National Library of Canada
Acquisitions and Bibliographic Services Branch
395 Wellington Street
Ottawa, Ontario
K1A 0N4

Bibliothèque nationale du Canada
Direction des acquisitions et des services bibliographiques
395, rue Wellington
Ottawa (Ontario)
K1A 0N4

Your file Votre référence

Our file Notre référence

NOTICE

The quality of this microform is heavily dependent upon the quality of the original thesis submitted for microfilming. Every effort has been made to ensure the highest quality of reproduction possible.

If pages are missing, contact the university which granted the degree.

Some pages may have indistinct print especially if the original pages were typed with a poor typewriter ribbon or if the university sent us an inferior photocopy.

AVIS

La qualité de cette microforme dépend grandement de la qualité de la thèse soumise au microfilmage. Nous avons tout fait pour assurer une qualité supérieure de reproduction.

S'il manque des pages, veuillez communiquer avec l'université qui a conféré le grade.

La qualité d'impression de certaines pages peut laisser à désirer, surtout si les pages originales ont été dactylographiées à l'aide d'un ruban usé ou si l'université nous a fait parvenir une photocopie de qualité inférieure.

Canada

DATA PARALLEL SOLUTION STRATEGIES FOR IRREGULAR PROBLEMS

by

Dhrubajyoti Goswami

School of Computer Science
McGill University
Montréal, Québec
Canada

April 1995

A DISSERTATION
SUBMITTED TO THE FACULTY OF GRADUATE STUDIES AND RESEARCH
OF MCGILL UNIVERSITY
IN PARTIAL FULFILLMENT OF THE REQUIREMENTS FOR
THE DEGREE OF MASTER OF SCIENCE

ISBN 0-612-05556-6

Canada

Abstract

Irregular problems arise in many areas of computational physics and other scientific applications. A parallel solution for such a problem requires a suitable mapping strategy to map the irregular problem to the interconnection topology of the parallel machine so that communication overhead is as low as possible, there is proper load balancing among the processors and, locality of the problem graph is preserved. As a result, there is sufficient speedup in computation. In this thesis, we discuss some general strategies and associated results in data-parallel solutions of such problems. Some of these strategies use the geometrical co-ordinates of the nodes of the problem graph for partitioning them to the processors. In certain situations, geometrical co-ordinates alone may not capture the topology of the problem graph. To handle this, we design a number of schemes for assigning topological co-ordinates to nodes, and then use the same mapping strategies for partitioning nodes to processors based on their topological co-ordinates. We test our strategies on experimental problem graphs and discuss the results.

Résumé

Les problèmes irréguliers se présentent dans plusieurs domaines de calcul de physique et d'autres applications scientifiques. Une solution parallèle pour un de ces problèmes requiert une stratégie appropriée pour configurer ce problème irregulier à la topologie d'interconnection de la machine parallèle de façon à ce que la communication entre les processeurs soit minimale, qu'il y ait une distribution équilibrée du travail entre les processeurs, et que la localité du graphe du problème soit retenue. Par conséquent il y aura une accélération suffisante dans le calcul. Dans cette thèse, nous discutons de quelques stratégies générales et des résultats associés pour les solutions parallèles de ces problèmes. Quelques unes de ces stratégies utilisent les coordonnées géometriques des noeuds du graphe du problème pour les répartir entre les processeurs. Dans certaines situations, les coordonnées géometriques seules ne suffisent pas à capturer la topologie du graphe. Pour résoudre ce problème, nous concevons un nombre d'arrangements pour donner des coordonnées topologiques aux noeuds, et par conséquent utiliser les mêmes stratégies de configuration pour repartir les noeuds entre les processeurs en se basant sur leurs coordonnées topologiques. Nous testons nos stratégies sur des graphes de problèmes experimentaux et nous en discutons les résultats.

Acknowledgements

I would like to thank all the people who have made working on this thesis a more pleasant task. In particular, I would like to thank my supervisor, Prakash Panangaden, who gave me broad support and guidance throughout my time at McGill. I would like to thank Vincent Van Dongen from CRIM for all his suggestions, and Christophe Bonello for all the discussions regarding HPC. I would also like to thank Qui Ning for providing me with all the documents about Maspar and DECmpp12000/Sx, and Gilles Hurteau for all the help in trouble. I am thankful to Ajit Singh for the initial discussion on the topic.

