drag, stability behavior, etc.).

Although numerous wind-tunnel tests are performed, there are fundamental limitations for these facilities[1], especially in the high-temperature range, apart from the question of cost-effectiveness and versatility or development speed. On the other hand, computational fluid dynamics (CFD) has matured during the last decade and now allows the computation of flow fields past complete configurations. Apart from the question of modeling and code validation, computation time is a severe problem. Firstly, there are very different length scales in the problem, in principle ranging from the turbulent scale where dissipation takes place to the characteristic length of the body. Secondly, there are very different time scales ranging from the time for a chemical reaction to the flow velocity in the boundary layer (BL), if viscosity is accounted for. In order to resolve all the relevant physical phenomena, an enormous number of grid points is needed, even when turbulence is modeled algebraically. It is not unrealistic to assume that 10 million grid points are needed for a 3-D configuration. From the above discussion we know that in a high-temperature environment additional equations have to be incorporated, leading for the simple atmosphere model to a set of 12 coupled non-linear partial differential equations (PDE)[2] in 3-D. It is known that for an implicit solution technique some 10^{-4} s per grid point per iteration are needed based on a computation power of 100 MFlops (sustained, that is, at 1 GFlops peak). Assuming 300 iterations (optimistic) for convergence and a grid of 10 million grid points amounts to a total computation time of 3×10^5 s or some 90 h. If a more sophisticated turbulence model is included, the computation time will be substantially higher.

To give the reader an impression about the type of simulation being performed, Plates 1 to 4 may serve as an example. Plate 1 depicts a structured grid for the Hermes space plane, similar to but smaller than the Space Shuttle. Two grid planes are shown, one in the stream-wise (along the centerline) direction, the other one representing a cross-section with the winglets. The grid is orthogonal at the surface of the vehicle. Grid points are clustered in the vicinity of the body and nose part. Grid size is 143 (circumferential), 33 (radial, for Euler calculations) and 62 stream-wise, i.e. some 293.000 points are used.

For parallelization this solution domain has to be subdivided into blocks which are mapped onto boxes and the connectivity of the boxes has to be determined. Hence, the main parallelization effort is the construction of these blocks. The communication between blocks is straightforward, since information is exchanged through neighboring faces. It is interesting to note that no additional communication structures in comparison to a multi-block stucture running on a sequential machine have to be developed. The reason is that a multi-block structure necessitates communication among blocks, independent of the computer architecture. Plate 2 depicts the surface grid used. The blocks on the surface serve as the boundary for the volume grid. Plate 3 shows a grid around a cone where two different physical parameters were used to capture the oblique shock and the boundary layer. In order to achieve load balancing, a blocking methodology to ensure

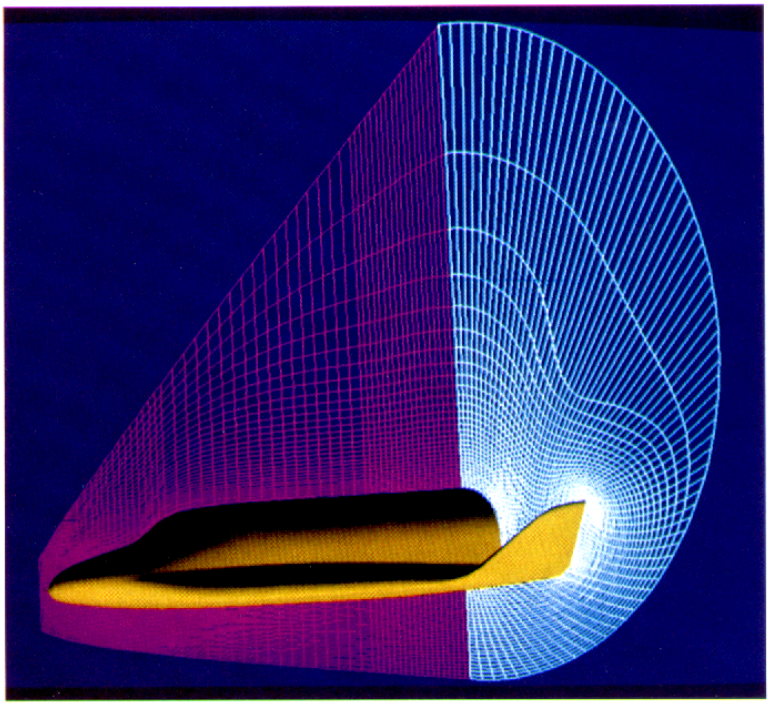

Plate 1: *Structured grid used for typical inviscid Hermes calculations. Grid size is about 300,000 points; 143 in circumferential direction, 33 in radial and 62 points in streamwise direction.*

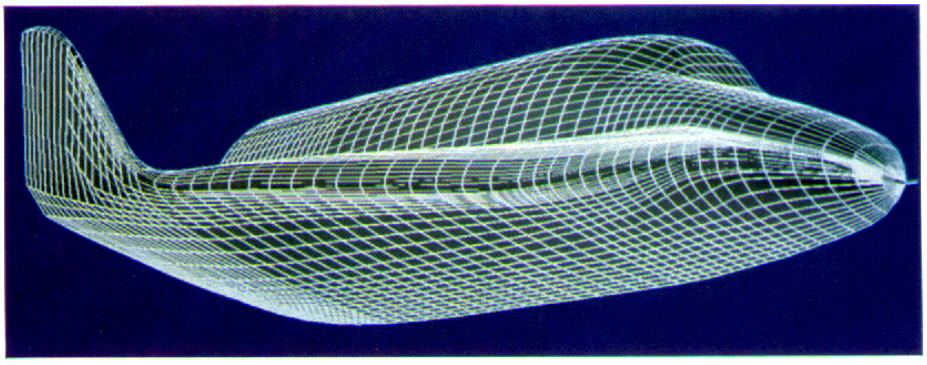

Plate 2: *The figure depicts the Hermes surface grid. Surface grid generation is a major issue, since the surface grid critically determines the quality of the volume grid.*

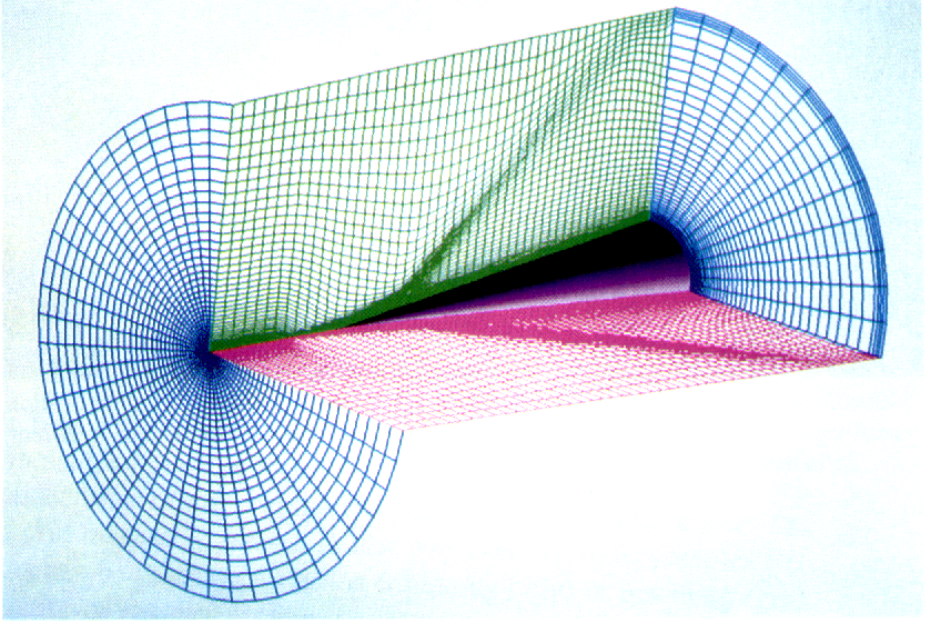

Plate 3: *Solution adaptive grids are of high importance to improve the solution accuracy and to limit the number of grid points needed. Efficiency of parallel algorithms should not be affected by grid point movement.*

