CHAPTER TWENTY

Computing Techniques

In Part Two and Part Three, various numerical schemes in CFD including FDM, FEM, and FVM have been discussed. We have presented methods of grid generation and adaptive meshing in both structured and unstructured grids in Part Four. Equation solvers for both linear and nonlinear algebraic equations resulting from FDM, FEM, and FVM have also been discussed in appropriate chapters. We are now at the stage of embarking on extensive CFD calculations in large-scale industrial problems, which will be presented in Part Five. To this end, it is informative to examine computational aspects associated with supercomputer applications and multi-processors. Among them are the domain decomposition methods (DDM), multigrid methods (MGM), and parallel processing. In DDM the domain of study is partitioned into substructures to make solvers perform more efficiently with reduction of storage requirements, whereas in MGM the solution convergence is accelerated with low-frequency errors being removed through coarse mesh configurations and with high-frequency errors removed through fine mesh configurations. These two methods lend themselves to parallel processing to speed up and reduce computer time. Development of parallel programs and both static and dynamic load balancing will be presented. The topics in this chapter are designed toward more robust computational strategies in dealing with geometrically complicated, large-scale CFD problems. Some selected example problems are also included.

20.1 DOMAIN DECOMPOSITION METHODS

In dealing with geometrically large, complicated systems, it is natural to seek an approach to split the domain into small pieces, known as domain decomposition methods (DDM). This is one of many possible applications to parallel processing to be discussed in Section 20.3. The basic idea of DDM was originated from the concept of linear algebra in solving the partial differential equations iteratively in subdomains, known as the Schwarz method [Schwarz, 1869]; and subsequently implemented in applications [Lions, 1988; Glowinski and Wheeler, 1988, among others]. The main advantages of DDM include efficiency of solvers, savings in computational storage conducive to parallel processing, and applications of different differential equations in different subdomains (representing viscous flow in one subdomain and inviscid flow in another subdomain, for example).

There are two approaches in the Schwarz method: (1) Multiplicative procedure which resembles the block Gauss-Seidel iteration, and (2) Additive procedure analogous to a block Jacobi iteration. We elaborate these procedures in the following sections.

20.1.1 MULTIPLICATIVE SCHWARZ PROCEDURE

In a typical domain decomposition approach, we divide the domain Ω into subdomains Ω_i such that

$$\Omega = \bigcup_{i=1}^{n} \Omega_i \tag{20.1.1}$$

an example of which is shown in Figure 20.1.1 In this example, there are three interior domains, $\Omega_1(1-12)$, $\Omega_2(13-21)$, $\Omega_3(22-27)$, and three boundary interfaces, $\Gamma_{1,2}$, $\Gamma_{1,3}$, $\Gamma_{2,3}(28-36)$. Here, for simplicity, boundary interface nodes are labeled last.

Let us consider the Poisson equation and the resulting matrix equations from FDM, FEM, or FVM formulations for this geometry in the form,

$$\begin{bmatrix} \underset{27\times 27}{K_{aa}} & \underset{27\times 9}{K_{ab}} \\ \underset{9\times 27}{K_{ba}} & \underset{9\times 9}{K_{bb}} \end{bmatrix} \begin{bmatrix} \underset{27\times 1}{U_a} \\ \underset{9\times 1}{U_b} \end{bmatrix} = \begin{bmatrix} \underset{27\times 1}{F_a} \\ \underset{9\times 1}{F_b} \end{bmatrix} \tag{20.1.2}$$

where the subscripts a, b denote the interior subdomains and interfaces, respectively, as related to the global stiffness matrix K_{aa} (27×27) with the subdomain stiffness matrices, $K_1(12 \times 12)$, $K_2(9 \times 9)$, $K_3(6 \times 6)$ for Ω_1, Ω_2, Ω_3, respectively, and the boundary interface stiffness matrix, $K_{bb}(9 \times 9)$ together with the interface-subdomain interaction stiffness matrices $K_{ab}(27 \times 9)$ and $K_{ba}(9 \times 27)$ as shown in Figure 20.1.2. From the subdomain equations, we obtain

$$U_a = K_{aa}^{-1}(F_a - K_{ab}U_b) \tag{20.1.3}$$

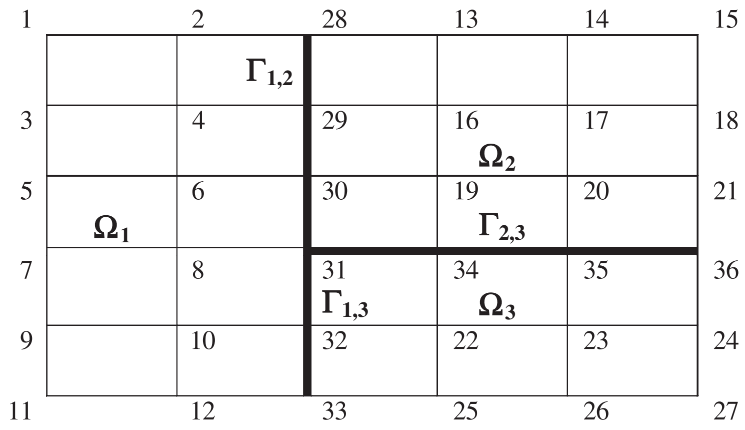


Figure 20.1.1 Decomposed domain (subdomains): Interior nodes (1–27), subdomain Ω_1 (1–12), subdomain Ω_2 (13–21), subdomain Ω_3 (22–27), Interfaces Γ_{12}, Γ_{13}, Γ_{23} (28–36).

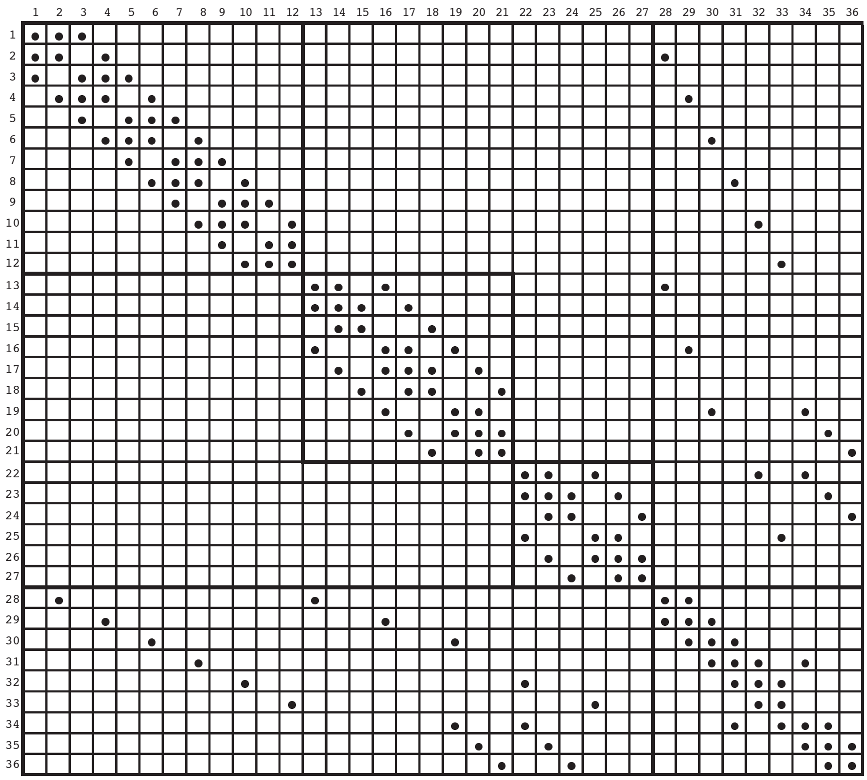


Figure 20.1.2 Global stiffness matrix, $K_{aa}(27 \times 27)$ for Figure 20.1.1 with the subdomain stiffness matrices $K_1(12 \times 12)$, $K_2(9 \times 9)$, $K_3(6 \times 6)$, for Ω_1, Ω_2, Ω_3, respectively, and the boundary interface stiffness matrix, $K_{bb}(9 \times 9)$ together with the interface-subdomain interaction stiffness matrices $K_{ab}(27 \times 9)$ and $K_{ba}(9 \times 27)$.

Substituting (20.1.3) into the interface equations leads to

$$S_{bb}U_b = F_b - K_{ba}K_{aa}^{-1}F_a \tag{20.1.4}$$

with

$$S_{bb} = K_{bb} - K_{ba}K_{aa}^{-1}K_{ab} \tag{20.1.5}$$

which is known as the Schur complement matrix. Note that determination of the unknowns U_a, U_b requires the matrix inversion, K_{aa}^{-1}. To avoid this inversion operation, we employ the block Gaussian elimination approach as follows: First we return to (20.1.3) and write in the form

$$U_a = F_a^* - K_{ab}^* U_b \tag{20.1.6}$$

with

$$F_a^* = K_{aa}^{-1}F_a \tag{20.1.7}$$

$$K_{ab}^* = K_{aa}^{-1}K_{ab} \tag{20.1.8}$$

Premultiplying F_a^* by K_{aa}, and K_{ab}^* by K_{aa}, we obtain, respectively,

$$K_{aa}F_a^* = K_{aa}K_{aa}^{-1}F_a = F_a \tag{20.1.9}$$

$$K_{aa}K_{ab}^* = K_{aa}K_{aa}^{-1}K_{ab} = K_{ab} \tag{20.1.10}$$

Now, any standard equation solver may be used to solve F_a^* and K_{ab}^* from (20.1.9) and (20.1.10), respectively. We then compute

$$F_b^* = F_b - K_{ba}F_a^* \tag{20.1.11}$$

and the Schur complement matrix in the form

$$S_{bb} = K_{bb} - K_{ba}K_{ab}^* \tag{20.1.12}$$

Finally, we solve the interface unknowns U_b using (20.1.11) and (20.1.12) from

$$S_{bb}U_b = F_b^* \tag{20.1.13}$$

and the interior subdomain unknowns using (20.1.9) and (20.1.10) from (20.1.3)

$$U_a = F_a^* - K_{ab}^*U_b \tag{20.1.14}$$

It is well known that any system of equations may be altered in such a manner that conditioning of the equations (eigenvalues) can be improved in order to assure accuracy. To this end, let us examine the global equation of the form

$$\underset{n\times n}{K}\ \underset{n\times 1}{U} = \underset{n\times 1}{F} \tag{20.1.15}$$

The preconditioned system of (20.1.15) may be written as

$$M^{-1}KU = M^{-1}F \tag{20.1.16}$$

where M is the preconditioning matrix and M^{-1} is the preconditioning operator. This is called the multiplicative Schwarz procedure which is equivalent to a block Gauss-Seidel iteration. In order to derive this preconditioning operator, we seek the restriction operator R_i and the prolongation operator (transpose of the restriction operator) with the subscript i denoting the number of subdomains such that

$$\underset{(n_i\times n_i)}{K_i} = \underset{(n_i\times n)}{R_i}\ \underset{(n\times n)}{K}\ \underset{(n\times n_i)}{R_i^T} \tag{20.1.17}$$

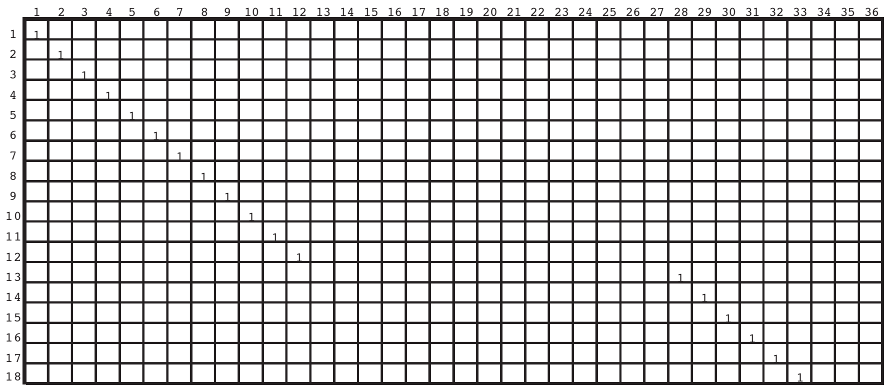
or

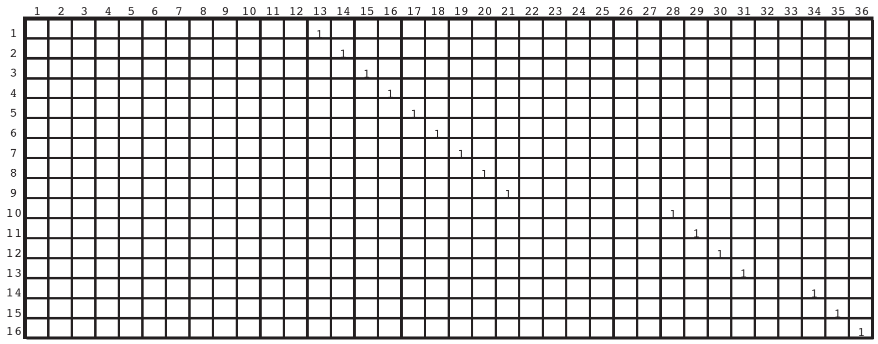
$$K^{-1} = R_i^T K_i^{-1} R_i \tag{20.1.18}$$

where the n_i refers to the total number of nodes for each subdomain and its boundary interface. Note that the subscript i here is not a tensorial index. For example, for the geometry represented by Figure 20.1.1, we have $n = 36$ and n_i for Ω_1, Ω_2, Ω_3 are 18, 16, 12, respectively, leading to the global stiffness matrix K shown in Figure 20.1.2. Here, the restriction matrices R_i consist of ones at associated nodes and zeros elsewhere (Figure 20.1.3), resulting in subdomain stiffness matrices as shown in Figure 20.1.4.