There are many other people who I would like to thank. Clark Verbrugge gave me invaluable help in dealing with sequential CVFEM, and provided me with lots of information regarding papers on this topic, and also provided me with an abundant supply of *nodedata* files to carry out my experiments. Both Claudio Caballero and Nasser Elmasri were good friends in trouble, one providing me with good discussion on music and the other giving me good company in food. Wen Renhua was a good company for my entire stay in CRIM. I am thankful to Behram Kapadia for making the grammatical corrections of my thesis. My special thanks go to Herve Avril and Nasser Elmasri for helping me in writing the French version of the abstract.

I would like to acknowledge the following persons for being helpful to me in many ways: V.C.Sreedhar, Azzedine Boukerche, Rakesh Ghiya, Maryam Emami, Justiani, Alberto Jimenez, Anurag Garg, Vipul Jain, Alain Turki and myself. I would also like to thank my parents and my sister whose unconditional support made this thesis possible.

I would also like to acknowledge support from CRIM as part of the EPPP project, which made this work possible.

Dedicated to my parents.

Contents

List of Figures

Chapter 1

Introduction

In fields like fluid dynamics, electromagnetics, heat conduction, structural mechanics, combustion, and many other scientific applications, the mathematical model of a problem is often represented by partial differential equation(s) over a given regular or irregular domain, and initial and/or boundary conditions. One numerical approach to solving these type of equations is to discretize the given continuous domain into finite elements, and then convert the given system of differential equations into an equivalent system of algebraic equations [1, 14, 15]. In a 2-dimensional case, either triangular or 4-node or 8-node quadrilateral elements are chosen. In the case of triangular elements, delaunay triangulation is the technique employed in discretizing the domain [2, 11]. We call each corner of a triangle a *problem node* or simply a *node*. A wrench shaped domain and its corresponding triangular discretization is illustrated in *Fig. 1.1.*

The solution of these algebraic equations requires each node in the discretization to exchange data with its neighboring nodes, and to subsequently perform some computations with this data. In a direct iterative solution scheme, this process of communication and computation repeats over several iterations, until the computed values converge to some fixed values. Since each node in the discretization performs similar computations but on different sets of data, these types of problems are some of the best candidates for data-parallel solution.