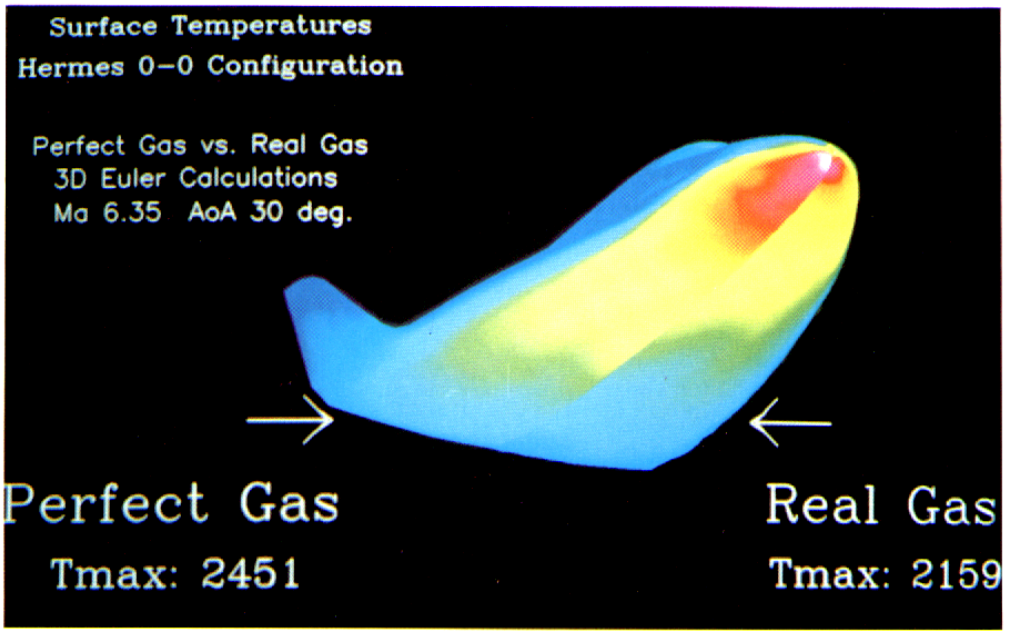

Plate 4: *Comparison of inviscid Hermes solutions obtained from perfect gas and real gas (accounting for additional degrees of freedom such as vibration, dissociation etc.).*

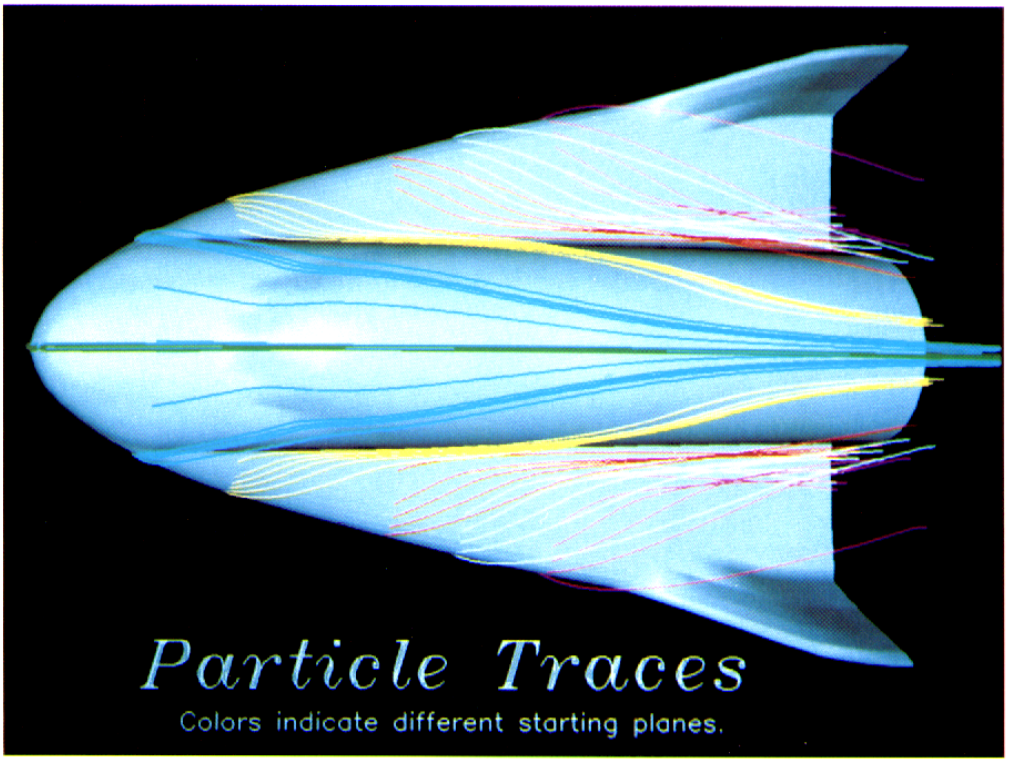

Plate 5: *Visualization of the flow field by introducing colored particles that move along stream lines, a technique used in wind tunnels (colored smoke). It gives the design engineer special information about, for example, vortices that might hit the structure.*

that blocks with approximately the same number of grid points is used, provided that this is the correct measure to determine computational load. Plate 4 shows a comparison of the temperature fields, using the grid of Plate 1, of the inviscid Euler equations for a perfect gas (left part) and the chemical equilibrium. Since chemistry intoduces additional degrees of freedom, the transitional temperature is substantially lowered. However, to determine the heat flux in the vehicle, Navier–Stokes calculations are mandatory. Plate 5 shows the utility of advanced visualization of 3-D results.

Although there has been a dramatic growth in computer power in the past, a limit, caused by the speed of light, of some 1 GFlops per uniprocessor can be foreseen. If a sophisticated CFD code is to be used as a design tool, the computation time should not exceed some 15 min, which amounts to the use of 360 of these 1 GFlops peak rate uniprocessors in parallel. The question then is, provided the hardware as well as the parallel compiler along with the corresponding operating system is available, how can aerospace computer codes be parallelized to make efficient use of a large number of very powerful computational nodes. In any case, it can be seen that using only a handful of parallel nodes (computers) will not be sufficient for this type of problem; thus one has to resort to massively parallel systems where each node has its own private memory to avoid access conflicts which arise if a shared memory is used.

2. MESH GENERATION FOR AEROSPACE APPLICATIONS

2.1. Multi-block meshes

Boundary fitted grids (BFG) (Plate 1) are now in wide use since they allow the distribution of grid points with alignment to the geometry, which may be irregular. BFGs are structured, and at each grid point corresponding co-ordinate directions exist. A BFG can easily be visualized by considering the electric field lines and equipotential lines (which intersect orthogonally) between the (curved) plates of a condensator. From electrostatics it is known that this line distribution is generated by Laplace's equation. In addition, the grid line density can be changed by introducing charged particles in the space between the plates, which leads to the solution of the Poisson equation. Following the same principle, a general BFG is simply constructed by solving a Poisson equation for each co-ordinate direction. This means that the irregular solution domain (SD) is mapped onto a rectangle (2-D) or a box (3-D) in the computational domain (CD). This approach works well as long as the shape for the SD is not too complex, because then the grid can become too distorted. The next step then is to introduce branch cuts to map a multiply connected SD onto a single rectangle or box in the CD. Although this can be done, it is easy to show that certain grid line configurations cannot be obtained, e.g. if the grid line distribution is to resemble the stream line pattern of a flow past a cylinder. (For a more complete discussion see Reference 3.) A way out of this dilemma lies in the use of multi-block grids, first introduced in Reference 4. The SD is covered by a set of overlapping charts (or blocks) where each single block is mapped onto a rectangle or box in the CD. The whole SD then is mapped onto a set of connected blocks in the CD, which retains the connectivity of the original physical domain and thus allows the generation of an arbitrary grid line configuration. In that way even the most complex SD can be handled. The blocks can be patching (grid line continuity only) or matching (grid line continuity and continuity of tangent vectors, i.e. slope continuity). Patching grids

are easier to construct, but from the standpoint of flow solution, matching grids are to be preferred, because of the implementation of the numerical scheme that demands access to neighboring points. Therefore this approach has been implemented in the general mgp-3d code[5], where further details can be found. In this work an overlap of one edge (2-D) or one face (3-D) with the neighboring block is used.

Although higher overlaps can be constructed, such a scheme can lead to substantially increased storage requirements, since all the overlaps have to be stored twice. Figure 1 shows the principle of overlap.