Let us assume that at each iterative solution step there is an error given by the error vector d,

$$d = U^* - U \tag{20.1.19}$$


R_1 (18×36)


R_2 (16×36)

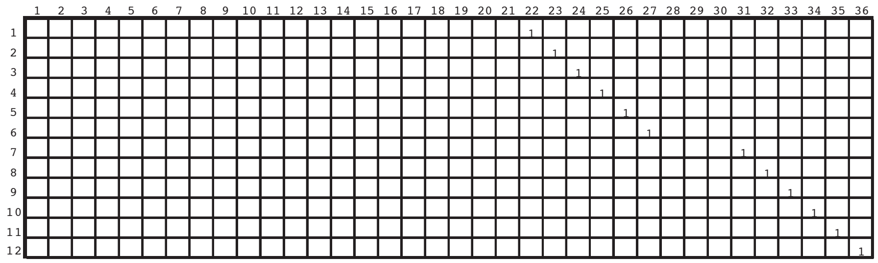

R_3 (12×36)

Figure 20.1.3 Restriction operators for subdomains given in Figure 20.1.1.

where U^* is the solution at the current step with U being the previous step. Then, we have

$$F - KU = Kd = K(U^* - U) \tag{20.1.20}$$

It follows from the above relations that

$$d = K^{-1}(F - KU) \tag{20.1.21}$$

$$U^* = U + R_i^T K_i^{-1} R_i (F - KU) \tag{20.1.22}$$

	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18
1	$K_{1,1}$	$K_{1,2}$	$K_{1,3}$	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
2	$K_{2,1}$	$K_{2,2}$	0	$K_{2,4}$	0	0	0	0	0	0	0	0	$K_{2,28}$	0	0	0	0	0
3	$K_{3,1}$	0	$K_{3,3}$	$K_{3,4}$	$K_{3,5}$	0	0	0	0	0	0	0	0	0	0	0	0	0
4	0	$K_{4,2}$	$K_{4,3}$	$K_{4,4}$	0	$K_{4,6}$	0	0	0	0	0	0	0	$K_{4,29}$	0	0	0	0
5	0	0	$K_{5,3}$	0	$K_{5,5}$	$K_{5,6}$	$K_{5,7}$	0	0	0	0	0	0	0	0	0	0	0
6	0	0	0	$K_{6,4}$	$K_{6,5}$	$K_{6,6}$	0	$K_{6,8}$	0	0	0	0	0	0	$K_{6,30}$	0	0	0
7	0	0	0	0	$K_{7,5}$	0	$K_{7,7}$	$K_{7,8}$	$K_{7,9}$	0	0	0	0	0	0	0	0	0
8	0	0	0	0	0	$K_{8,6}$	$K_{8,7}$	$K_{8,8}$	0	$K_{8,10}$	0	0	0	0	0	$K_{8,31}$	0	0
9	0	0	0	0	0	0	$K_{9,7}$	0	$K_{9,9}$	$K_{9,10}$	$K_{9,11}$	0	0	0	0	0	0	0
10	0	0	0	0	0	0	0	$K_{10,8}$	$K_{10,9}$	$K_{10,10}$	0	$K_{10,12}$	0	0	0	0	$K_{10,32}$	0
11	0	0	0	0	0	0	0	0	$k_{11,9}$	0	$k_{11,11}$	$k_{11,12}$	0	0	0	0	0	0
12	0	0	0	0	0	0	0	0	0	$k_{12,10}$	$k_{12,11}$	$k_{12,12}$	0	0	0	0	0	$k_{12,33}$
13	0	$k_{13,2}$	0	0	0	0	0	0	0	0	0	0	$k_{13,28}$	$K_{13,29}$	0	0	0	0
14	0	0	0	$k_{14,4}$	0	0	0	0	0	0	0	0	$k_{14,28}$	$K_{14,29}$	$k_{14,30}$	0	0	0
15	0	0	0	0	0	$k_{15,6}$	0	0	0	0	0	0	0	$K_{15,29}$	$k_{15,30}$	$k_{15,31}$	0	0
16	0	0	0	0	0	0	0	$k_{16,8}$	0	0	0	0	0	0	$k_{16,30}$	$k_{16,31}$	$k_{16,32}$	0
17	0	0	0	0	0	0	0	0	0	$k_{17,10}$	0	0	0	0	0	$k_{17,31}$	$k_{17,32}$	$k_{17,33}$
18	0	0	0	0	0	0	0	0	0	0	0	$k_{18,12}$	0	0	0	0	$k_{18,32}$	$k_{18,33}$

	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
1	$K_{1,13}$	$K_{1,14}$	0	$K_{1,16}$	0	0	0	0	0	$K_{1,28}$	0	0	0	0	0	0
2	$K_{2,13}$	$K_{2,14}$	$K_{2,15}$	0	$K_{2,17}$	0	0	0	0	0	0	0	0	0	0	0
3	0	$K_{3,14}$	$K_{3,15}$	0	0	$K_{3,18}$	0	0	0	0	0	0	0	0	0	0
4	$K_{4,13}$	0	0	$K_{4,16}$	$K_{4,17}$	0	$K_{4,19}$	0	0	0	$K_{4,29}$	0	0	0	0	0
5	0	$K_{5,14}$	0	$K_{5,16}$	$K_{5,17}$	$K_{5,18}$	0	$K_{5,20}$	0	0	0	0	0	0	0	0
6	0	0	$K_{6,15}$	0	$K_{6,17}$	$K_{6,18}$	0	0	$K_{6,21}$	0	0	0	0	0	0	0
7	0	0	0	$K_{7,16}$	0	0	$K_{7,19}$	$K_{7,20}$	0	0	0	$K_{7,30}$	0	$K_{7,34}$	0	0
8	0	0	0	0	$K_{8,17}$	0	$K_{8,19}$	$K_{8,20}$	$K_{8,21}$	0	0	0	0	0	$K_{8,35}$	0
9	0	0	0	0	0	$K_{9,18}$	0	$K_{9,20}$	$K_{9,21}$	0	0	0	0	0	0	$K_{9,36}$
10	$K_{10,13}$	0	0	0	0	0	0	0	0	$K_{10,28}$	$K_{10,29}$	0	0	0	0	0
11	0	0	0	$k_{11,16}$	0	0	0	0	0	$k_{11,28}$	$k_{11,29}$	$k_{11,30}$	0	0	0	0
12	0	0	0	0	0	0	$k_{12,19}$	0	0	0	$k_{12,29}$	$k_{12,30}$	$k_{12,31}$	0	0	0
13	0	0	0	0	0	0	0	0	0	0	0	$k_{13,30}$	$k_{13,31}$	$k_{13,34}$	0	0
14	0	0	0	0	0	0	$k_{14,19}$	0	0	0	0	0	$k_{14,31}$	$k_{14,34}$	$k_{14,35}$	0
15	0	0	0	0	0	0	0	$k_{15,20}$	0	0	0	0	0	$k_{15,34}$	$k_{15,35}$	$k_{15,36}$
16	0	0	0	0	0	0	0	0	$k_{16,21}$	0	0	0	0	0	$k_{16,35}$	$k_{16,36}$

	1	2	3	4	5	6	7	8	9	10	11	12
1	$K_{1,22}$	$K_{1,23}$	0	$K_{1,25}$	0	0	0	$K_{1,32}$	0	$K_{1,34}$	0	0
2	$K_{2,22}$	$K_{2,23}$	$K_{2,24}$	0	$K_{2,26}$	0	0	0	0	0	$K_{2,35}$	0
3	0	$K_{3,23}$	$K_{3,24}$	0	0	$K_{3,27}$	0	0	0	0	0	$K_{3,36}$
4	$K_{4,22}$	0	0	$K_{4,25}$	$K_{4,26}$	0	0	0	$K_{4,33}$	0	0	0
5	0	$K_{5,23}$	0	$K_{5,25}$	$K_{5,26}$	$K_{5,27}$	0	0	0	0	0	0
6	0	0	$K_{6,24}$	0	$K_{6,26}$	$K_{6,27}$	0	0	0	0	0	0
7	0	0	0	0	0	0	$K_{7,31}$	$K_{7,32}$	0	$K_{7,34}$	0	0
8	$K_{8,22}$	0	0	0	0	0	$K_{8,31}$	$K_{8,32}$	$K_{8,33}$	0	0	0
9	0	0	0	$K_{9,25}$	0	0	0	$K_{9,32}$	$K_{9,33}$	0	0	0
10	$K_{10,22}$	0	0	0	0	0	$K_{10,31}$	0	0	$K_{10,34}$	$K_{10,35}$	0
11	0	$k_{11,23}$	0	0	0	0	0	0	0	$K_{11,34}$	$k_{11,35}$	$k_{11,36}$
12	0	0	$k_{12,24}$	0	0	0	0	0	0	0	$k_{12,35}$	$k_{12,36}$

Figure 20.1.4 Final forms of stiffness matrices.

Define the error e^* to be the difference between the right-hand side and the left-hand side of (20.1.22),

$$e^* = e - R_i^T K_i^{-1} R_i K(U^* - U) \tag{20.1.23}$$

which may be rewritten for subiteration steps i and $i - 1$ as

$$e_i = e_{i-1} - R_i^T K_i^{-1} R_i K e_{i-1} \tag{20.1.24}$$

with $i = 1, \ldots s$, s being the total number of subdomains. This gives

$$e_i = (I - P_i)e_{i-1} \tag{20.1.25}$$

where P_i is known as the projector,

$$P_i = R_i^T K_i^{-1} R_i K \tag{20.1.26}$$

For the error at step s, we have

$$e_s = (I - P_s)(I - P_{s-1}) \ldots (I - P_1)e_0 \tag{20.1.27}$$

or

$$e_s = Q_s e_0 \tag{20.1.28}$$

with

$$Q_s = (I - P_s)(I - P_{s-1}) \ldots (I - P_1)$$

The multiplicative Schwarz procedure described above may be extended to overlapping subdomains, which will be elaborated in Section 20.4.1 together with parallel processing.

20.1.2 ADDITIVE SCHWARZ PROCEDURE

In contrast to the multiplicative Schwarz procedure, which is similar to the block Gauss-Seidel iteration, the additive Schwarz procedure consists of updating all the new block components from the same residual, analogous to a block Jacobi iteration, and thus the components in each subdomain are not updated until a whole cycle of updates through all domains is completed.