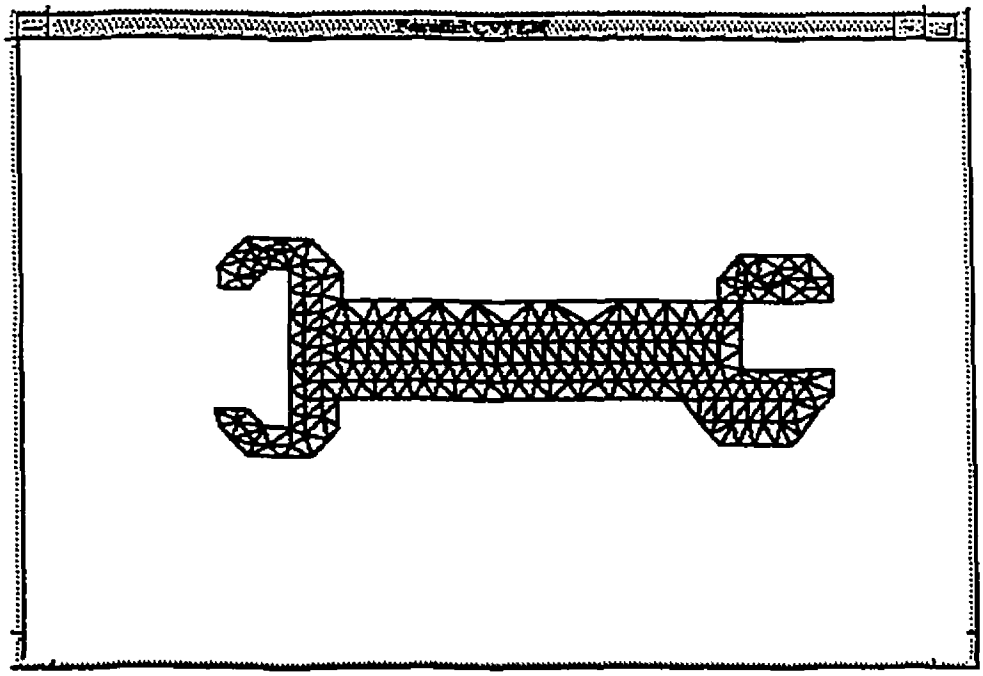


Figure 1.1: An irregular domain

In the rest of our discussion, we will use the term *problem graph* to represent the interconnection topology of the nodes in a discretization. Each node in a problem graph is called a *problem node* or simply a *node*. Similarly, the term *system graph* will represent the processor interconnection topology of a parallel machine. Each node in a system graph is called a *system node*, or simply a *processor*. For any node in a graph, all other nodes in the graph which are connected by a single edge to this node are called *neighbors* of this node.

Regular problems, where discretizations are fixed and topologically simple, can be effectively mapped to the parallel machine topology and then solved in a data-parallel way. In the case of irregular problems, where the number of neighbors of each node varies from node to node, some strategy has to be adopted in mapping the irregular problem graph to the system graph representing the interconnection topology of the parallel machine so that neighboring nodes in the discretization do not go far away from each other after being mapped, and the locality of the nodes is not destroyed. These issues are important, because the communication overhead in these machines

is much higher in comparison with the computational time. Also, the communication overhead increases significantly with increasing distance between two communicating processors.

The general mapping problem can be divided into two sub-parts. The first part is to find a suitable initial mapping strategy so that nodes and neighbors are initially mapped as close together as possible. The next part is to perform further adjustment of nodes among processors, either sequentially or in parallel, so that nodes and neighbors might come much closer. The present emphasis is to obtain some *good* initial mapping strategy so that there is no need for further adjustment of nodes among processors, which might save a considerable amount of pre-processing time.

One important issue here is load balancing. In a SIMD machine, where all processors work synchronously under a global clock, the time taken in some compound computation, for example a *while* loop, is the time taken by the last processor to finish the computation. So, if the load is not balanced properly, the processor which is loaded the most will have to do most of the work, while some other processors will have to remain idle until all the other processors finish. The total time is the time taken by the busiest processor. Hence, in order to keep all processors equally busy, load balancing is important.

In this thesis, we discuss some of the general strategies adopted in the initial mapping of an irregular problem graph to the system graph, keeping in mind the different issues discussed previously. First, we discuss a *greedy close neighbor heuristic*, which tries to map neighboring nodes in the problem graph as closely as possible. Being *greedy*, this strategy is not optimal in situations and it destroys locality of nodes in problem graphs. As an improvement over this, we design a quad-tree based mapping strategy which partitions nodes to processors based on geometric information of the nodes. This strategy is better at maintaining geometric locality of the problem graph over the previous strategy. However, it has got other limitations as well, which we further rectify in a binary-decomposition based mapping scheme. In certain situations, geometric information alone may not capture the topology of the problem

graph, because two nodes may be geometrically close enough, but they may not be neighbors in the actual discretization. Or in situations, they may be geometrically far off, but may be neighbors in the actual discretization. So, we bring in a notion of assigning topological co-ordinates to nodes, and design at least two different schemes for assigning topological co-ordinates to nodes. Finally, we combine all the ideas and design quad-tree based and binary-decomposition based mapping strategies based on topological co-ordinates of the nodes.

Some of these general strategies were implemented and tested on DECmpp12000/Sx, which is a mesh connected SIMD machine with toroidal wrap-around. The languages of implementation are C and MPL. It is found that both the quad-tree based and the binary-decomposition based mapping strategies based on both geometrical and topological co-ordinates give the best results.

1.1 Thesis background

A great deal of work has already been done in this direction. Some general readings on the sequential and parallel solutions of fluid flow and similar problems can be found in [16, 17, 18]. Different numerical methodologies on solutions of fluid flow, aerodynamics, and similar problems can be found in any journal on numerical methods for engineering. Some of the prominent journals dealing with numerical methodologies and computer solution techniques are *International Journal for Numerical Methods in Engineering*, and *Computer Methods in Applied Mechanics and Engineering*, to name a few.