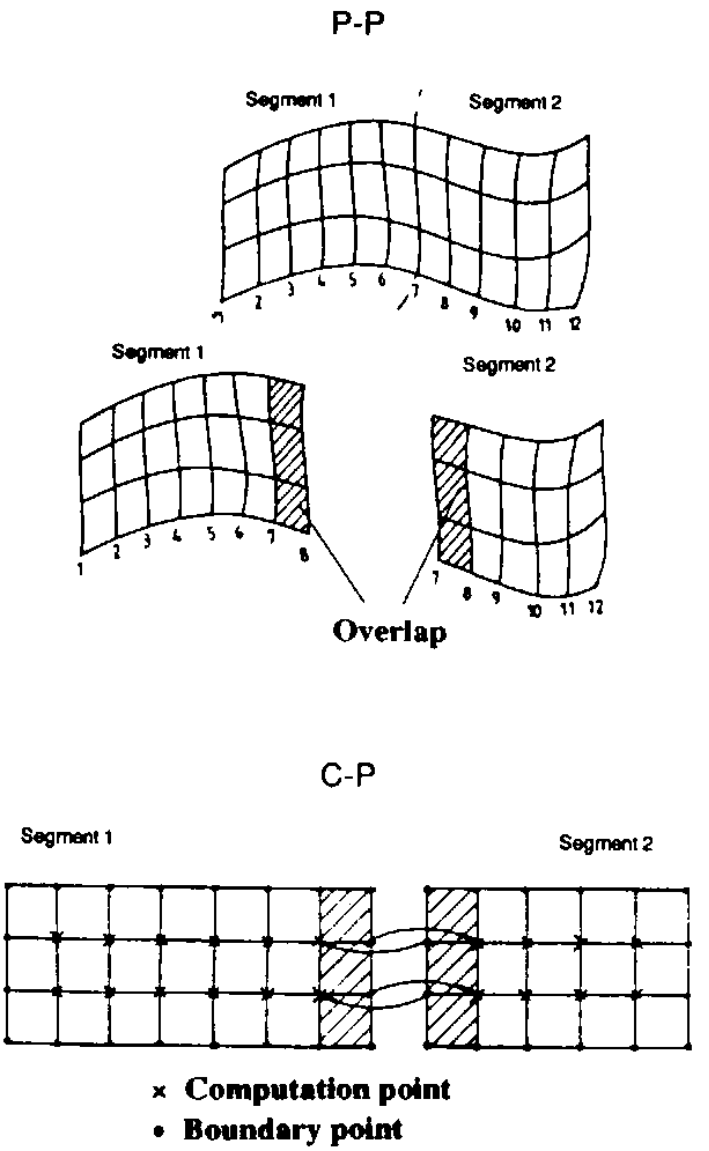

Figure 1. Overlap between neighboring blocks in 2-D. Here exactly one row or column overlaps, ensuring slope continuity of grid lines when passing block boundaries, that is, these boundaries are not visible when the final grid is depicted. Therefore an additional set of rows and columns is stored in each block, forming the boundary of this block. These points, however, are interior points in the respective neighboring blocks

It should be pointed out that the generation of multi-block grids for complex 3-D configurations is not a trivial task, since the blocking topology can become quite complicated, and each block has its own co-ordinate system where the orientation with respect to neighboring blocks has to be determined[5].

2.2. Mesh-generation equations

The following general co-ordinate transformation from the Cartesian co-ordinate system, denoted by co-ordinates (x, y, z), to the CD plane, denoted by co-ordinates (ξ, η, ζ), is considered. The one-to-one transformation is given by (except for a finite (small) number of singularities):

$$\begin{aligned} x &= x(\xi,\eta,\zeta); \qquad \xi = \xi(x,y,z) \\ y &= y(\xi,\eta,\zeta); \qquad \eta = \eta(x,y,z) \\ z &= z(\xi,\eta,\zeta); \qquad \zeta = \zeta(x,y,z) \end{aligned} \tag{1}$$

Since there is a one-to-one correspondence between grid points of the SD and CD, indices i, j and k can be used to indicate the grid point position. The grid in the CD is assumed to be uniform with grid spacings $\Delta\xi = \Delta\eta = \Delta\zeta = 1$. As mentioned in Section 2.1, a set of Poisson equations is used to determine the positions of (ξ_i, η_j, ζ_k) in the SD as functions of x, y, z. In addition, proper boundary conditions (BC) have to be specified. Normally, Dirichlet BCs are used, prescribing the points on the surface, but for adaptation purposes von Neumann BCs are sometimes used, allowing the points to move on the surface, in order to produce an orthogonal grid in the first layer of grid points off the surface. The Poisson equations for the grid generation read:

$$\begin{aligned} \xi_{xx} + \xi_{yy} + \xi_{zz} &= P \\ \eta_{xx} + \eta_{yy} + \eta_{zz} &= Q \\ \zeta_{xx} + \zeta_{yy} + \zeta_{zz} &= R \end{aligned} \tag{2}$$

where P, Q, R are so-called control functions that depend on ξ, η and ζ. However, this set of equations is not solved on the complex SD; instead it is transformed (e.g. Reference 3) to the CD. The elliptic type of equations (2) is not altered, but since ξ, η and ζ are co-ordinates themselves, the equations become non-linear. Using the more compact notation

$$\xi^1 := \xi \subset ; \quad \xi^2 := \eta ; \quad \xi^3 := \zeta ; x^1 := x ; \; x^2 := y ; \; x^3 := z \tag{3}$$

Equations (2) can be written in the form

$$\nabla^2 \xi^l = g^{ll} P_l(\xi^1,\xi^2,\xi^3); \quad l = 1,2,3 \tag{4}$$

where $g^{11}P_1 = P; g^{22}P_2 = Q; g^{33}P_3 = R$ was used in equations (2). There is no summation over l on the RHS of equations (1). The factor g^{ll} represents the diagonal terms of the contravariant components of the metric tensor g, with $g^{ll} = \mathbf{e}^l \bullet \mathbf{e}^l$, where $\mathbf{e}^l$ are the contravariant base vectors[3]. Interchanging dependent and independent variables, the transformed Poisson equations have the form

$$g^{jk}\, x^l_{\xi^j \xi^k} = g^{mm} P_m x^l_m; \quad l = 1,2,3 \tag{5}$$

The summation convention of Einstein is used in equations (5); i.e. indices occurring twice are summed over. For 2-D, using the original functions P and Q and co-ordinates x, y and ξ,η, one obtains:

$$\begin{aligned} g_{22}\, x_{\xi\xi} - 2g_{12}\, x_{\xi\eta} + g_{11}\, x_{\eta\eta} + g(x_\xi P + x_\eta Q) = 0 \\ g_{22}\, y_{\xi\xi} - 2g_{12}\, y_{\xi\eta} + g_{11}\, y_{\eta\eta} + g(y_\xi P + y_\eta Q) = 0 \end{aligned} \tag{6}$$

where

$$g_{11} = x_\xi^2 + y_\xi^2;\ g_{12} = g_{21} = x_\xi x_\eta + y_\xi y_\eta;\ g_{22} = x_\eta^2 + y_\eta^2 \tag{7}$$

$$g_{11} = g^{22} g;\ g_{12} = g_{21} = -g g^{12} = -g g^{21};\ g_{22} = g g^{11};\ g = (x_\xi y_\eta - x_\eta y_\xi)^2$$

was used. The numerical solution of equations (6) along with specified control functions P, Q as well as proper BCs is straightforward, and a large number of schemes is available. Here a very simple approach is taken, namely the solution by successive-over-relaxation (SOR).