It follows from (20.1.22) and (20.1.26) that

$$U^* = \left(I - \sum_{i=1}^{s} P_i \right) U + \sum_{i=1}^{s} T_i F \tag{20.1.29}$$

with

$$T_i = P_i K^{-1} = R_i^T K_i^{-1} R_i \tag{20.1.30}$$

Note that, upon convergence, $U^* = U$, the solution (20.1.29) becomes

$$\sum_{i=1}^{s} P_i U = \sum_{i=1}^{s} T_i F \tag{20.1.31}$$

Comparing (20.1.16) and (20.1.31), we find that

$$\begin{aligned} \sum_{i=1}^{s} P_i &= M^{-1} K \\ \sum_{i=1}^{s} T_i &= \sum_{i=1}^{s} P_i K^{-1} = M^{-1} \end{aligned} \tag{20.1.32}$$

which identifies the preconditioning as given by (20.1.16),

$$M^{-1} K U = M^{-1} F$$

It is seen that the preconditioned iterative solution (20.1.29) has multiple benefits. Here, only the restricted and prolongated subdomain matrices are involved, the solution is more accurate due to preconditioning, convergence is faster, and computational storage requirements are less with domain decomposition.

The domain decomposition may be carried out in unstructured grids. The basic algebra for the structured grids presented above can be applied equally well to the unstructured grids. Furthermore, the domain decomposition lends itself to parallel processing which will be presented in Section 20.3. Examples of both overlapping and nonoverlapping subdomains together with parallel processing will be presented in Section 20.3.4.

20.2 MULTIGRID METHODS

20.2.1 GENERAL

The basic idea of multigrid methods (MGM), as originally pioneered by Brandt [1972, 1977, 1992], is to accelerate the convergence of iterative solvers. The low-frequency or large wavelength components of error on a fine mesh become high frequency or small wavelength components on a coarser mesh. Thus, it is preferable to use coarse grids to remove low-frequency errors, with accuracy ensured by means of fine grids. Two or more levels of solutions from fine to coarse grids (restriction process) and from coarse to fine grids (prolongation process) may be repeated until convergence is reached. In general, MGM is regarded as the most efficient technique to accelerate convergence among the iterative methods in solving the linear and nonlinear algebraic equations.

In multigrid operations, asymptotic behavior of the error (or of the residual) is dominated by the eigenvalues of the amplification matrix close to one in absolute value. The error components situated in the low-frequency range of the spectrum of the space-discretization are the slowest to be damped in the iterative process. The higher frequencies are the first to be reduced and a large part of the high-frequency error components will be damped, thus acting as a smoother of the error.

The simplest case of a multigrid procedure consists of nested structured grid in which a fine grid is coarsened by eliminating every other node in all directions so that all nodes in the coarse mesh appear in the fine mesh. In contrast, unstructured grids are in general unnested. We present the general procedure of nested structured multigrid methods in Section 20.2.2, followed by unnested unstructured multigrid methods in Section 20.2.3.

20.2.2 MULTIGRID SOLUTION PROCEDURE ON STRUCTURED GRIDS

For structured grid FDM computations, we may begin with the finest grid and coarsen the mesh by eliminating every other node, resulting in nested grids. An example for the three-level nested multigrid system is shown in Figure 20.2.1. In practice, several levels of multigrid discretization are desirable. The simplest descriptions of multigrid methods may be given as follows:

Restriction Process

Do n iterations (two or three relaxation sweeps) on the fine grid using any iterative solution method such as the Gauss-Seidel scheme. Interpolate the residual R onto the

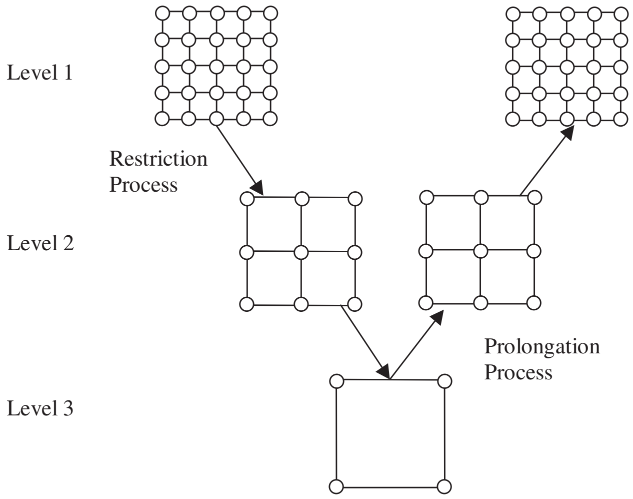


Figure 20.2.1 Three-level multigrid discretization.

coarse grid, with the residuals not updated during the iterations because we want the computed values to represent corrections to the fine-grid solution. This is known as the restriction process (restricted to the next coarsest grid), represented by the following steps:

Step 1 (finest grid)	$LU_1 = R_1$
Step 2 (level 2 coarse grid)	$L\Delta U_2 = -I_1^2 R_1$
Step 3 (level 3 coarse grid)	$L\Delta U_3 = -I_2^3 R_2 = -I_2^3\left(L\Delta U_2 + I_1^2 R_1\right)$

.

.

where L is the differential or difference operator and the subscript represents the previous step with the superscript being the destination. For illustration, the restriction process of two-level one-dimensional Poisson problems is shown in Figure 20.2.2(a) with the restriction operator I_1^2 given by

$$I_1^2 = \frac{1}{4}\begin{bmatrix} 1 & 2 & 1 & & \\ & & 1 & 2 & 1 \end{bmatrix}$$

Prolongation Process

With the corrections now available on the coarsest grid, they are to be prolongated or interpolated onto the next finer grid by adding them to the previous corrections obtained at the restriction process, until the finest grid is reached. These steps are given as follows:

Step 4 (level 3 fine grid)	$L\Delta U_3 = -I_3^2 R_3$	(updated from Step 3)
Step 5 (level 2 fine grid)	$L\Delta U_2 = -I_2^1 R_2$	(updated from Step 2)
Step 6 (level 1 fine grid)	$L\Delta U_1 = R_1$	(updated from Step 1)

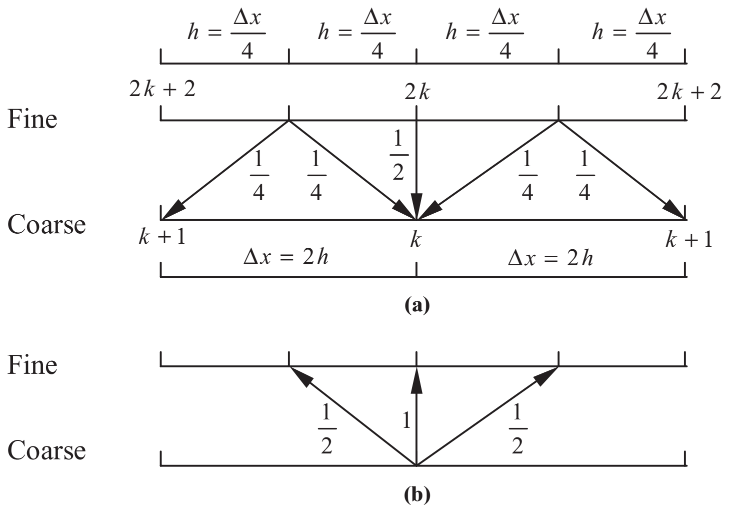


Figure 20.2.2 Restriction and prolongation. (**a**) Restriction (weighting). (**b**) Prolongation (interpolation).

where the subscript denotes the previous step with the superscript being the destination. Here, the prolongation operator for one-dimensional Poisson solver assumes the form,

$$I_2^1 = \frac{1}{2}\begin{bmatrix} 1 & \\ 2 & \\ 1 & 1 \\ & 2 \\ & 1 \end{bmatrix}$$

The above procedure is known as the V-cycle [Figure 20.2.3(a)]. The V-cycle may be repeated as many times as required until the desired accuracy is obtained, resulting in the so-called W-cycles [Figure 20.2.3(b)].

To illustrate the MGM procedure using the FEM notations, let us assume that we have a sequence of grids m and $m+1$ to solve the FEM equations of the form,

$$K_{\alpha\beta}^{m,m} U_\beta^m = F_\alpha^m \qquad \text{on coarse grid} \tag{20.2.1}$$

$$K_{\alpha\beta}^{m+1,m+1} U_\beta^{m+1} = F_\alpha^{m+1} \qquad \text{on fine grid} \tag{20.2.2}$$

with α, β denoting the number of global nodes for either coarse grid or fine grid. A low-frequency component on a fine grid becomes a high-frequency component on a

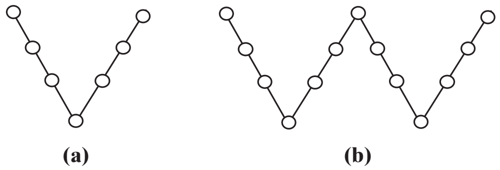


Figure 20.2.3 (**a**) V-cycle multigrid. (**b**) W-cycle multigrid.

coarse grid. Thus, the multigrid methods are intended for exploiting the high-frequency smoothing of the relaxation (iteration) procedure.

The coarse grid equation (20.2.1) for U_α^m is prolongated onto the next finer grid (20.2.2). After a few steps of this iterative process, the high-frequency components of the residual E^{m+1} are obtained

$$E_\alpha^{m+1} = F_\alpha^{m+1} - K_{\alpha\beta}^{m+1,m+1} U_\beta^{m+1} \tag{20.2.3}$$

The residual can then be reduced and adequately resolved on the coarse grid:

$$K_{\alpha\beta}^{m,m} \overline{U}_\beta^m = K_{\alpha\beta}^{m,m+1} E_\beta^{m+1} = \overline{F}_\alpha^m \tag{20.2.4}$$

where $\overline{U}_\beta^m$ is the correction on the coarse grid and $K_{\alpha\beta}^{m,m+1}$ is the nonsquare matrix, known as the restriction operator. For nonlinear problems we may replace (20.2.3) by

$$K_{\alpha\gamma}^{m,m}\left(K_{\gamma\beta}^{m,m+1} U_\beta^{m+1} + \overline{U}_\gamma^m\right) = \overline{F}_\alpha^m + K_{\alpha\gamma}^{m,m} K_{\gamma\beta}^{m,m+1} U_\beta^{m+1} \tag{20.2.5}$$

or

$$K_{\alpha\beta}^{m,m} \overline{U}_\beta^m = \overline{F}_\alpha^m \tag{20.2.6}$$

The solution of either (20.2.4) for linear problems or (20.2.6) for nonlinear problems enables U_β^{m+1} to be updated by adding to it the prolongation of $\overline{U}_\beta^m$ onto the finer grid so that U_β^{m+1} as calculated from (20.1.2) is updated to $\overline{U}_\beta^{m+1}$ as

$$\overline{U}_\alpha^{m+1} = U_\alpha^{m+1} + K_{\alpha\gamma}^{m+1,m}\left(U_\gamma^m - K_{\gamma\beta}^{m,m+1} U_\beta^{m+1}\right) \tag{20.2.7}$$

where $K_{\alpha\gamma}^{m+1,m}$ is the nonsquare matrix, known as the prolongation operator. The procedure described above will be repeated until the converged solution of (20.2.2) is obtained.

If FDM discretizations are employed, the restriction and prolongation operators can be replaced by appropriate finite difference formulas. To identify these operators, let us begin with the FDM formulations using the FEM notations.

$$K_{\alpha\beta}^{m+1} \Delta U_\beta^{m+1} = F_\alpha^{m+1} - K_{\alpha\beta}^{m+1} U_\beta^m = E_\alpha^{m+1} \tag{20.2.8}$$

with

$$U_\alpha^{m+1} = U_\alpha^m + \Delta U_\alpha^{m+1} \tag{20.2.9}$$

The residual E_α^m upon a few relaxation steps on the $(m+1)$th grid to smooth the high-frequency components is of the form

$$\overline{E}_\alpha^m = E_\alpha^m - K_{\alpha\beta}^m \Delta \overline{U}_\beta^m \tag{20.2.10}$$

where $\Delta\overline{U}_\beta^m$ is obtained through a few relaxation steps.

The residual E_α^{m-1} on the mth grid is obtained from $\overline{E}_\alpha^m$ as

$$E_\alpha^{m-1} = I_r^{m-1} E_{\alpha r}^m \tag{20.2.11}$$

which represents the transfer from the fine to the coarse grid with I_r^{m-1} being the restriction operator similar to $K_{\alpha\beta}^{m,m+1}$ in (20.2.4). This operator shows how the mesh values on the coarse grid are derived from the surrounding fine mesh values. This is a contrast to the nonsquare matrix $K_{\alpha\beta}^{m,m+1}$ relating the stiffness matrix of the coarse grid to the fine grid.