An overview of the existing approaches on mapping of irregular problem graphs to multiprocessors can be found in [7]. There the authors discuss the different existing schemes, and conclude that methods based on recursive spectral bisection and recursive clustering have the most potential. As it is discussed in the paper, the different approaches commonly used for unstructured mesh decomposition are: simulated annealing, recursive graph bisection, nearest neighbor, recursive spectral bisection, and

recursive clustering. The last three approaches are the most prominent, and hence we discuss them briefly.

1.1.1 Nearest neighbor approaches

Nearest neighbor algorithms can be useful for mapping certain unstructured meshes with a fairly organized structure onto a multiprocessor configuration. Typically they proceed in two steps: an initial mapping where neighboring elements are grouped into clusters, and exchange of elements along boundaries to improve load balancing. The initial mapping is done by a technique called strip partitioning. The mesh is divided into horizontal and vertical strips. Techniques called 1-D and 2-D strip partitioning for meshes with rectangular elements and a hypercube system, are discussed in detail in [8, 13]. It was found that these techniques did not always work well, particularly with highly unstructured meshes. The reason being that, essentially the nearest neighbor is a geometric approach, whereas the mesh decomposition is a topological one.

Some techniques discussed in this thesis, namely the *quad-tree* based and the *binary-decomposition* based schemes, are conceptually similar to the above technique. The noticeable differences are that, (i) we use triangular elements, (ii) load balancing is taken care of in our initial mapping itself so that we do not perform a second phase of boundary refinement, and (iii) the most important is that, we use the topological information of the mesh in partitioning which should take care of unstructured meshes.

A similar binary-decomposition technique dealing with irregular meshes with rectangular elements and using the geometric information of the mesh can be found in [10]. There the authors theoretically study the communication cost of mapping this partitioning onto different multiprocessors: a mesh-connected array, a tree machine, and a hypercube.

1.1.2 Recursive sprectral bisection

As explored in [7], the recursive spectral bisection procedure(RSB) has been developed separately by both R.D. Williams and H.D. Simon. The approach is based upon the computation of a specific eigenvector of the Laplacian matrix of the connected graph, G. The algorithm assumes that the decomposition will produce connected sub-meshes. However, the authors in [7] point out that this is not guaranteed and it is not then clear how the algorithm would proceed, since the underlying theory only applies to connected graphs. Moreover, whenever the sub-meshes are connected then the resulting partition is nearest neighbor, and thus it has the advantages and suffers the same disadvantages as the general nearest neighbor strategy. The pros and cons of the nearest strategy have already been discussed previously.

1.1.3 Recursive clustering

This is an alternative topological approach which does not suffer the disadvantages of the RSB, and is discussed in [19]. It works as follows:

1. Arbitrarily assign each element to one of two clusters A and B so that there is an approximately equal number of elements in each cluster.
2. Evaluate the communication cost of this partition and find out which pair of elements when swapped give the maximum reduction in cost.
3. Temporarily removing the previously swapped pair, find the next best pair and continue until no more pairs remain.
4. From the set of all swaps, find the subset which minimizes the communication cost. Make the swap.
5. Repeat steps 1..4 recursively to obtain 2^n partitions.

This approach was found to work well on a wide variety of unstructured meshes. However, it has a number of limitations:

1. The mesh can only be partitioned into 2^n clusters.
2. Each split does not imply an overall optimal situation.
3. The optimization procedure tends to get caught in local minima.

4. One more important point, which is not mentioned in [7], is the pre-processing cost. Intuitively one can predict that a large amount of computational cost is involved in computing the communication cost for each swapped pair, and then finding the minimal cost subset, as in steps 2..4. Moreover, this cost accumulates recursively. In situations, this pre-processing cost might mask the benefit obtained from actual parallel computation.

The authors in [7] suggest an *extended recursive clustering algorithm* to take care of some of the limitations of the above scheme. The above scheme is limited to splits of 2^n clusters. They suggest an alternative scheme, where the mesh is arbitrarily split into N clusters, where N is the number of processors in the parallel system. As a next step, every pair of clusters is operated on to minimize the communication cost as in the previous scheme. However, this scheme also has the tendency to get stuck in some local minima. To take care of this, they suggest a change of the cost function in the optimization procedure. The simplest function to minimize is the total inter-processor distance traveled over the topology which enables all relevant communication to take place. However, it still has a large pre-processing cost, as pointed out in 4 above.