The second derivatives are described in the form

$$\begin{aligned} (x_{\xi\xi})_{ij} &= x_{i+1j} - 2x_{ij} + x_{i-1j} \\ (x_{\eta\eta})_{ij} &= x_{ij+1} - 2x_{ij} + x_{ij-1} \\ (x_{\xi\eta})_{ij} &= 1/4(x_{i+1j+1} - x_{i-1j+1} - x_{i+1j-1} + x_{i-1j-1}) \end{aligned} \tag{8}$$

Solving the first of equations (6) for x_{ij} yields the following scheme:

$$\begin{aligned} x_{ij} = 1/2(\alpha + \zeta)^{-}1 \{ & \alpha_{ij}(x_{i+1j} - x_{i-1j}) \\ & - \beta_{ij}(x_{i+1j+1} - x_{i-1j+1} - x_{i+1j-1} + x_{i-1j-1}) \\ & + \gamma_{ij}(x_{ij+1} - x_{ij-1}) \\ & + 1/2\, J_{ij}^2\, P_{ij}(x_{i+1j} - x_{i-1j}) \\ & + 1/2\, J_{ij}^2\, Q_{ij}(x_{ij+1} - x_{ij-1}) \} \end{aligned} \tag{9}$$

where the notation $J^2 = g$; $\alpha = g_{22}$, $2\beta = g_{12}$, and $\gamma = g_{11}$ was used. Over-relaxation is achieved by computing the new values from

$$x_{new} = x_{old} + \omega(x - x_{old}); \quad 1 \leq \omega < 2 \tag{10}$$

Figure 2 shows a 32-block grid where slope continuity of the grid lines makes the block boundaries invisible. This is a complex grid since a C-type grid is embedded in an H-type grid, leading to a singularity and producing volumes of non-rectangular shape. This, however, does not pose a problem as long as the volume of an element is different from zero. In Figure 3 a view of the block structure is presented.

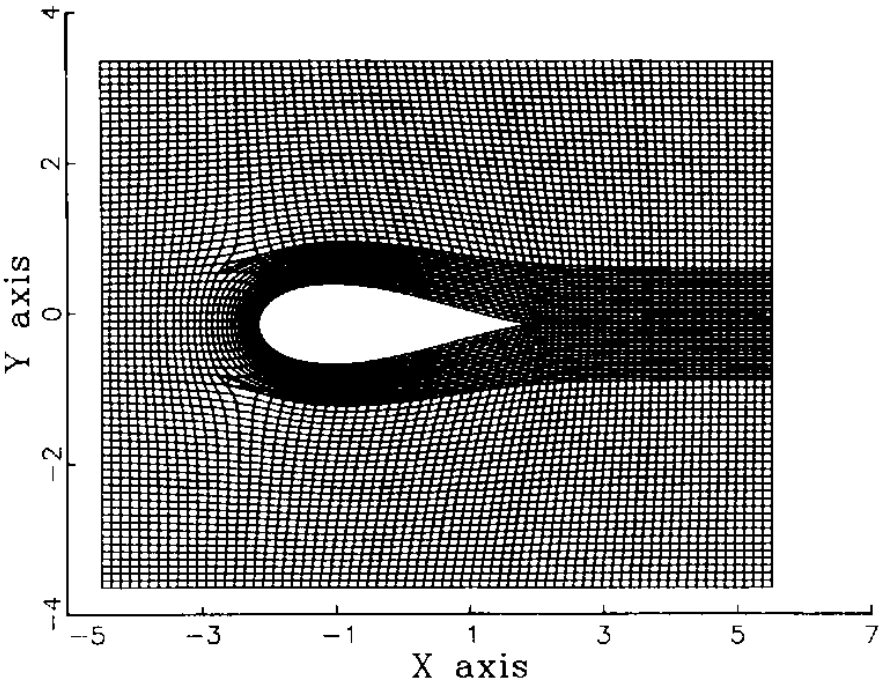

Figure 2. Multi-block grid around an airfoil comprising 32 blocks. Because of the slope continuity, block boundaries are not visible. Since a C-type grid is embedded in an H-type grid, singularities arise, which do not pose any numerical problem as long as the volume of the triangular elements remains finite

The solution process for a multi-block grid is achieved by updating the boundaries of each block, i.e. by receiving the proper data from neighboring blocks and sending overlapping data to neighboring blocks (see Figure 4). Then one iteration step is performed using these boundary data, iterating all interior points. In the next step, then, the newly iterated points which are part of the overlap are used to update the boundary points of neighboring blocks and the whole cycle starts again, until a certain number of iterations has been performed or until the change in the solution of two successive iterations is smaller than a specified bound.

3. PARALLELIZATION FOR MULTI-BLOCK MESHES

3.1. Parallelization by domain decomposition

From Section 2 it should be clear how parallelization of a multi-block code can be achieved in a straightforward way. If there are more blocks than processors, they are

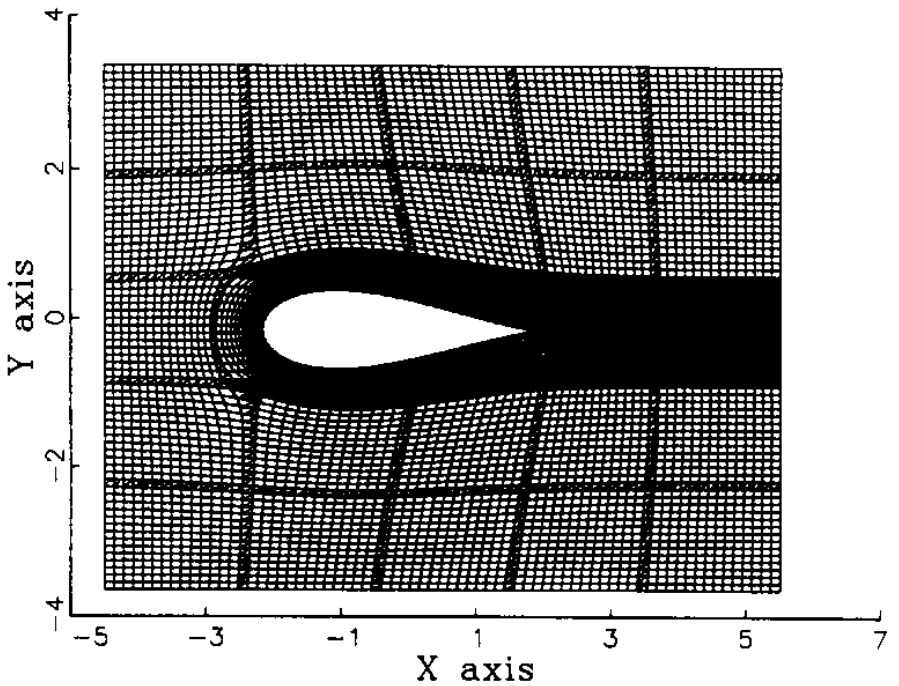

Figure 3. View of the block structure of the airfoil multi-block grid. The question of the shape of the domain in the CD is not meaningful. In the CD there is only a set of 32 connected rectangles

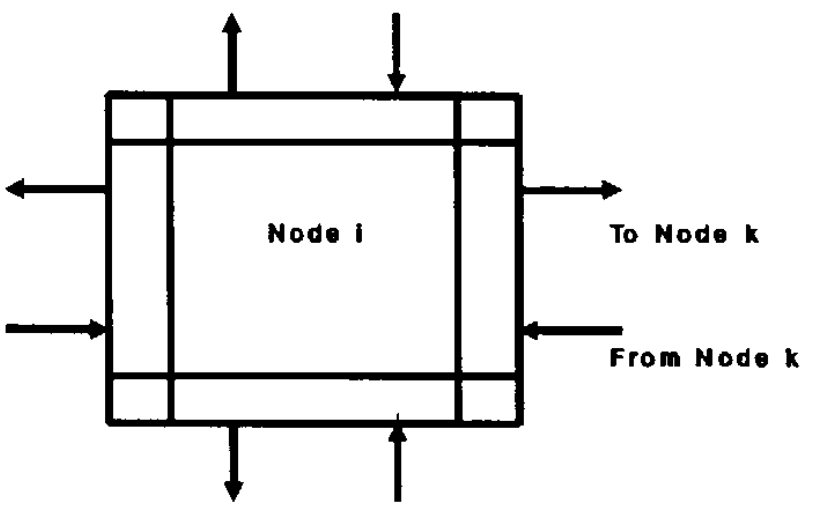

Figure 4. In 2-D each block sends edge information out to up to 4 neighbors and has to receive edge information from up to 4 neighbors. Send and receive have to be synchronized only in a loose way; that is, each node can continue its computation upon receiving the updating information, without waiting for other nodes. Hence, the receive command must be a blocking one, because the program execution has to be interrupted until the data have been received