These relaxation processes (20.2.10) and restriction processes (20.2.11) continue until the coarsest grid $m = 1$ is reached. The correction $\Delta\overline{U}_{\alpha}^{m+1}$ is obtained from $\Delta\overline{U}_{\alpha}^{m}$ by prolongation factor I_r^{m+1} as we move from a coarse grid to a fine grid,

$$\Delta U_{\alpha}^{m+1} = I_r^{m+1}\Delta U_{\alpha r}^{m} \tag{20.2.12}$$

in which a few relaxation steps are to be made on a fine grid.

20.2.3 MULTIGRID SOLUTION PROCEDURE ON UNSTRUCTURED GRIDS

Although the multigrid methods can be applied to the structured grids for both FEM and FDM with nested grids in which all coarse grid nodes appear in the fine grid nodes, such nested structured grids may not be possible in general or impractical for complicated geometries. Multigrid methods for unstructured and unnested grids have been studied by many investigators in the past [Löhner and Morgan, 1987; Mavriplis and Jameson, 1990, among others]. To this end, it is convenient to use FEM or FVM in which any arbitrary geometries may easily be accommodated.

Let us consider an unstructured grid system as shown in Figure 20.2.4 in which the fine mesh A, B, C, D contains the coarse grid node 1 and the coarse grid element 1, 2, 3, 4 contains a fine grid element H, I, J, K. Our task is to determine how the flow variables, residuals, and corrections are to be transformed between fine and coarse grids. This can be achieved as follows:

(1) **Flow variables** A linear interpolation is used to transfer from the fine grid nodes A, B, C, D to the coarse grid node 1.
(2) **Residuals** (Restriction process) The fine grid residual at H is linearly distributed to the coarse grid nodes 1, 2, 3, 4 enclosing H. It is convenient to use the finite element trial functions for this purpose.
(3) **Corrections** (Prolongation process) The corrections from the coarse grid back to the fine grid E can be transferred using a simple linear interpolation of the

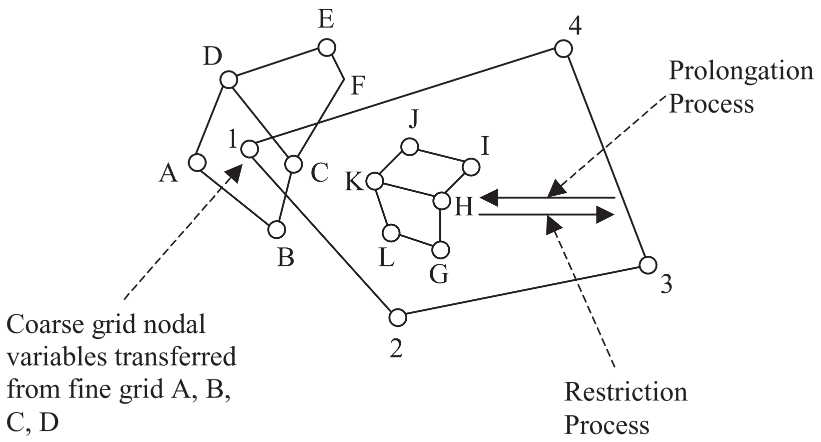

E
D
F
4
Prolongation Process
J
1
I
A
C
K
H
B
L
G
3
2
Coarse grid nodal variables transferred from fine grid A, B, C, D
Restriction Process

Figure 20.2.4 Grid transfers from coarse to fine and vice versa in unstructured quadrilateral multigrids.

coarse nodes 1, 2, 3, 4. An efficient strategy such as tree search algorithm may be employed to locate the coarse grid cell enclosing a particular fine grid node. In this algorithm, it requires information about the neighbors of each node or cell and a series of tests are carried out to determine if the coarse grid cell encloses the fine grid node.

As was indicated in Section 20.1 for domain decomposition, the parallel processing can be applied to multigrid methods also to obtain speedup in computer time. We shall discuss the subject of parallel processing in Section 20.3.

20.3 PARALLEL PROCESSING

20.3.1 GENERAL

Computational procedures in CFD in general as well as the adaptive mesh (Chapter 19), domain decomposition (Section 20.1), and multigrid methods (Section 20.2) discussed earlier will benefit from parallel processing, in which significant computational efficiency can be achieved. There are different forms of parallelism: multiple functional units, pipelining, vector processing, multiple vector pipelines, multiprocessing, and distributed computing.

In multiple functional units, we multiply the number of functional units such as adders and multipliers together. Here, the control units and the registers are shared by the functional units.

The concept of pipelining resembles an automobile assembly line. Let us assume that n number of operations takes s stages to complete in time t. The speedup factor S in this case can be given by the ratio, $S = nst/[(n + s - 1)t]$. It is seen that for a large number of operations, the speedup factor is approximately equal to the number of stages.

Vector computers are equipped with vector pipelines such as a pipeline floating point adder or multiplier. Also, vector pipe lines can be duplicated to take advantage of any fine grain parallelism available in loops.

A multiprocessor system is a set of several computers with several processing elements, each consisting of a CPU, a memory, an I/O subsystem, etc. These processing elements are connected to one another with some communication medium, either a bus or some multistage network. In a tightly coupled system, processors cooperate closely on the solution to a problem. A loosely coupled system consists of a number of independent and not necessarily identical processors that communicate with each other via a communication network. The multiprocessor computer architecture may be classified in terms of the sequence of instructions performed by the machine and the sequence of data manipulated by the instruction stream as follows:

(1) The single instruction-single data stream (SISD) architecture allows instructions to be executed sequentially but they may be overlapped in their execution stages (pipelining). Instructions are fetched from the memory in serial fashion and executed in a single processor.
(2) In single instruction-multiple data stream (SIMD) architecture multiple processing elements are all supervised by the same control unit. All processors

receive the same instructions broadcast from the control unit, but operate on different data sets from distinct data streams.

(3) With multiple instruction-multiple data stream (MIMD), each processor has its own control unit and the processors execute independently. The processors interact with each other either through shared memory or by using message passing to execute an application.

Distributed computing is a more general form of multiprocessing, linked by some local area network such as the parallel virtual machine (PVM) and the message passing interface (MPI). This system is cost effective for large applications with high volume of computation performed before more data is to be exchanged. In distributed multiprocessors, each processor has a private or local memory but there is no global shared memory in the system. The processors are connected using an interconnection network, and they communicate with each other only by passing messages over the network.

Multiprocessors rely on distributed memory in which processing nodes have access only to their local memory, and access to remote data is accomplished by request and reply messages. Numerous designs on how to interconnect the processing nodes and memory modules include Intel Paragon, N-Cube, and IBM's SP systems. As compared to shared memory systems, distributed (or message passing) systems can accommodate a larger number of computing nodes.

Although parallel processing systems, particularly those based on the message passing (or distributed memory) model, have led to several large-scale computing systems and specialized supercomputers, their use has been limited for very specialized applications. This is because message passing is difficult when a sequential version of the program as well as the message passing version is to be maintained. Thus, the new trend is that the programmers approach the two versions completely independently and that programming on a shared memory multiprocessor system (SMP) is considered easier. In shared memory paradigm, all processors or threads of computation share the same logical address space and access directly any part of the data structure in a parallel computation. A single address space enhances the programmability of a parallel machine by reducing the problems of data partitioning, migration, and local balancing. The shared memory also improves the ability of parallelizing compilers, standard operating systems, resource management, and incremental performance.

In the following sections, we discuss the development of parallel algorithms, parallel solution of linear systems on SIMD and MIMD machines, and applications of parallel processing in domain decomposition and multigrid methods, new trends in parallel processing, and some selected CFD problems.

20.3.2 DEVELOPMENT OF PARALLEL ALGORITHMS

SIMD and MIMD Structures

In numerical methods such as CFD, the basis for development of parallel algorithms is the evaluation of arithmetic expressions. The evaluation can be represented by graphs or trees. To this end, let us consider the problem of mapping a given arithmetic expression E into an equivalent expression $\tilde{E}$ that can be performed parallel on SIMD or MIMD computers by means of commutative, distributive, or associative laws of linear

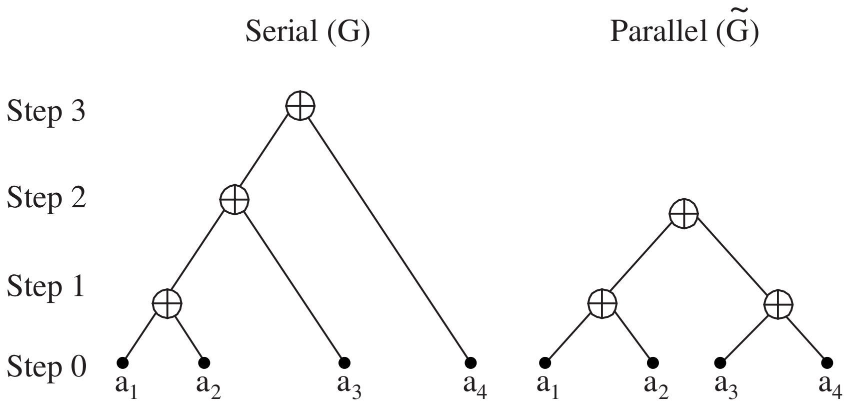


Figure 20.3.1 SIMD structure.

algebra. For example, two additions can be made parallel as follows:

$$E = a_4 + [a_3 + (a_2 + a_1)] \tag{20.3.1}$$

This can be transformed by the associativity of addition into

$$\tilde{E} = (a_4 + a_3) + (a_2 + a_1) \tag{20.3.2}$$

A typical SIMD structure is characterized by

$$E = a_1 + a_2 + a_3 + a_4 \tag{20.3.3}$$

By using the associative property of addition, we obtain

$$\tilde{E} = (a_1 + a_2) + (a_3 + a_4) \tag{20.3.4}$$

as schematically shown in Figure 20.3.1 in which G and $\tilde{G}$ denote the serial tree and parallel tree, respectively.

In MIMD structure, if we wish to compute

$$E = a_1 + a_2 \times a_3 + a_4 \tag{20.3.5}$$

it should be noted that the serial tree G is not a unique tree, and no tree height reduction can be obtained by applying the associative law. Instead, we apply the commutative property of addition with E being transformed into

$$\tilde{E} = (a_1 + a_4) + a_2 a_3 \tag{20.3.6}$$

with the tree height reduced by one step as shown in Figure 20.3.2.

The speedup of a parallel algorithm is given by

$$S_p = T_1 / T_p \tag{20.3.7}$$

where T_p is the execution time using p processors. The efficiency is defined by

$$E_p = S_p / p \tag{20.3.8}$$

Thus, for the case shown in Figure 20.3.2, we obtain $T_2 = 2$, $S_2 = T_1/T_2 = 3/2$, $E_2 = S_2/2 = 3/4$. In parallel processing, we must determine how many tree height reductions

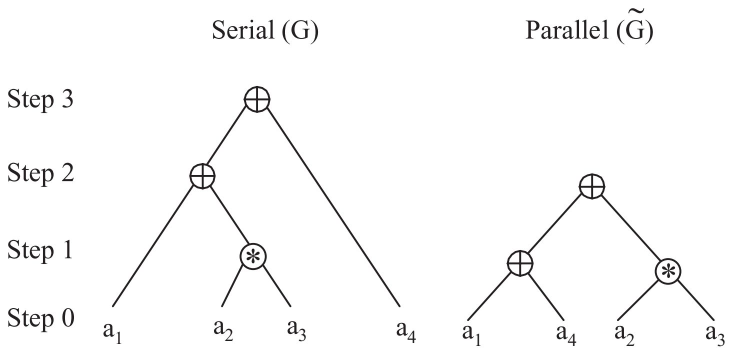


Figure 20.3.2 MIMD structure.

can be achieved for a given arithmetic expression and how many processors are needed for optimality.