A similar technique has been used by the authors in [3] for mapping to CM-2, which is an SIMD machine with 64K processors. The underlying topology is an 11-dimensional hypercube of sprint nodes, where each sprint node is composed of 32 processors. The mapping comprises of an *initial mapping* which maps an equal number of tasks to each processor, followed by an iterative improvement of the mapping - parallel pairwise exchanges of tasks between the processors, somewhat similar to the above scheme, which also takes advantage of the interconnection topology and parallel communication features of the underlying machine. This scheme also suffers from the same limitations as pointed out in 2..4 above.

Some other mapping scheme specific to the Connection Machine system CM-2 have been discussed in [20]. The 65536 processors of the CM-2 the authors used were packed into 4096 16-processor chips, each having its own router node. The 4096 router

nodes were arranged in a hypercube of dimension 12. To cope with this topology, they proceed in two steps. First, the given mesh is decomposed into 4096 sub-meshes, each containing 16 connected finite elements. Next, they apply a heuristic mapper which identifies which hardware chip is to be mapped onto which sub-mesh. Finally, within each sub-mesh, elements are assigned randomly to the processors of the corresponding chip. The heuristic mapper in the second step above basically searches iteratively for a better mapping candidate through a two-step procedure for the minimization of the communication cost associated with a specific parallel machine topology.

However, the authors in [21] point out that although the above methods for CM-2 generate efficient mappings, the computational time required to compute a mapping may represent a substantial part of the total processing time. This makes them unsuitable for very large applications, and for applications where adaptive remeshing may be necessary. Alternatively, the authors use some random mapping strategies for their finite element solution of computational fluid dynamics problems on CM-2 and CM-200. However, they point out that these random mappings are not necessarily optimal for finite element problems, and the search for other mappings will be the topic for future research.

Some mapping strategies discussed above, namely the nearest-neighbor approaches, did not work quite well with highly unstructured meshes, mainly because they used the geometric information of the mesh for partitioning to processors. The idea of assigning some form of topological co-ordinates to nodes relative to one reference set and the subsequent mapping onto a processor array can be found in [8]. Similar techniques with reference to two reference sets can be found in [9]. Some of the many other interesting readings on the issue of mapping are [3, 4, 12].

1.2 Thesis organization

The thesis is organized as follows. First, we give a brief overview of a typical irregular problem, and the numerical method employed to solve it. Then we give a

brief overview of the machine on which we implement and test these general solution strategies. It is essential to adopt some suitable criteria to measure the quality of any specific mapping strategy for effective comparison. We discuss it in *chapter 3*. In the subsequent chapters, we discuss the different strategies adopted and their pros and cons, followed by a discussion on the implementation details on DECmpp12000/Sx and the results obtained with experimental problem domains. Finally, we conclude the thesis with a note on future work.

The next two chapters are mainly reviews. Our contributions are discussed in the subsequent chapters.

Chapter 2

An irregular problem

We consider a steady-state, 2-D, heat conduction type problem. As an example, let us consider a rectangular plate, one side of which is being constantly heated by a steady heat source. There is no turbulent motion of the air surrounding the plate (or, the plate is in vacuum) and, as a result, the temperature of the plate has reached a steady-state. We are interested in finding the steady-state temperatures at different points of the plate. The mathematical model of such a problem is composed of partial differential equation(s), and some domain of interest, with given boundary conditions. In cartesian co-ordinates, the governing equation for such problems is of the general form:

$$\Gamma_\phi\left(\frac{\partial^2\phi}{\partial x^2}+\frac{\partial^2\phi}{\partial y^2}+\frac{\partial^2\phi}{\partial z^2}\right)+S_\phi=0 \tag{2.1}$$

Here, ϕ is a general scalar dependent variable, Γ_ϕ is the corresponding diffusion coefficient, and S_ϕ is the appropriate volumetric generation rate or the source term. In the case of our example problem, the scalar dependent variable is the temperature T, the diffusion coefficient is the thermal conductivity K of the material of the plate, and S_T is the volumetric rate of heat generation. Given a domain of interest, with given boundary conditions, we are interested in finding the steady-state temperatures at different points of the domain.