distributed to the processors in such a way that the total number of grid points is approximately the same. That means it may be necessary to start more than one process per processor, which is possible with the Unix Operating System (OS). It is possible with the present grid generator to generate almost any desired number of blocks. However,

there is a minimum number of blocks dictated by the complexity of the configuration. So far the question of load balancing has not been addressed, which can be satisfied by subdivision of existing blocks, which leads to the next topic: scalability. This means that there must be a certain problem size on each single processor in order to have a high ratio of computation time against communication (or transfer) time. Obviously it is not useful to distribute a fixed-size problem to a very large number of processors, ending up with only a few grid points per processor. Here the advantage of the present multi-block approach is striking. The problem can be run on a coarser grid on a small number of processors such that a high ratio of computation against communication time is obtained. The scaled problem, that is more grid points and more blocks, can then be distributed on a higher-dimensional hypercube so that the number of grid points on a single processor remains about the same. This approach assumes that a large enough number of processors is available. For a Teraflop machine , for example, most likely several thousand nodes will be needed. Hence, for a grid of 10 million points, a block comprises several thousand internal points, generating a compute-dominated problem. Thus, the speed-up observed on the small grid will be nearly the same as on the large one. This is clear since the ratio of computational time and communication time, τ_{CT}, remains almost unchanged. Therefore the values measured on a low-dimensional hypercube can be used to predict performance on a higher-dimensional cube (and similar ones in Reference 6), if the problem is scaled accordingly. Although the present investigations are restricted to a 32-processor version of the hypercube, results will most likely remain valid for larger systems. In the following it will be outlined why it is important to deal with the parallelization of the multi-block grid code.

There have been many papers—for example, the excellent publication of Gustafson *et al.*[7], where a large number of processors was already used. Despite these successful calculations, there are two additional points to ponder, which so far have not been demonstrated, but are essential in the parallelization of real production CFD codes. Firstly, the parallelization must not depend on domains that are of regular shape, e.g. a rectangle or a box or any type of domain, which has to be mapped on a simply connected area. These cases are not relevant for practical purposes. Instead, speed-up and efficiency have to be demonstrated for multi-block grids that form a completely irregular pattern in the CD. This is mandatory, as was outlined in Section 2, to obtain the most general grid line distribution. Secondly, the parallelization strategy must not depend on a nearest-neighbor basis for the blocks, e.g. the arrangement of blocks in a mesh. In the multi-block case, blocks are simply numbered by the user and this block number is used to map the block to the corresponding processor number where, for the sake of simplicity, it is assumed that the number of blocks and processors are the same. However, blocks could be numbered using some type of gray code in order to minimize the total communication distance between the various blocks. So far no attempt to do this has been made. Regarding equation (9) it is clear that the computation work per grid point is not very large, since the numerics are very simple. It has been estimated that the computational work per grid point for the implicit solution of the system of 11 non-linear equations (2-D) for hypersonic flow, incorporating thermo-chemical non-equilibrium[2, 8], is a factor of 100–200 higher than for the solution of equation 9, while the communication overhead increases roughly by a factor of 10. If one goes to three space dimensions, τ_{CT} increases even more. Thus if a reasonable speed-up can be obtained for the simple grid generation equations, a much better speed-up can be

expected for the flow equations solver. Since the grid generation code uses exactly the same multi-block structure as the much more complicated flow solver, it is advisable to investigate the grid generation code first, since this definitely provides a lower bound for the speed-up and the efficiency of the parallelized code. With the experience of this code it will be relatively straightforward to parallelize the aerothermodynamics flow code[9].

3.2. Amdahl's law revisited

Amdahl's law, presented in 1967, equation (11), represents a type of barrier for parallel applications with respect to achievable speed-up, if a fraction of the code has to be serial. In general, Amdahl's law severely limits the achievable speed-up but is not a limiting factor for multi-block grids, as will be shown below. Next, we present this law in its original form. Let s and p be the normalized serial and parallel fractions of the code as measured on a uniprocessor. If the code is run on an n-processor massively parallel system the computation time is $s + p/n$. The speed-up S therefore is

$$S(n) := \frac{\text{uniprocessor time}}{\text{multiprocessor time}} = \frac{s + p}{s + p/n} = \frac{1}{s + p/n} \tag{11}$$

For $n \to \inf, S = 1/s$. If s is 1However, the law in this form does not account for the fact that s and p can be functions of n. It simply refers to the fixed-sized problem and therefore necessarily predicts a severe limitation in speed-up. The fixed-sized problem constraint is avoided if the question is asked, as in Reference 7: how long will it take to run a parallel code on a uniprocessor? Let s' and p' be the corresponding values on the multiprocessor system. The speed-up then is given by S, which is now a function of the number of processors n and the problem size P (e.g. number of grid points):

$$S(n,P) = \frac{s'(n,P) + np'(n,P)}{s'(n,P) + p'(n,P)} = n - (n - 1)s'(n,P) \tag{12}$$

where s'+p'=1 (normalized) was used.

One should keep in mind that s' and p' depend on the overall problem size, while in equation (11) s and p are constants. It is therefore clear that s and s' are different quantities. If n equals 1024 and s' is 1%, a speed-up of 1014 is obtained, which actually has been achieved in Reference 7. However, the question can be put in quite a different context. For the present multi-block formulation, Amdahl's law is not applicable at all. Regarding the construction of the multi-block grid, one has the situation that a completely parallel design has already been achieved; but, because of the lack of a parallel computer in 1984 when this work was started, the parallel code had to be executed on a serial machine. This was simply achieved by processing all blocks sequentially, but after each iteration the necessary updates had to be made, that is, the communication task on the serial machine was exactly the same as on the parallel system. Although in 1984 there was no intention to write a parallel application, the structure of the multi-block code is already parallel, so that the principal design remains unchanged.

A host and a node program were written for the parallel application. The only limitation therefore comes from the ratio τ_{CT}. However, it is interesting to note that the ratio of

communication time per byte to the computation time per floating point operation is relatively constant for various generations of computers, e.g. the N-cube as described in Reference 7 has a ratio of 1 MByte/s to 0.3 MFlops. The newer Intel iPSC/2 has a ratio of 2.5 MByte/s to 0.8 MFlops (sustained, if the SX option is used). This ratio is approximately the same for the intracluster-bus of Suprenum[10] and even lower when the Suprenum bus is doing the data transfer. Even if one anticipates a node with 100 MFlops sustained rate (peak rate some 1 GFlops), a bus rate of 250 MByte/s is optimistic, indicating that τ_{CT} remains relatively constant and at least will not be decreasing. In Section 4.3 some estimates are presented. This shows that speed-ups from slower systems can be transferred to the forthcoming massively parallel systems since the value of τ_{CT} remains roughly the same.

4. INSTALLATION OF THE PARALLEL MULTI-BLOCK MESH GENERATION PROGRAM (PMGP) ON THE INTEL IPSC/2

4.1. Technical aspects of the iPSC/2

A detailed discussion of the technical aspects of the iPSC/2 is given in Reference 11. Here the presentation is limited to explain the technicalities that are of importance for the performance of the multi-block algorithm. The iPSC/2 is a private memory and massively parallel system, whose hardware components are microprocessor industry standard. The CPU is an Intel 80386 processor with either an Intel 80387 arithmetic coprocessor (32-bit : 0.25 MFlops, 64-bit : 0.210 MFlops) or a Weitek 1167 coprocessor (64-bit : 0.65 MFlops). The floating-point performance can be enhanced by adding the iLBXII interface with a vector board (32-bit : peak 20 MFlops ; 64-bit : peak 6 MFlops). The topology of the iPSC/2 is a hypercube; that is, if $n = 2^d$ is the number of processors, each processor has d nearest neighbors. Therefore a maximum of d hops (node-to-node communication) is needed for a message transfer from the first to the last processor. Each node has seven bidirectional channels allowing a maximum of 128 processors to be connected. The direct connect routing module (DCM) is responsible for the link of any two nodes in the hypercube network where intermediate nodes are used. After the message has arrived on the target node, the communication channel is deblocked. DCM is nearly independent of the CPU activity of the node. The transfer time, even for short packets (100 bytes) is independent of the number of intermediate nodes. A value of 274 μs is given in Reference 12. Each node has its own OS, called NX/2. The whole system is directed by a host which runs the system resource manager (SRM). The SRM distributes NX/2 to all nodes when the cube is started. NX/2 is responsible for the message passing between nodes, the multi-tasking (e.g. if there is more than one block per processor), and the memory management. Node memory can vary from 1 to 16 MBytes.