Matrix-by-Vector Products in Parallel Processing

Matrix-by-vector multiplications are easy to implement on high-performance computers. Consider the matrix-by-vector product $y = Ax$. One of the most general schemes for storing matrices is the compressed sparse row (CSR) format. Here, the data structure consists of three arrays: a real array $A(1:nnz)$ to store the column positions of the elements row-wise, an integer array $JA(1:nnz)$ to store the column positions of the elements in the real array A, and finally, a pointer array $IA(1:n+1)$, the ith entry of which points to the beginning of the ith row in the arrays A and JA. Here, we note that each component of the resulting vector y can be computed independently as the dot product of the ith row of the matrix with the vector x. The algorithm for CSR format-dot product form may be given as follows:

```
1.  $Do\ i = 1, n$
2.  $k1 = ia(i)$
3.  $k2 = ia(i+1) - 1$
4.  $y(i) = dot\ product(a(k1:k2), x(ja(k1:k2)))$
5.  $EndDo$
```

Note that the outer loop can be performed in parallel on any parallel platform. On some shared memory machines, the synchronization of this outer loop is inexpensive and the performance of the above program can be effective. On distributed memory machines, the outer loop can be split in a number of steps to be executed on each processor. It is possible to assign a certain number of rows (often contiguous) to each processor and to also assign the component of each of the vectors similarly. When performing a matrix-by-vector product, interprocessor communication will be necessary to get the needed components of the vector x that do not reside in a given processor.

The indirect addressing involved in the second vector in the dot product is called a *gather* operation. The vector $x(ja(k1:k2))$ is first "*gathered*" from memory into a vector of contiguous elements. The dot product is then carried out as standard dot-product operation between two dense vectors, as illustrated in Figure 20.3.3.

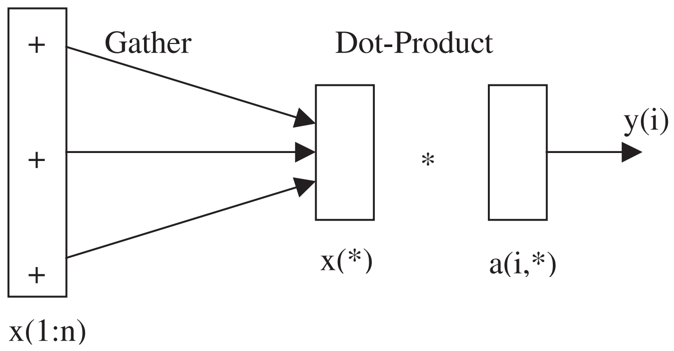


Figure 20.3.3 Row-oriented matrix-by-vector multiplication – Gather operation.

Let us now assume that the matrix is stored by compressed sparse column (CSC) storage format. The matrix-by-vector product can be performed by the following algorithm:

```
1. $y(1:n) = 0.0$
2. $Do\, i = 1, n$
3. $k1 = ia(i)$
4. $k2 = ia(i+1) - 1$
5. $y(ja(k1:k2)) = y(ja(k1:k2)) + x(j)^* a(k1:k2)$
6. $EndDo$
```

The above code initializes y to zero and then adds the vectors $x(j) \times a(1:n, j)$ for $j = 1, \ldots, n$ to it. It can also be used to compute the product of the transpose of a matrix by a vector, when the matrix is stored (row-wise) in the CSR format. Normally the vector $y(ja(k1:k2))$ is gathered and the vector update operation is performed in vector mode. Then the resulting vector is "*scattered*" back into the positions $ja(*)$ by what is called a *scatter* operation, as illustrated in Figure 20.3.4.

Notice that the inner loop involves writing back results of the right-hand side into memory positions that are determined by the indirect address function *ja*. To be correct, $y(ja(1))$ must be copied first, followed by $y(ja(2))$, etc. However, if it is known that the mapping $ja(i)$ is one-to-one, then the order of the assignments no longer matters. Since

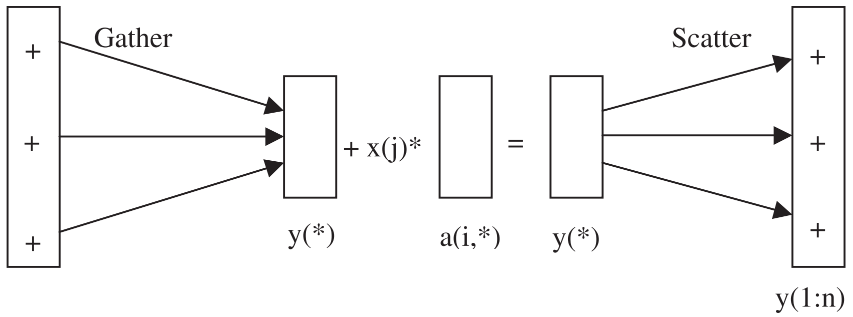


Figure 20.3.4 Column-oriented matrix-by-vector multiplication – Scatter operation.

compilers are not capable of deciding whether this is the case, a compiler directive from the user is necessary for the scatter to be invoked.

20.3.3 PARALLEL PROCESSING WITH DOMAIN DECOMPOSITION AND MULTIGRID METHODS

Although it is difficult to characterize multiprocessors in a simple manner, we may assume that they are individual processors and memory modules that are interconnected in some way. This interconnection can occur in a number of ways, but in general, processor memory modules communicate with one another directly or through a common shared memory. The processing unit in the model can be a simple bit processor, a scalar processor, or a vector processor. The memory unit in the module can be a few registers or a cache memory. Because of nonlinearity in fluid mechanics, it is important that the interaction between the computer modules in a multiprocessing system be controlled by a single operating system.

There are two forms of multiprocessors: the loosely coupled or distributed memory multiprocessors and the tightly coupled or shared memory multiprocessors. In a loosely coupled system, each computer module has a relatively large local memory where it accesses most of the instructions and data. Because there is no shared memory, processes executing on different computer modules communicate by exchanging messages through an interconnection network. In fact, the communication topology of this interconnection network is the crucial factor of these systems. Thus, loosely coupled systems are usually efficient when the interaction between computational tasks is minimal.

Tightly coupled multiprocessor systems communicate through a globally shared memory. Hence, the rate at which data can communicate from one computer module to the other is of the order of the bandwidth of the memory. Because of the complete connectivity between the computer modules and memory, the performance may tend to degrade due to memory contentions.

Ideal numerical models for multiprocessors are those that can be broken down into algebraic tasks, each of which can be executed independently on a computer module without ever having to obtain or pass data between the modules during the course of the execution. This framework allows a mechanism for analyzing the movement of data within a multiprocessing system. The basic idea is to regard the computational tasks being performed by the individual computer modules as numerical solutions of individual boundary value problems. In this way numerical data being obtained or transmitted between computer modules are the initial and boundary data of the differential equations. The solution of the overall mathematical model is then provided by “piecing” together each of the subproblems.

For the domain decomposition methods presented in Section 20.1, the domain $\Omega(t)$ is expressed as a union of subdomains (such as in Figure 20.1.1)

$$\Omega(t) = \bigcup_{j=1}^{k(t)} \Omega_j(t) \tag{20.3.9}$$

Each processor then assumes the task of solving one or more of the partial differential equations over a prescribed time interval Δt. At the end of this time interval, a new substructuring of the domain is performed:

$$\Omega(t+\Delta t)=\bigcup_{j=1}^{k(t+\Delta t)} \Omega_j(t,\Delta t) \tag{20.3.10}$$

and the process is repeated. The numerical mathematical relationship between the computed subdomain solutions and the solution of the global problem is delicate and is a function of the partial differential equation being solved. However, it is precisely this relationship that determines the efficiency of the computation on a multiprocessing system.

New Trends in Parallel Processing

It appears that the use of small clusters of SMP systems, often interconnected to address the needs of complex problems requiring the use of large numbers of processing nodes, is gaining popularity [Kavi, 1999]. Even when working with networked resources, programmers are relying on messaging standards such as MPI and PVM or relying on systems software to automatically generate message passing code from user-defined shared memory programs. The reliance on software support to provide a shared memory programming model (i.e., distributed shared memory systems) can be viewed as a logical evolution in parallel processing. Distributed shared memory (DSM) systems aim to unify parallel processing systems that rely on message passing with the shared memory systems. The use of distributed memory systems as shared memory systems addresses the major limitation of SMPs, namely scalability.

The growing interest in multithreading programming and the availability of systems supporting multithreading (Pthreads, NT-threads, Linux threads, Java) further emphasizes the trend toward shared memory programming model [Nichol, Buttlar, and Farrell, 1996]. The so-called OpenMP Fortran is designed for the development of portable parallel programs on shared memory parallel computer systems. One effect of the OpenMP standard will be to increase the shift of complex scientific and engineering software development from the supercomputer world to high-end desktop workstations.

Distributed shared memory systems (DSM) attempt to unify the message passing and shared memory programming models. Since DSMs span both physically shared and physically distributed memory systems, DSMs are also concerned with the interconnection networks that provide the data to the requesting processor in an efficient and timely fashion. Both the bandwidth (amount of data that can be supplied in a unit time) and latency (the time it takes to receive the first piece of requested data from the time the request is issued) are important to the design of DSM. It should be noted that because of the generally longer latencies encountered in large-scale DSMs, multithreading has received considerable attention in order to tolerate (or mask) memory latencies.

The management of large logical memory space involves moving data dynamically across the memory layers of a distributed system. This includes the mapping of the user data to the various memory modules. The data may be uniquely mapped to a physical address as done in cache coherent systems, or replicating the data to several physical addresses as done in reflective memory systems and, to some extent, in cache-only systems. Even in uniquely mapped systems, data may be replicated in lower levels of memories (i.e., cache). Replication, in turn, requires means for maintaining consistency. Directories are often the key to tracking the multiple copies and play a key role in coherency of replicated data. Hardware or software can maintain the coherency.

The granularity of the data that is shared and moved across memory hierarchies is another design consideration. The granularity can be based on objects without semantic meaning, based purely on a sequence of bytes (e.g., a memory word, a cache block, or a page) or it can be based on objects with semantic basis (e.g., variable, data structures or objects in the sense of object-oriented programming model). Hardware solutions often use finer grained objects (often without semantic meaning) while software implementations rely on coarser grained objects.

Synchronization and coordination among concurrent computations (i.e., processes or threads) in shared memory programming rely on the use of mutual exclusions, critical sections, and barriers. Implementation of the mechanisms needed for mutual exclusion (often based on the use of locks) and barriers explores the design space spanning hardware instructions, spin locks, tree barriers, etc. These solutions explore trade-offs between the performance (in terms of latency, serialization, network traffic) and scalability.

In addition to the issues mentioned above, resource management – particularly in terms of minimizing data migration, thread migration, and messages exchanged – are all significant in achieving cost-performance ratios in making DSM systems as contenders.

In summary, threads are used in multitasking to better utilize processor capabilities, resulting in better throughput and response time. For example, we can assign a separate task/thread to process user command inputs, while other tasks/threads can do the actual processing. This eliminates the need for "freezing" keyboard while processing a previous request. Thus, threads represent "concurrency." The complex flow without threads and simple flow with threads are illustrated in Figure 20.3.5a and Figure 20.3.5b, respectively. It is seen that a complex program can be viewed as concurrent activities and the flow of control becomes cleaner (Figure 20.3.5b). Without threads, we can still simulate the concurrent activities by interleaving the execution. Such structures cannot utilize multiple CPUs or kernel threads. With threads, we can use multiple processors or kernel threads. If multiple processors are available, we can increase the performance by using

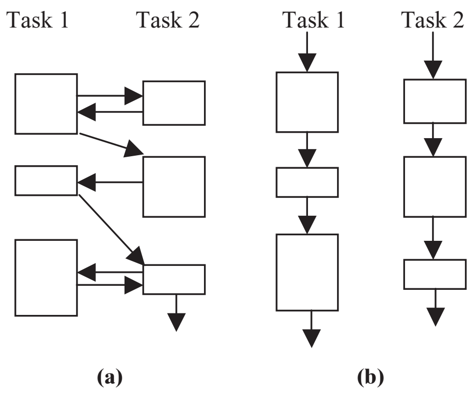


Figure 20.3.5 Threads and concurrency. **(a)** Complex flow without threads or concurrency. **(b)** Simple flow with threads or concurrency.

multiple concurrent activities. Multitasking or other concurrent programming methods utilize the multiple processing units. Multithreaded programs can be executed either on a single processor system or on an SMP with minimum changes. This is in contrast to traditional (old) parallel programming which requires careful and tedious changes to the program structure to utilize the multiple-processing unit. As each workstation is becoming more powerful and cheaper, the trend has been to use a network of such systems instead of supercomputers or massively parallel systems.

20.3.4 LOAD BALANCING

An important consideration in CFD is the problem of distributing the mesh across the memory of the machine at runtime so that the calculated load is evenly balanced and the amount of interprocessor communication is minimized. Load balancing is difficult in large distributed systems. Algorithms must minimize both load balance and communication overhead of the application. These algorithms should balance the load with as little overhead as possible, and they should be scalable.