2.1 Derivation of the heat conduction equation

Let $\vec{q}$ be the heat flux and S_T be the volumetric rate of heat generation. In vectorial form, the heat flux at any point (x, y, z) can be represented as:

$$\vec{q} = q_x\hat{i} + q_y\hat{j} + q_z\hat{k} \tag{2.2}$$

where q_x, q_y and q_z are the components along X, Y and Z axes respectively. Let us consider a control volume $\Delta x \Delta y \Delta z$, as in *Fig. 2.1.*

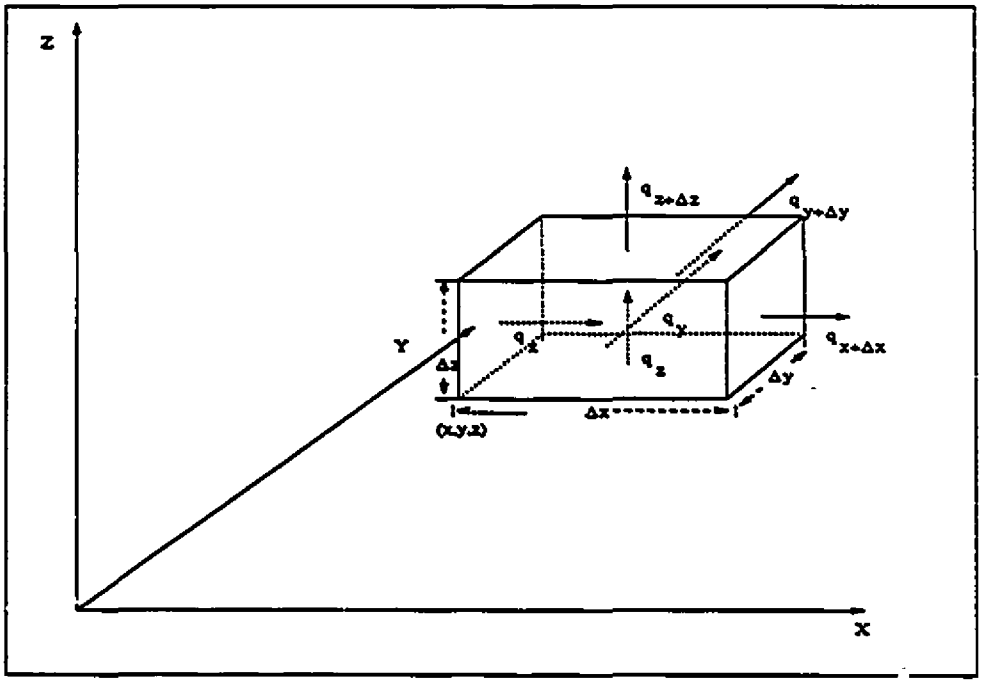


Figure 2.1: Energy conservation in a control volume

By the energy balance of the control volume, we obtain

$$(q_{x+\Delta x} - q_x)\Delta y \Delta z + (q_{y+\Delta y} - q_y)\Delta z \Delta x + (q_{z+\Delta z} - q_z)\Delta x \Delta y = S_T \Delta x \Delta y \Delta z \tag{2.3}$$

By Taylor's series expansion, we have:

$$q_{x+\Delta x} = q_x + \frac{\partial q_x}{\partial x}(\Delta x) + (\frac{1}{2})\frac{\partial^2 q_x}{\partial x^2}(\Delta x)^2 + \cdots \tag{2.4}$$

Similarly, we can expand $q_{y+\Delta y}$ and $q_{z+\Delta z}$. Now, substituting in (2.3) and simplifying,

$$\begin{aligned}(\frac{\partial q_x}{\partial x}\Delta x + (\frac{1}{2})\frac{\partial^2 q_x}{\partial x^2}(\Delta x)^2 + \cdots)\Delta y\Delta z + (\frac{\partial q_y}{\partial y}\Delta y + (\frac{1}{2})\frac{\partial^2 q_y}{\partial y^2}(\Delta y)^2 + \cdots) \\ \Delta z\Delta x + (\frac{\partial q_z}{\partial z}\Delta z + (\frac{1}{2})\frac{\partial^2 q_z}{\partial z^2}(\Delta z)^2 + \cdots)\Delta x\Delta y = S_T\Delta x\Delta y\Delta z \\ \Rightarrow (\frac{\partial q_x}{\partial x} + (\frac{1}{2})\frac{\partial^2 q_x}{\partial x^2}(\Delta x) + \cdots) + (\frac{\partial q_y}{\partial y} + (\frac{1}{2})\frac{\partial^2 q_y}{\partial y^2}(\Delta y) + \cdots) + \cdots = S_T \qquad (2.5)\end{aligned}$$