The maximum bandwidth, that is the highest transfer rate between nodes, is 2.8 MBytes/s. For comparison, the iPSC/1 had 512 KByte per node, 80286, 8MHz, CPU and a 0.03 MFlops arithmetic coprocessor. Each node has eight communication channels that can be attached to a SCSI-bus (small computer system interface) to perform concurrent I/O at a rate of 2.8 MBytes/s. As host, a 80386/80387 PC can be used for housing all the compilers. If a deadlock occurs, the cube has to be reset from the host, which is somewhat inconvenient if one works on a terminal that is distant from the host. The cube can be partitioned into subcubes to make it a multi-user system.

4.2. Algorithmic structure of PMGP

The parallel algorithm comprises two parts. The first part runs on the host computer, while the second part is downloaded on all the nodes. In Table 2 a short description of the host program is given, while Table 3 presents the node program where all calculations and communications are performed. The host program simply initiates the computation and collects the results from the single nodes. In the next section the details of the communication are outlined.

4.3. Message-passing for PMGP

The iPSC/2 disposes of a large number of system calls for message-passing. Since updating of boundary values is necessary after one iteration step, some type of synchronization is needed. That means no strict synchronization is needed as in single instruction multiple data (SIMD) machines, where at the same instant of time the same statement on each processor is executed, but instead a type of loose synchronization is needed where the nodes are constrained to intermittently communicate with each other. There are therefore two stages in the algorithm, namely the compute phase (one iteration step) and the communication phase (sending and receiving data). Since a loose synchronization has to be established, the corresponding blocking send and receive commands have to be used; that is, program execution is suspended until the information is moved to the buffer (send) or the information was received. The commands are shown in Table 1.

Table 1. Simple blocking calls as used in PMGP

Message-passing calls:			
	csend(msgid, buf, length, node, pid)		
	crecv(msgid, buf, length)		
Message information calls:			
	infotype()	:	msgid of last message received
	infonode()	:	number of node sending message
	infopid()	:	process id of message
General information calls:			
	myhost()	:	returns host id
	mynode()	:	returns own node number
	mypid()	:	returns own process number

The routine csend sends a message of *length* bytes which resides in array *buf* with the identifier *msgid* (positive integer) to the corresponding nodenumber *node* and processnumber *pid* (positive integer). crecv waits for a message with identifier *msgid* and stores it in array *buf* using *length*. Routine csend blocks the calling process until the message is sent (no wait for acknowledgement), and routine crecv blocks the calling process until the message is received. Tables 2 to 6 show how the communication is performed.

Table 2. Host program for the parallel mesh generation

H1	(Start) Prompt user for the cube dimension and name of input file.
H2	(Start host timers) Initialize elapsed timer and host timer to zero.
H3	(Start nodes) Open the hypercube with the requested dimension. Load the MGP node program. Send block parameters.
H4	(Assemble block data) Assemble block data from each hypercube node. The mapping of the hypercube nodes is simply performed by block number.
H5	(Time) Read node timers and write out all host and node times.
H6	(Close hypercube) Return all nodes to SRM.

Table 3. Node program for the parallel mesh generation

N1	(Start timers) Initialize all node performance timers.
N2	(Initial data) Receive block data from host and send signal to host.
N3	(Start) Receive start signal to host.
N4	(Send block boundaries) Send boundary values to corresponding edges of neighboring block. If neighboring block does not exist, no data are sent.
N5	(Receive block boundaries) Receive boundary values from corresponding edges of neighboring block. If neighboring block does not exist, no data are received.
N6	(Iterate) Do exactly 1 iteration step.
N7	(Converge) Converged? If not go to N4, updating boundaries of neighboring blocks.
N8	(Time) Send node time to host.
N9	(Data) Send block data to host.

5. RESULTS FOR PMGP

5.1. Efficiency and speed-up measurements

An ensemble of 2, 4, 8, 16 and 32 processors was used to run PMGP. For the configuration of 2, 4 and 8 processors there is no processor communicating on all four edges. Only when 16 or more processors are present are there at least some processors with the full communication load. This is reflected in Figure 5. The ratio of boundary points to inner points is $4/N$ (2-D) and $6/N$ (3-D), where N denotes the number of grid points in one dimension.

From Figure 5 it is clear that a certain problem size has to be provided to achieve a good speed-up if 16 or more processors are used. A simple estimate of the time consumed by the message-passing between neighboring blocks (far node) can be done straightforwardly, assuming a message length from 100 to 1000 bytes, resulting in a transfer rate of 0.1 MBytes/s, which is a fairly low transmission speed. Approximately 1 ms is needed for the transfer of a message (Figure 6). Since a maximum of four send and receive calls is necessary in 2-D, this amounts to a total message transfer time of some 8 ms. Therefore, in order to make this time negligible with respect to computation time,

Table 4. The listing shows part of the main iteration loop of a node program. First information is sent and then received from the neighboring nodes

```
.
.
.
c       main iteration loop in node program
        do 30 iter=1, max
c
c       send own block dimensions for corresponding edges of
c       neighboring processors
c       neighboring processors are specified by values of ic(5,*)
c       for a fixed edge (ic(5,*)=0) no neighboring processor exists
c       in that case no information is sent (see subroutine sendbn)
c       same holds for subroutine recvbn
c
        call sendbn
c       receive boundary data from neighboring processors
c...    the denotation of edges as seen from neighboring processor
        call recvbn( ic(5,1), 1, xw, yw, jl)
        call recvbn( ic(5,2), 2, xs, ys, il)
        call recvbn( ic(5,3), 3, xe, ye, jl)
        call recvbn( ic(5,4), 4, xn, yn, il)
c
        emesh = 0.0
c...    update edges
        do 10 edge=1,4
          call bset(ic(3,edge), ic(4,edge), ic(5,edge), ic(6,edge),
        *   xe,xn,xw,xs,ye,yn,yw,ys)
10      continue
c...    perform one iteration step
        call monly
        sum(iter) = emesh
30      continue
c
c...    send information to host
        call csend(55, sum, it4, mh, 1)
        end
```

80 ms should be spent on computation. Assuming a computation power of 1 MFlops and some 80 floating-point operations per grid point for the present problem, there should be 1000 grid points per block, or some 32 points per dimension, which amounts to an l=10 (Figure 6). This is a lower limit, since subroutine calls and arithmetic operations performed with the contents of the messages have not been counted. For 64 points per dimension, a speed-up of 14 is achieved with 16 processors (Figure 6).

It should be noted that the communication time for 32 points and 64 points is nearly the same, since in the first case $2 \times 4 \times 32$ bytes have to be sent, and in the latter one $2 \times 4 \times 64$ bytes are needed (Figure 6). For these message lengths the transfer time is approximately constant, as measurements have shown.