We consider a parallel system as with P processors as a graph $H = (U, F)$ with nodes $U = \{0, \ldots, P-1\}$ and edges $F \subseteq U \times U$. Similarly, a parallel application is modeled as graph $G = (V, E, \rho, \sigma)$ with nodes $V = \{0, \ldots, N-1\}$, edges $E \subseteq V \times V$, node weights $\rho : V \to \tilde{R}$, and edge weights $\sigma : E \to \tilde{R}$.

We may view the load balancing as a graph embedding problem. Our task is to find a mapping $M : G \to H$ of the application graph to the processor graph minimizing a cost function. The processor graph H is usually static (constant during the runtime), whereas the parallel application graph G may be static or dynamic, that is, the computational load of the application may or may not change during runtime.

The Static Load Balancing

In the static load balancing, neither the structure nor the weights of the application graph G change during runtime. It is assumed that G is completely known prior to the start of the application such as in nonadaptive methods for numerical simulation. The static load balancing problem calculates a good mapping of the application graph $G = (V, E)$ onto the processor graph $H = (U, F)$.

Cost functions determining the quality of a mapping are its *load, dilation, and congestion*. The load of a mapping M is the maximum number of nodes from G assigned to any single node of H. The dilation is the maximum distance of any route of a single edge from G in H. The congestion is the maximum number of edges from G that must be routed via any single edge in H. The load determines the balancing quality of the mapping. It should be kept as low as possible to avoid idle times of the processor. The dilation of and edge of G determine the slowdown of a communication on this edge due to routing latency in H. The goal is to find a mapping function M which minimizes all three measures – load, dilation, and congestion [Leighton, 1992].

A graph is split into as many as there are numbers of processors such that as few as possible edges are external. This can be done by recursively bisecting the graph into two pieces. There are efficient solution heuristics which approximate the best value in terms of numbers of external edges. Some of the examples are (1) global methods partitioning the nodes into two subsets of equal size [Jones and Plassmann, 1994; Kaddoura, Ou, and Ranka, 1995]; (2) local methods where local heuristics determine equally sized sets of nodes which can be exchanged between parts such that the size of the cut decreases [Kerninghan and Lin, 1970; Fiduccia and Mattheyses, 1982; Hendrickson and Leland, 1993]; (3) multilevel hybrid methods in which a large graph is shrunk to a smaller one with similar characteristics, efficiently partitioned, and extrapolated to the original graph [Karypis and Kumar, 1995; Hendrickson and Leland, 1993].

Dynamic Load Balancing

The application graph $G = (V, E, \rho, \sigma)$ of problems in this class is dynamic; that is, nodes and edges are generated or deleted during runtime. Here, operations are carried out in phases. Changes to G do not occur at arbitrary, nonpredictable times but in synchronized manner. The mesh is usually refined based on error estimates of the current solution [Bornemann, Erdmann, and Kornhuber, 1993]. In general, we split the task of load balancing into two steps. First, we calculate how much load is to be shifted between processors, and second we determine which load is to be moved [Diekmann, Meyer, and Monien, 1997; Lüling and Monien, 1993].

Lin and Keller [1987] proposed a gradient model in which they assign a status of high, medium, or low to processors depending on their load. The algorithm then pushes the load from high to low. Lüling and Monien [1992] make processors balance their load with a fixed set of neighbors if the load difference between them increases above a certain threshold. Rudolph, Slivkin-Allouf, and Upfal [1991] showed that if processor j initiates a balancing action with a randomly chosen other processor with probability $(c\,/load\,j)$, then the expected load of j is at most c times the average load plus a constant.

The first step of the load balancing is to calculate how much load has to be transferred across each edge of H in order to achieve a globally balanced system. There are many approaches to this task: (1) Token distribution. This is the synchronized setting of the re-embedding problem in which a number of independent tokens on a network of processors are evenly distributed [Meyer et al., 1996]. (2) Random matchings. Ghosh et al. [1995] show that the load deviation halves in a minimal number of steps if a random matching of H's edges is chosen and some load is sent via these edges when the corresponding processors are not balanced. However, this approach is impractical in general situations. (3) Diffusion. A simple diffusive distributed load balancing strategy in which each processor balances its load with all its neighbors in each round was suggested by Cybenko [1989] and Boillat [1990]. These rounds are iterated until the load is completely balanced.

In addition to determining how much load is to be transferred, it is also important to choose load items which can be migrated in order to fulfill the flow requirements. For example, global iterative methods for solving linear systems such as multigrid or conjugate gradient computations can be parallelized by choosing load items so that the communication demands are minimized. Here, we must take into account the total length of subdomain boundaries, communication characteristics of the parallel system, etc. An example of recursive graph bisection for airfoils as demonstrated by Diekmann et al. [1997] is shown in Figure 20.3.6a. An aspect ratio optimization may be applied as shown in Figure 20.3.6b.

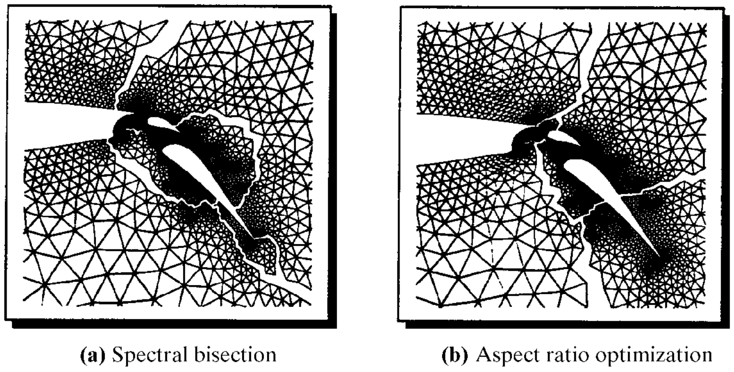


Figure 20.3.6 Dynamic load balancing of airfoil grid generation [Diekmann et al., 1997].

20.4 EXAMPLE PROBLEMS

In this section, two examples of parallel processing with domain decomposition are presented. Solutions of Poisson equation and Navier-Stokes system of equations will be discussed.

20.4.1 SOLUTION OF POISSON EQUATION WITH DOMAIN DECOMPOSITION PARALLEL PROCESSING

Domain decompositions methods are used effectively in parallel processing. Subdomains may be nonoverlapping, or overlapping. First, let us consider a nonoverlapping case (Figure 20.4.1a) and construct the matrix equations of the form,

$$Lu = \begin{bmatrix} K_{11} & 0 & K_{13} \\ 0 & K_{22} & K_{23} \\ K_{31} & K_{32} & K_{33} \end{bmatrix} \begin{bmatrix} u_1 \\ u_2 \\ u_3 \end{bmatrix} = \begin{bmatrix} f_1 \\ f_2 \\ f_3 \end{bmatrix} = f \tag{20.4.1}$$

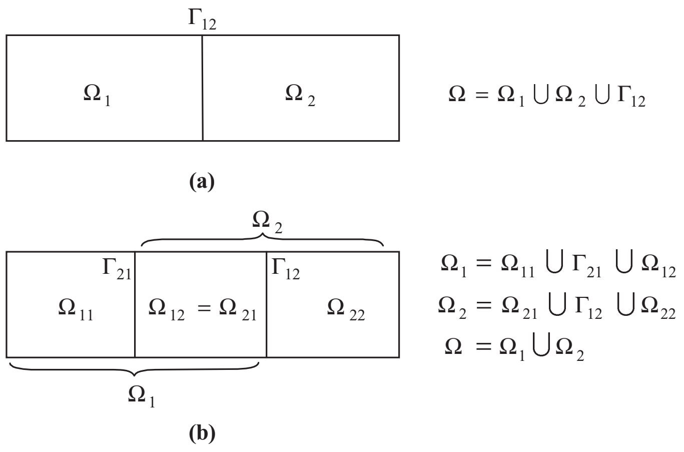

Γ_{12}
Ω_1
Ω_2
$\Omega = \Omega_1 \cup \Omega_2 \cup \Gamma_{12}$
(a)
Ω_2
Γ_{21}
Γ_{12}
Ω_{11}
$\Omega_{12} = \Omega_{21}$
Ω_{22}
$\Omega_1 = \Omega_{11} \cup \Gamma_{21} \cup \Omega_{12}$
$\Omega_2 = \Omega_{21} \cup \Gamma_{12} \cup \Omega_{22}$
$\Omega = \Omega_1 \cup \Omega_2$
Ω_1
(b)

Figure 20.4.1 Domain decomposition. **(a)** Nonoverlapping subdomains. **(b)** Overlapping subdomains.

which is similar to (20.1.2). Here, the first two rows indicate subdomains Ω_1 and Ω_2, with the third row representing the boundary interface Γ_{12}. The subdomain variables u_1 and u_2 are calculated as

$$\begin{aligned} u_1 &= K_{11}^{-1}(f_1 - K_{13}u_3) \\ u_2 &= K_{22}^{-1}(f_2 - K_{23}u_3) \end{aligned} \tag{20.4.2}$$

where the boundary interface variables u_3 are determined from

$$\left(K_{33} - K_{31}K_{11}^{-1}K_{13} - K_{32}K_{22}^{-1}K_{23}\right)u_3 = f_3 - K_{31}K_{11}^{-1}f_1 - K_{32}K_{22}^{-1}f_2 \tag{20.4.3}$$

The above unknowns can be solved using two MIMD parallel processors. Here, we may utilize the preconditioning operator as described in Section 20.1.1.

The two subdomains used in the above example may be overlapped as shown in Figure 20.4.1b. In this case, the matrix equations take the form

$$Lu = \begin{bmatrix} K_{11} & K_{12} & 0 & 0 & 0 \\ K_{21} & K_{22} & K_{23} & 0 & 0 \\ 0 & K_{32} & K_{33} & K_{34} & 0 \\ 0 & 0 & K_{43} & K_{44} & K_{45} \\ 0 & 0 & 0 & K_{54} & K_{55} \end{bmatrix} \begin{bmatrix} u_1 \\ u_2 \\ u_3 \\ u_4 \\ u_5 \end{bmatrix} = \begin{bmatrix} f_1 \\ f_2 \\ f_3 \\ f_4 \\ f_5 \end{bmatrix} = f \tag{20.4.4}$$

which is partitioned into two systems, Ω_{11} and Ω_{22} such that

$$L_1 v_1 = \begin{bmatrix} K_{11} & K_{12} & 0 \\ K_{21} & K_{22} & K_{23} \\ 0 & K_{32} & K_{33} \end{bmatrix} \begin{bmatrix} v_{11} \\ v_{12} \\ v_{13} \end{bmatrix} = \begin{bmatrix} g_{11} \\ g_{12} \\ g_{13} \end{bmatrix} = F_1 \tag{20.4.5}$$

$$L_2 v_2 = \begin{bmatrix} K_{33} & K_{34} & 0 \\ K_{43} & K_{44} & K_{45} \\ 0 & K_{54} & K_{55} \end{bmatrix} \begin{bmatrix} u_3 \\ u_4 \\ u_5 \end{bmatrix} = \begin{bmatrix} f_3 \\ f_4 \\ f_5 \end{bmatrix} = F_2 \tag{20.4.6}$$

with

$$F_1 = \begin{bmatrix} f_1 \\ f_2 \\ f_3 \end{bmatrix} - \begin{bmatrix} 0 & 0 & 0 \\ 0 & 0 & 0 \\ 0 & K_{34} & 0 \end{bmatrix} \begin{bmatrix} u_3 \\ u_4 \\ u_5 \end{bmatrix} = F_1' - G_2 v_2 \tag{20.4.7}$$

$$F_2 = \begin{bmatrix} f_3 \\ f_4 \\ f_5 \end{bmatrix} - \begin{bmatrix} 0 & K_{32} & 0 \\ 0 & 0 & 0 \\ 0 & 0 & 0 \end{bmatrix} \begin{bmatrix} u_1 \\ u_2 \\ u_3 \end{bmatrix} = F_2'' - G_1 v_1 \tag{20.4.8}$$

The above process results in the system of equations in the form

$$\begin{bmatrix} L_1 & G_2 \\ G_1 & L_2 \end{bmatrix} \begin{bmatrix} v_1 \\ v_2 \end{bmatrix} = \begin{bmatrix} F_1 \\ F_2 \end{bmatrix} \tag{20.4.9}$$

This can be solved using the block Jacobi scheme:

$$\begin{bmatrix} L_1 & 0 \\ 0 & L_2 \end{bmatrix} \begin{bmatrix} v_1 \\ v_2 \end{bmatrix}^{k+1} = \begin{bmatrix} F_1 \\ F_2 \end{bmatrix} - \begin{bmatrix} 0 & G_2 \\ G_1 & 0 \end{bmatrix} \begin{bmatrix} v_1 \\ v_2 \end{bmatrix}^{k} \tag{20.4.10}$$

This system suggests that we can utilize two processors on a MIMD machine, forming a global and inner parallelism of the algorithm.