In the limiting case as Δx, Δy, $\Delta z \to 0$, we obtain from (2.5),

$$\frac{\partial q_x}{\partial x} + \frac{\partial q_y}{\partial y} + \frac{\partial q_z}{\partial z} = S_T \qquad (2.6)$$

Now, by Fourier law of thermal diffusion:

$$\begin{aligned}\vec{q} &= -K\vec{\nabla}T \\ \Rightarrow q_x\hat{i} + q_y\hat{j} + q_z\hat{k} &= -K(\tfrac{\partial T}{\partial x}\hat{i} + \tfrac{\partial T}{\partial y}\hat{j} + \tfrac{\partial T}{\partial z}\hat{k}) \qquad (2.7)\end{aligned}$$

Combining (2.6) and (2.7), we obtain,

$$K(\frac{\partial^2 T}{\partial x^2} + \frac{\partial^2 T}{\partial y^2} + \frac{\partial^2 T}{\partial z^2}) + S_T = 0 \qquad (2.8)$$

Equation (2.8) can be represented in the form:

$$\vec{\nabla}.(K\vec{\nabla}T) + S_T = 0 \qquad (2.9)$$

where $\vec{\nabla} = \frac{\partial}{\partial x}\hat{i} + \frac{\partial}{\partial y}\hat{j} + \frac{\partial}{\partial z}\hat{k}$.

In general, we are interested in solving equation of the form:

$$\vec{\nabla}.(\Gamma_\phi\vec{\nabla}\phi) + S_\phi = 0 \qquad (2.10)$$

2.2 Numerical solution of the problem

We consider a numerical solution for a 2-D case of such a problem. One numerical way of solving these types of problems is the *control volume finite element method*, *CVFEM* for short [1]. It involves the following steps:

- Discretization of the calculation domain into finite elements.
- Prescription of suitable element based interpolation, or shape, functions for the dependent variables.
- Derivation of discretization equations, which are algebraic approximations to the governing equations.
- Solve the discretization equations.

Below, we elaborate the different steps.

- Step 1: This involves the discretization of the given domain into finite elements. In a two dimensional case, either triangular or 4-node or 8-node quadrilateral elements are chosen. In our case, we use triangular elements. The domain of interest is divided into triangular elements. *Delaunay triangulation* using Bowyer's algorithm [2] is the technique employed in triangulating the domain. As a next step, the centroids of the triangular elements are joined to the midpoints of the corresponding sides, which creates polygonal control volumes around each node in the calculation domain. A sample domain discretization is shown in *Fig. 2.2.* Here, the solid lines denote the domain and element boundaries, the dashed lines represent the control-volume faces, and the shaded areas show the control volumes associated with one internal and one boundary node.

  In *Fig. 2.3(a,b,c)*, the control volume around an arbitrary node i is shown by a dashed outline. The values of ϕ are computed at the nodes of the discretization, which correspond to the corners of the triangular elements.

- Step 2: This involves the prescription of suitable interpolation, or shape, functions for the dependent variables ϕ, Γ_ϕ and S_ϕ. The reason being that, these variables are available only at the interfaces of the control volumes, and are not available within the triangular elements. The solution is to prescribe some suitable interpolation functions so that they can be defined inside these triangular