If we now consider the 2-D (or 3-D) flow problem with thermo-chemical non-equilibrium, which is 100–200 times more computationally intensive than the present grid generation problem, the following conclusions can be drawn. For 32 points per

Table 5. Subroutine recvbn receives boundary data from neighboring block and updates respective edge of own block

```
      subroutine recvbn(nbl, edge, xbnd, ybnd, dim)

c     receive edge number pnedge and block dimensions nil(), njl()
c     from neighboring block with block number nbl
      parameter ( maxdim = 32 )
      integer edge, nbl, dim
      real xbnd(maxdim), ybnd(maxdim)
      real rmessg(2*maxdim)

      if (nbl .ne. 0) then
c...  message length in bytes
      call crecv(edge, rmessg, 4*2*maxdim)
      do 10 i=1, dim
        xbnd(i)=rmessg(i)
        ybnd(i)=rmessg(i+dim)
10    continue
      endif
      return
      end
```

Table 6. Subroutine sendbn sends the updated boundary data to neighboring processors. Element ic(6, 1) denotes the east side; ic(6, 2) (not shown) the north side, etc.

```
      subroutine sendbn
c
c...  maxdim = max(ilmax, jlmax)
c...  send x, y boundary to corresponding edge of neighboring block nbl
c     pnedge is corresponding edge of neighboring block
c     il, jl are dimensions of own block
      real x(ilmax, jlmax), y(ilmax, jlmax), ic(6, 6)
      integer i, j, maxdim
      parameter (maxdim = 32)
      real rmessg(2*maxdim)

c...  send values of east side
      if (ic(5,1) .ne. 0) then
      do 10 j=1, jl
        rmessg(j)=x(il-1, j)
        rmessg(jl+j)=y(il-1, j)
10    continue
      call csend(ic(6,1), rmessg, 4*2*jl, ic(5,1)-1, 1)
      endif

      ... SAME CODE FOR OTHER SIDES
      return
      end
```

dimension, some 11 × 4 × 32 bytes have to be communicated, resulting in about 1.2 ms transfer time (as was measured; see Figure 6). Note that the relationship between communication time and message length is non-linear, and longer messages are favored. The maximum bandwidth of about 2.4 MBytes/s for messages to far nodes is obtained for

Table 7. Computing time against problem size and number of processors. All values are in milliseconds. There is a steep increase in time for the 16-processor configuration, due to the full communication load, that is the maximal number of 4 neighboring processors is reached for that configuration

Number of nodes Points per node	1	2	4	8	16	32
30 × 30	8339	8470	8773	9201	12293	14209
60 × 30	17301	17593	18092	18758	23181	27424
60 × 60	35935	36156	36514	37453	49155	52976
120 × 60	73704	73999	74982	77350	102748	109906
120 × 120	149751	150454	151261	159199	204714	206856
240 × 120	305773	308323	310143	320745	401967	417301

Table 8. Efficiency with respect to problem size and number of processors is shown. Values in parenthesis in the last column depict efficiency based on the 16-processor results. A substantial decrease for the 16-processor performance is observed. The reason is that with 16 processors the full communication load is felt, that is, communication to four neighbors takes place

Number of nodes Points per node	1	2	4	8	16	32
30 × 30	1	0.985	0.951	0.906	0.678	0.587 (0.865)
60 × 30	1	0.987	0.960	0.926	0.750	0.633 (0.845)
60 × 60	1	0.994	0.984	0.959	0.731	0.678 (0.928)
120 × 60	1	0.996	0.983	0.953	0.717	0.671 (0.935)
120 × 120	1	0.995	0.990	0.941	0.732	0.724 (0.990)
240 × 120	1	0.992	0.986	0.953	0.761	0.733 (0.963)

a length of 16 KByte. The total transfer time for the flow code therefore is in the range of 10 ms. If a ratio of 10 between computation time and communication time is demanded, at least 100 ms are to be spent on computational work per block. To make calculations simple, 10^4 floating-point operations per grid point are assumed (a number that has been supported by sequential calculations). With a computation power of 1 MFlops, 10 points per block are needed; that is, four interior points per dimension would be sufficient. This rough estimate already shows that parallelization of general flow solvers will lead to very high efficiencies on massively parallel systems.

Table 9. Efficiency values converted to speed-up numbers. Speed-up decreases for the 16-processor configuration, since only for that configuration is the full communication load obtained. Observe that speed-up is approximately a factor of 2 when the 32-processor topology is used. Results are not speed-up measurements in the standard sense, because comparison is with a monoblock example where no communication takes place. However, a 16-block example on a uniprocessor would have the same communication load as the parallel version. Therefore the results here are a lower bound. As the results from 16 to 32 processors indicate, linear speed-up is received

Number of nodes Points per node	1	2	4	8	16	32
30 × 30	1	1.970	3.804	7.248	10.848	18.784 (1.730)
60 × 30	1	1.974	3.840	7.408	12.000	20.256 (1.690)
60 × 60	1	1.988	3.936	7.672	11.696	21.696 (1.856)
120 × 60	1	1.992	3.932	7.624	11.472	21.472 (1.870)
120 × 120	1	1.990	3.960	7.528	11.712	23.168 (1.988)
240 × 120	1	1.984	3.944	7.624	12.176	24.236 (1.990)

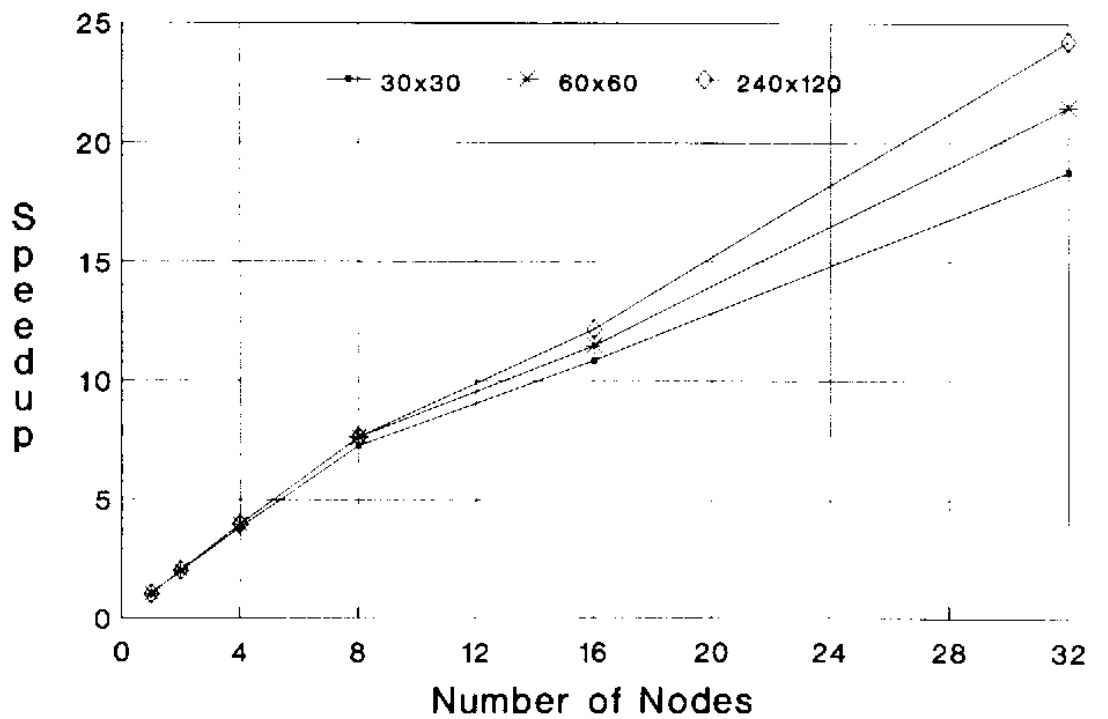

Figure 5. Speed-up against the number of nodes. Problem size (points per node) used as parameter