20.4.2 SOLUTION OF NAVIER-STOKES SYSTEM OF EQUATIONS WITH MULTITHREADING

Multithreaded programming is utilized to take advantage of multiple computational elements on the host computer [Schunk et al., 1999]. Typically, a multithreaded process will spawn multiple threads which are allocated by the operating system to the available computational elements (or processors) within the system. If more than one processor is available, the threads may execute in parallel, resulting in a significant reduction in execution time. If more threads are spawned than available processors, the threads appear to execute concurrently as the operating system decides which threads execute while the others wait. One unique advantage of multithreaded programming on shared memory multiprocessor systems is the ability to share global memory. This alleviates the need for data exchange or message passing between threads as all global memory allocated by the parent process is available to each thread. However, precautions must be taken to prevent deadlock or race conditions resulting from multiple threads trying to simultaneously write to the same data.

Threads are implemented by linking an application to a shared library and making calls to the routines within that library. Two popular implementations are widely used: the Pthreads library [Nichol et al., 1996] (and its derivatives) that are available on most Unix operating systems and the NTthreads library that is available under Windows NT. There are differences between the two implementations, but applications can be ported from one to the other with moderate ease and many of the basic functions are similar albeit with different names and syntax.

Domain decomposition methods (Section 20.3.1) can be used in conjunction with multithreaded programming to create an efficient parallel application. The subdomains resulting from the decomposition provide a convenient division of labor for the processing elements within the host computer. The additive Schwarz domain decomposition method discussed in Section 20.1.2 is utilized. The method is illustrated below (Figure 20.4.2.1) for a two-dimensional square mesh that is decomposed into four subdomains. The nodes belonging to each of the four subdomains are denoted with geometric symbols while boundary nodes are identified with bold crosses. The desire is to solve for each node implicitly within a single subdomain. For nodes on the edge of each subdomain, this is accomplished by treating the adjacent node in the neighboring subdomain as a boundary. The overlapping of neighboring nodes between subdomains is illustrated in Figure 20.4.2.2. Higher degrees of overlapping, which may improve convergence at the expense of computation time, are also used.

In a parallel application, load balancing between processors is critical to achieving optimum performance. Ideally, if a domain could be decomposed into regions requiring an identical amount of computation, it would be a simple matter to divide the problem between processing elements as shown in Figure 20.4.2.3 for four threads executing on an equal number of processors.

Unfortunately, in a “real world” application the domain may not be decomposed such that the computation for each processor is balanced, resulting in lost efficiency. If the execution time required for each subdomain is not identical, the CPUs will become idle for portions of time as shown in Figure 20.4.2.4.

One approach to load balancing, as implemented in this application, is to decompose the domain into more subdomains than available processors and use threads to perform

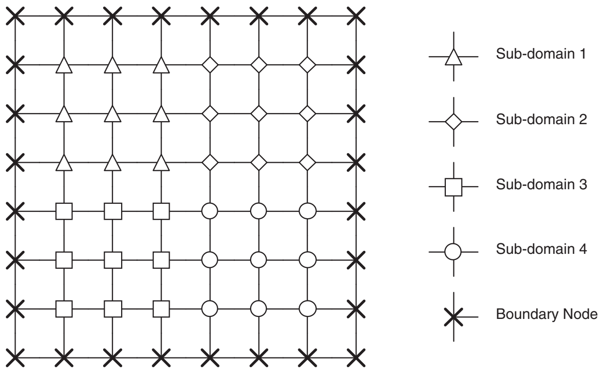


Figure 20.4.2.1 Multiple subdomains.

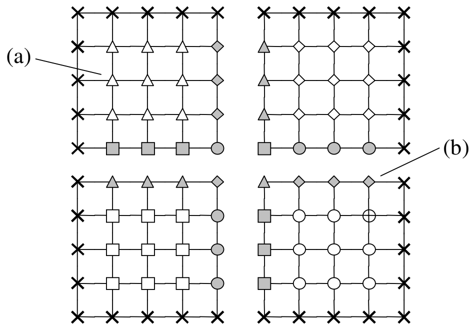


Figure 20.4.2.2 Domain decomposition. **(a)** Nodes shaded in white are solved implicitly within each subdomain. **(b)** Nodes from neighboring subdomains (shaded) are treated as boundary nodes and allowed to lag one time step.

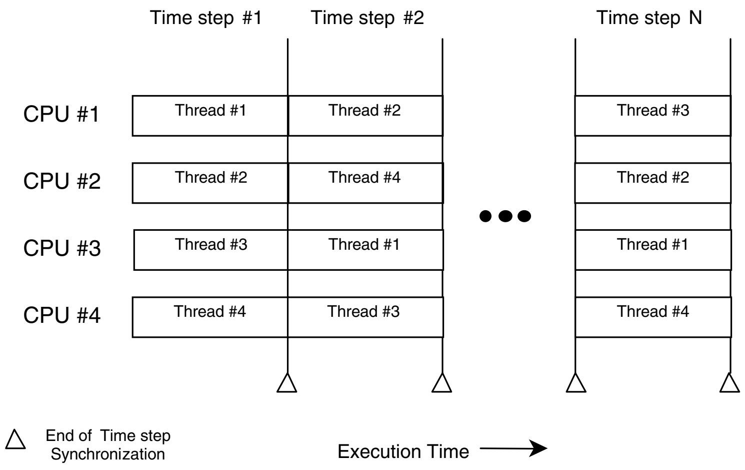


Figure 20.4.2.3 Ideal load balancing.

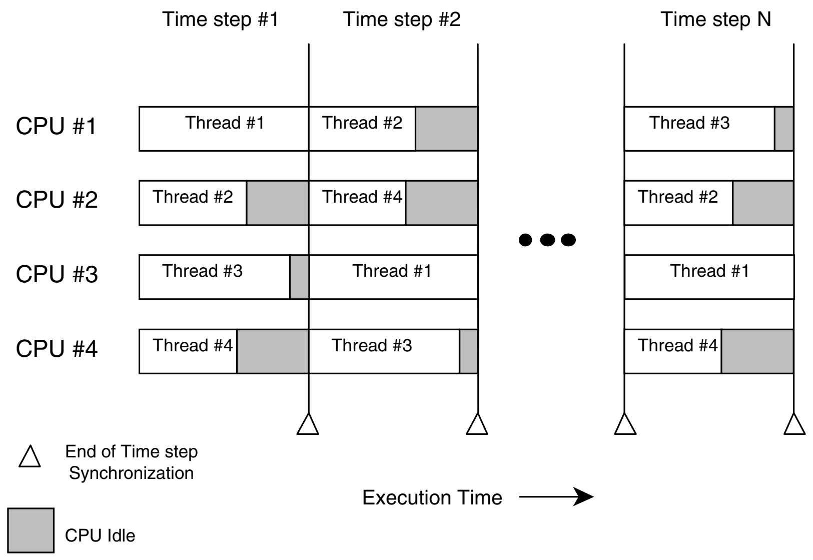


Figure 20.4.2.4 "Real world" load balancing.

the computations within each block. The finer granularity permits a more even distribution of work among the available processing elements, as shown in Figure 20.4.2.5.

In this approach, the number of threads spawned is equal to the number of available processors, with each thread marching through the available subdomains (which preferably number at least two times the number of processors), solving one at a time in an "assembly-line" fashion. A stack is employed where each thread pops the next subdomain to be solved off of the top of the stack. Mutual exclusion locks are employed to protect the stack pointer in the event two or more threads access the stack simultaneously. Each thread remains busy until the number of subdomains is exhausted. If the number of subdomains is large enough, the degree of parallelism will be high, although decomposing a problem into too many subdomains may adversely affect convergence. This approach is illustrated in Figures 20.4.2.6 and 20.4.2.7.

We revisit the triple shock wave boundary layer interaction problem presented in Section 6.7.5. In this example, the flowfied-dependent variation (FDV) method (Section 6.5) FDM is used. The FDV solver is based upon the Generalized Minimum Residual (GMRES). The application is coded to be multithreaded to take advantage of parallelism in the host computer. The number of threads is specified at run time and is based on the expected number of available CPUs. The results for three different architectures are provided in Table 20.4.1. Typical utilizations (defined as CPU time/elapsed time) range from 180% to 380% for two to eight threads. It should be noted that both the number of threads and number of processors impose theoretical limits on the maximum performance gain. Obviously, the normalized performance increase cannot exceed the number of threads and, aside from tertiary performance issues (such as on processor cache), nor can the normalized performance increase exceed the number of processors. For the coarse mesh model, actual speed increases range from 1.77 to 3.44 for 2 to 4 processors. The results are encouraging when considering the CPU contention between multiple users on the host machines. For the coarse mesh model on a dual processor Pentium II workstation (with no other users) a CPU utilization of 196% is observed

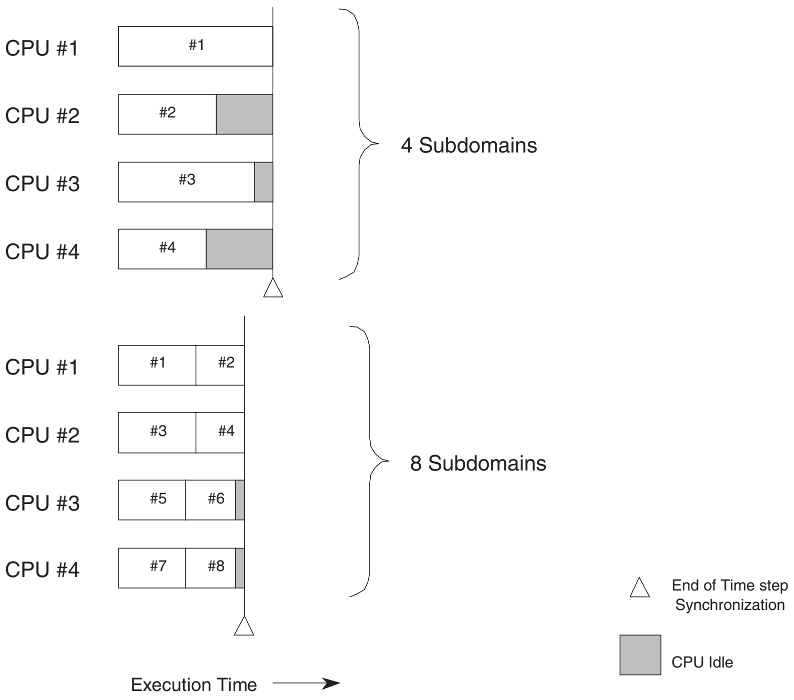


Figure 20.4.2.5 Domain decomposition improves parallelism.

with a real-time speedup of 1.92. The four-processor machine did exhibit a significant amount of overhead when moving beyond a single thread. Utilizing four threads for the fine mesh model resulted in CPU utilizations of 357% and 370% for a domain decomposed into 27 and 64 regions, respectively. The fine mesh model was not run with a single processor or thread so that no relative speedup data are available. The CPU utilization is encouraging considering the high CPU contention on the twelve-processor machine.

Density contours for the inviscid shock interaction (x-z plane, as viewed from above the wind tunnel model) are shown in Figure 20.4.2.8. The 15° fins produce inviscid shocks that are predicted to intersect and reflect approximately 92 mm from the ramp entrance. The reflected shock does not intersect with the exit corner of the ramp as expected. Two

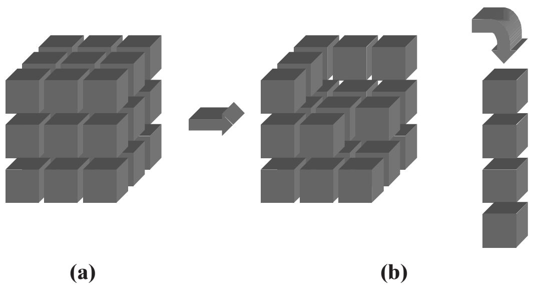


Figure 20.4.2.6 Decompose the domain and push onto stack. **(a)** Decompose the domain. **(b)** Push each subdomain onto a software stack.

Table 20.4.1 Computational Performance Summary

Threads	Grid	Decomposition	CPU Time (hours)	Elapsed Time (hours)	CPU Utilization	Speedup	Processor	Number of Processors
1	$55 \times 41 \times 31$	$4 \times 4 \times 4$	5.05	5.05	100%	1.00	Pentium II	2
2	$55 \times 41 \times 31$	$4 \times 4 \times 4$	5.13	2.62	196%	1.93	Pentium II	2
1	$55 \times 41 \times 31$	$4 \times 4 \times 4$	4.69	4.72	99%	1.00	Alpha	4
2	$55 \times 41 \times 31$	$4 \times 4 \times 4$	5.19	2.66	195%	1.77	Alpha	4
4	$55 \times 41 \times 31$	$4 \times 4 \times 4$	5.30	1.42	373%	3.32	Alpha	4
6	$55 \times 41 \times 31$	$4 \times 4 \times 4$	5.30	1.40	378%	3.36	Alpha	4
8	$55 \times 41 \times 31$	$4 \times 4 \times 4$	5.16	1.37	377%	3.44	Alpha	4
4	$109 \times 81 \times 61$	$3 \times 3 \times 3$	37.40	10.47	357%	NA	R10000	12
4	$109 \times 81 \times 61$	$4 \times 4 \times 4$	52.76	14.25	370%	NA	R10000	12

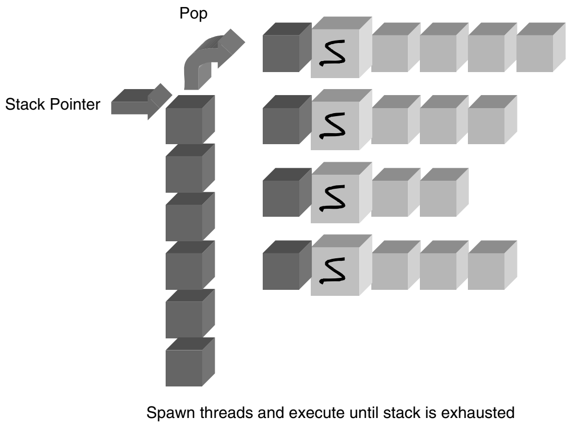


Figure 20.4.2.7 Allow threads to process each subdomain.

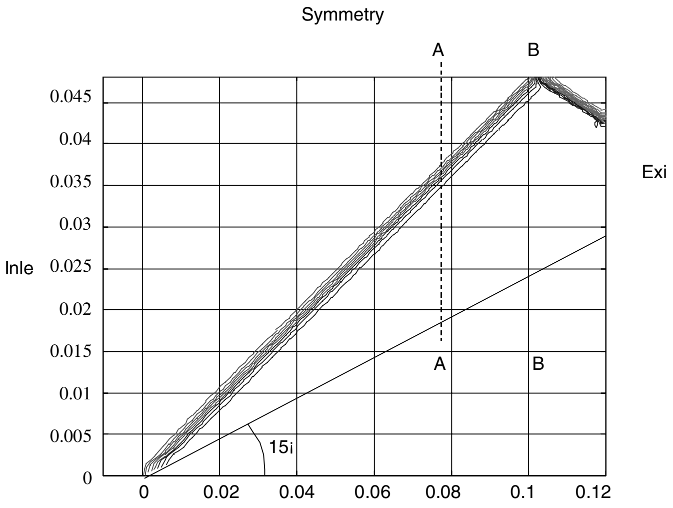


Figure 20.4.2.8 Density contours for *X*-*Z* cross section (top), slip boundary.

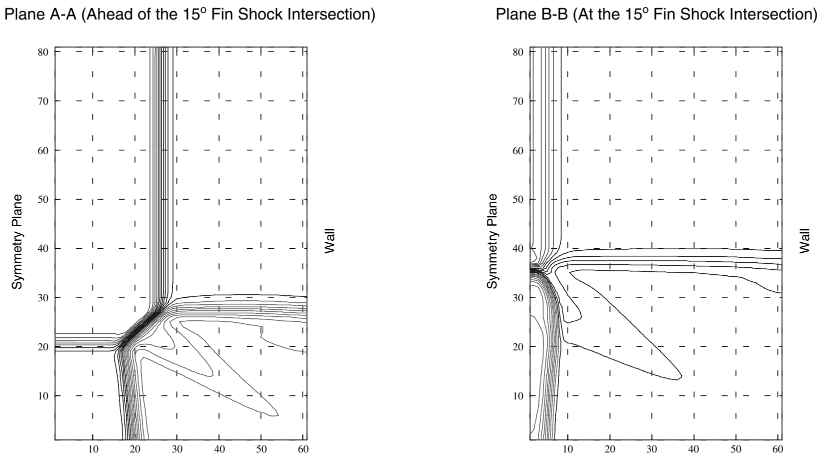


Figure 20.4.2.9 Density contours for Y-Z cross section, slip boundary.

cross sections, located at 67 mm and 92 mm, respectively, from the entrance are noted on the plot.

Density contours for the flow in x-y planes located 67 mm (upstream of the inviscid shock intersection) and 92 mm (coincident with the inviscid shock intersection) from the combined fin/ramp entrance are shown in Figure 20.4.2.9. It appears that the upstream predictions correlate well with the experimental images. The inviscid ramp and fin shocks, as well as the corner reflection, are easily discernible in the upstream figure (see left). Interestingly, it appears that the triangular-shaped slip lines are present in the numerical results of the upstream plane. Since the sliplines divide constant pressure regions with differing velocities, this feature is not visible in the static pressure plots. As in the experimental imagery, the inviscid fin shocks merge together in the symmetry plane at the point where the inviscid shocks intersect (see right). No curvature of the inviscid fin shock intersection is observed in the numerical predictions. The reflection of the corner shock about the symmetry plane is observed, but the ramp embedded shock is lower relative to the height of the fin than in the experimental results.

20.5 SUMMARY

Three of the most important computing techniques have been discussed: domain decomposition, multigrids, and parallel processing. For large geometrical configurations, domain decomposition provides efficiency in data managements. The number of resulting algebraic equations can still be very large, and the multigrid method of solutions of the large algebraic system of equations is considered a most effective approach.

The trends in parallel processing have been leaning toward the use of small clusters of Symmetric Multiprocessors (SMP), often interconnected to address the needs of complex problems requiring a large number of processing nodes. In the past, programming based on message passing paradigms on massively parallel computers or specialized supercomputers has been used. These systems are becoming less popular (or available) and distributed networks of SMP clusters are becoming the preferred choice for engineering. The growing interest in multithreaded programming and the availability of systems supporting multithreading can be seen as evidence of the departure from the use of supercomputers.

Many engineering applications rely on adaptive grid techniques that require dynamic load balancing of the threads/processors. In this vein, it is necessary to develop new scheduling and load balancing approaches for adaptive grid applications on shared memory systems using thread migration. The shared memory model presents opportunities for exploiting finer-grained threads, faster thread migration, and load distribution. Thus, the advanced research in parallel processing remains a great challenge in the future.

REFERENCES

Boillat, J. E. [1990]. Load balancing and Poisson equation in a graph. *Currency Practice and Experience*, 2, 4, 289–313.

Bornemann, F., Erdmann, B., and Kornhuber, R. [1993]. Adaptive multilevel methods in three space dimensions. *Int. J. Num. Meth. Eng.*, 36, 3187–3203.

Brandt, A. [1972]. Multilevel adaptive technique (MLAT) for fast numerical solutions to boundary value problems. *Lecture Notes in Physics 18*, Berlin: Springer-Verlag, 82–89.

——— [1977]. Multilevel adaptive solutions to boundary value problems. *Math. Comp.* 31, 333–90.

——— [1992]. On multigrid solution of high Reynolds incompressible entering flows. *J. Com. Phys.*, 1101, 151–64.

Cybenko, G. [1989]. Load balancing for distributed memory multiprocessors. *J. Par. Distr. Comp.*, 7, 279–301.

Diekmann, R., Meyer, D., and Monien, B. [1997]. Parallel decomposition of unstructured FEM-meshes. *Proc. IRREGULAR 95*, Springer LNCS, 199–215.

Fiduccia, C. M. and Mattheyses, R. M. [1982]. A linear-time heuristic for improving network partitions. Proc. 19th IEEE Design Automation Conference, 175–81.

Ghosh, B., Leighton, F. T., Maggs, B. M., and Muthukrishnan, S. [1995]. Tight analyses of two local load balancing algorithms. Proc. 27th ACM Symp. in Theory of Computing (STOC, 95), 548–58.

Glowinski, R. and Wheeler, M. F. [1987]. Domain decomposition and mixed finite element methods for elliptic problems. In R. Glowinski et al., (ed.). *Domain Decomposition Methods for Partial Differential Equations*. SIAM Publications, 144–72.

Hendrickson, B. and Leland, R. [1993]. A multilevel algorithm for partitioning graphs. Technical Report SAND93-1301, Sandia National Lab., Sandia.

Jones, M. T. and Plassmann, P. E. [1994]. Parallel algorithms for the adaptive refinement and partitioning of unstructured meshes. Proc. Scalable High Performance Computing Conf., IEEE Computing Conf., IEEE Computer Society Press, 478–85.

Kaddoura, M., Ou, C. W., and Ranka, S. [1995]. Mapping unstructured computational graphs for adaptive and nonuniform computational environments. *IEEE Par. and Dir. Technology*.

Karypis, G. and and Kumar, V. [1995]. A fast and high quality multilevel scheme for partitioning irregular graphs. Tech. Report. 95-035, CD-Dept, University of Minnesota.

Kavi, K. M. [1999]. Multithreaded system implementations. *J. Microcomp. App.*, 18, 2.

Kerninghan, B. W. and Lin, S. [1970]. An effective heuristic procedure for partitioning graphs. The Bell Systems Tech. J., 291–308.

Leighton, F. T. [1992]. *Introduction to Parallel Algorithms and Architectures*. Morgan Kaufmann Publishers.

Lin, F. C. H. and Keller, R. M. [1987]. The gradient model load balancing methods. IEEE Trans. on Software Engineering. 13, 32–38.

Lions, P. L. [1988]. On the Schwarz alternating method. In R. Glowinski et al. (eds.). *Domain Decomposition Methods for Partial Differential Equations*. Philadelphia: SIAM Publications, 1–42.

Löhner, R. and Morgan, K. [1987]. An unstructured multigrid method for elliptic problems, *Int. J. Num. Eng.*, 24, 101–15.

Lüling, R. and Monien, B. [1992]. Load balancing for distributed branch and bound algorithms. Proc. 6th Int. Parallel Processing Symp. (IPPS, 92), 543–49.

——— [1993]. A dynamic distributed load balancing algorithm with provable good performance. Proc. 5th Annual ACM Symp. on Parallel Algorithms and Architectures (SPPS, 92), 543–49.

Mavriplis, D. J. and Jameson, A. [1990]. Multigrid solution of the Navier-Stokes equations on triangular meshes. *AIAA J.*, 28, 8, 1415–25.

Meyer, F., Heide, A. D., Oesterdiekhoff, B., and Wanka, R. [1996]. Strongly adaptive token distribution. *Algorithmica.*, 15, 413–27.

Nichol, B., Buttlar, D., and Farrell, J. [1996]. *Pthreads Programming*. Paris: O'Reilly and Associates.

Rudoph, L., Slivkin-Allouf, M., and Upfal, E. [1991]. A simple load balancing scheme for task allocation in parallel machines. Proc. 3rd Annual ACM Symp. On Parallel Algorithms and Architectures (APAA, 91), 237–45.

Schunk, R. G., Canabal, F., Heard, G., and Chung, T. J. [1999]. Unified CFD methods via flowfield-dependent variation theory. AIAA paper 99–3715.

——— [2000]. Airbreathing propulsion system analysis using multithreaded parallel processing. AIAA paper, AIAA-2000-3467.

Schwarz, H. A. [1869]. Uber einige abbildungsaufgauben. *J. fur Die Reine und Angewandte Mathematik*, 70, 1005–20.