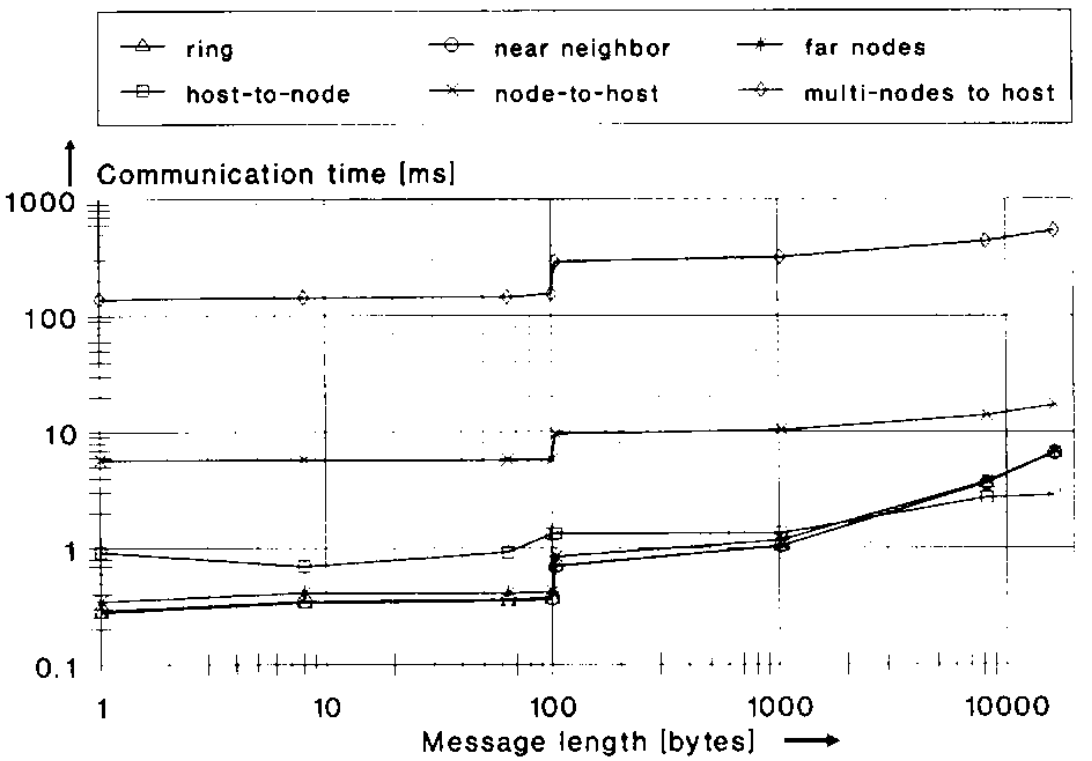

Figure 6. Measured communication time against message length for various topologies on the iPSC/2. Both scales are logarithmic

5.2. Discussion and outlook: parallelization of multi-block CFD-codes

Multi-block codes, introduced into CFD to handle the most geometrically complex configurations in 2-D and 3-D, while retaining the computational efficiency of finite differences, are unique with regard to parallelization. The implementation of matching (overlapping) multi-block grids on massively parallel systems is straightforward, since this algorithmic approach is inherently parallel; their implementation on a sequential machine in the past was only dictated by the non-availability of parallel systems. On a von-Neumann computer, blocks have to be processed in sequential order, performing one iteration step, and then the boundaries have to be updated by sending rows or columns from neighboring blocks. No loss in performance caused by domain decomposition will occur, since the possible convergence slowdown generated by the multi-block structure is also present in the sequential case. This shows that exactly the same algorithm can be used on parallel systems; hence the parallel algorithm has in principal no additional overhead compared to the sequential code. Modifications are needed only to implement the message-passing routines from the parallel OS. The resulting shape in the CD is of no importance; there is only a set of connected rectangles or boxes, each with its own local CS. How edges or faces of neighboring blocks are matched to each other is of no concern to the user. The orientation of the local CSs with respect to each other is completely done by the program[5], which is, however, much more complicated than the relatively straightforward implementation of the message transfer. The advantage of a local CS is that edges that have to be communicated are easily identified. If, for example, completely irregular grids—as in finite elements—were used, some type of local

CS has to be introduced (e.g. Chap 8 of Reference 13), in order to identify the matching points. Despite other disadvantages of finite elements as against finite differences (or finite volume) in CFD[2], a local CS makes the assumption of a completely irregular grid invalid. In addition, as is shown in Chap. 7 of Reference 13, parallelization of finite elements demands a much bigger effort compared with multi-block grids.

The algorithm of the present paper works not only for any arbitrary 2-D geometry, but also demonstrates that there is no dependence on arrays or any predefined topology of processors. No nearest-neighbor topology is used because the block number simply determines the processor, thus leading to the most general approach. Although PMGP is not a computing-intensive application, very good speed-ups can be achieved with 1000 or more points per block. Since the CFD code for the calculation of thermo-chemical non-equilibrium hypersonic flow fields is a factor of 100–200 more computing-intensive than the present problem, using exactly the same data structure and topology, it is obvious that the use of parallel systems will result in nearly optimal efficiencies for a very large number of nodes (1000 or more). Therefore, the next step is to parallelize the code described in Reference 9, which is the theme of Part II of this paper[8]. Although special libraries are available on some parallel systems, e.g. Suprenum[9], Sec. 4.3 has shown that only a handful of routines are necessary for loosely synchronized problems. These routines, which are very basic, are available on all message-passing systems. Therefore, the implementation of the present code on other massively parallel systems demands only minor changes; e.g. a direct implementation on the CrOS III as described in Reference 13 or on the Suprenum machine would be possible.

From the foregoing it is clear that the overlapping multi-block concept is ideally suited for massively parallel systems, combining great geometric flexibility with high efficiency and speedup for a very wide class of PDEs. It has been shown that for the simple system of two elliptical grid generation equations, high speed-up is obtained, and from equation (12) high efficiency for a large number of nodes can be predicted for all problems with sufficient computational demand, which is satisfied in applications in CFD. Thus the multi-block concept is a viable tool to gain the orders of magnitude in computation power needed for future aerospace applications.

ACKNOWLEDGEMENTS

The authors are very grateful to H. Schwamborn and H. Mierendorff from the GMD, Bonn, for providing access to the iPSC/2 and for advice with the OS as well as for many helpful discussions. The co-operation with Prof. Hirschel, Dr. Eberle and Dr. Schmatz from the Military Aircraft Division of MBB, Deutsche Aerospace is gratefully acknowledged. The support and encouragement provided by W. Berry, Aerothermodynamics and Propulsion Division, ESTEC, is appreciated.

REFERENCES

1. G.T. Chapman, 'An overview of hypersonic aerothermodynamics', *Commun. Appl. numer. methods*, **4**, 319–325 (1988).

2. J. Häuser, *et al.*, *Supercomputing in Aerospace*, to be published by Institute of Computational Mechanics, UK, 1990.
3. J. Häuser *et al.*, 'Boundary conformed coordinate systems for selected 2D fluid flow problems', Part I, *J. Numer. methods fluids*, 507–527 (1986).
4. R.M. Coleman, 'INMESH: an interactive program for numerical grid generation', DTNSRCDC-851054, 33 pp., 1985.
5. J. Häuser, H.G. Paap, H. Wong and M. Spel, 'Grid⋆: a general multiblock surface and volume grid generation toolbox', in *Numerical Grid Generation in Computational Fluid Dynamics and Related Fields*, A. Arcilla *et al.* (Eds.), North-Holland, 1991, pp. 817–838.
6. H. Schwamborn and W. Gentzsch, 'Solution of partial differential equations on the Intel iPSC/2 hypercube', GMD-Report, Bonn-Birlinghofen, 1990.
7. J.L. Gustofson *et al.*, 'Development of parallel methods for a 1024-processor hypercube', *SIAM, J. of Scientific and Statistical Computing*, **9**(4), July, 609–638 (1988).
8. J. Häuser *et al.*, 'Parallel computing in aerospace using multi-block grids. Part II: Application to hypersonic flow problems', to be published, *Concurrency: practice & experience*, 1992.
9. L. Bohmans, and R. Hempel, 'The Argonne IGMD macros in Fortran for portable parallel programming and their implementation on the Intel-iPSC/2', Arbeitspaper der GMD406, 1989.
10. W.K. Giloi, 'SUPRENUM: a trendsetter in modern supercomputer development', *Parallel Comput.*, **7**, North-Holland, 283–296 (1988).
11. P. Schuller, 'Hypersonic system am Beispiel iPSC/2', GI Paris Workshop, 10–12 April 1989.
12. A. Lin, 'Parallel numerical algorithms for fluid dynamics simulation', AIAA90-0333, 1990.
13. G. Fox *et al.*, *Solving Problems on Concurrent Processors, Vol. 1*, Prentice Hall, 1988, 592 pp.
14. W. Gentzsch and J. Häuser, 'Mesh generation on parallel computers', in *Numerical Grid Generation in Computational Fluid Dynamics*, S. Sengupta *et al.* (Eds.), Pineridge Press, 1988, pp. 113–124.
15. G. Scoon, *et al.*, *Mission to Mars*, esa SP-1117, 1989.
16. R. Duncan, 'A survey of parallel computer architectures', *Computer*, IEEE, **23**(2), 5–16 (1990).
17. *Conference Proceedings: Parallel CFD 89*, Intel Corporation, 1989.
18. G. Fox. *et al.*, *Solving Problems on Concurrent Computers, Vol. II*, Prentice Hall, 805 pp., 1990.
19. D. Hänel and E. Krause, 'Finite approximations in fluid mechanics II', DFG Priority Research Program, Results 1986–1988, Wiesbaden: Vieweg, 1989.