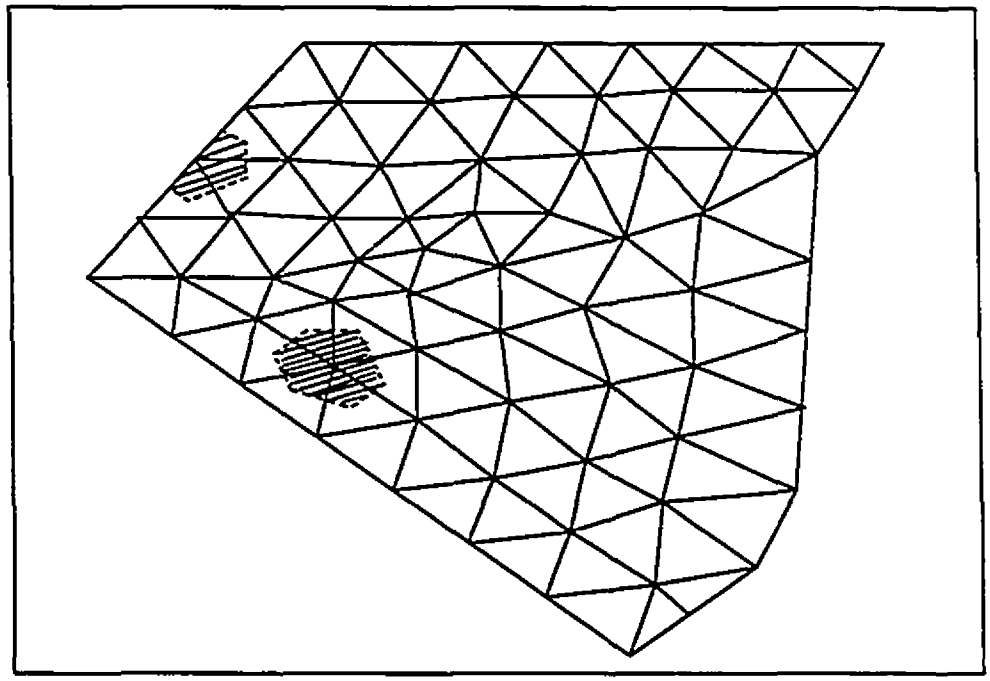


Figure 2.2: Discretization of an irregular domain

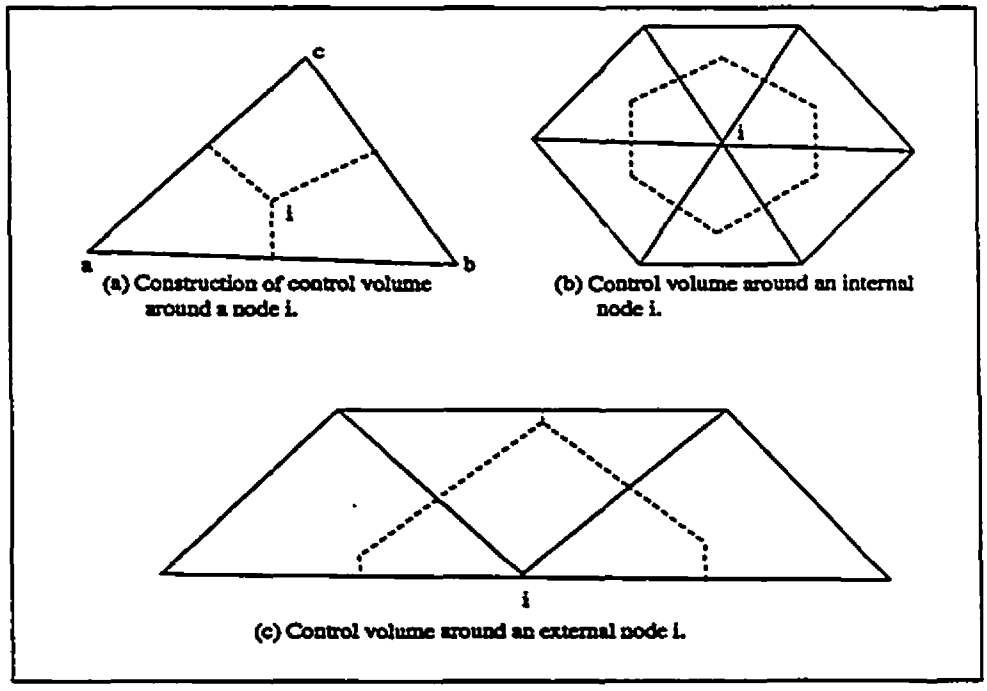


Figure 2.3: Control volume generation

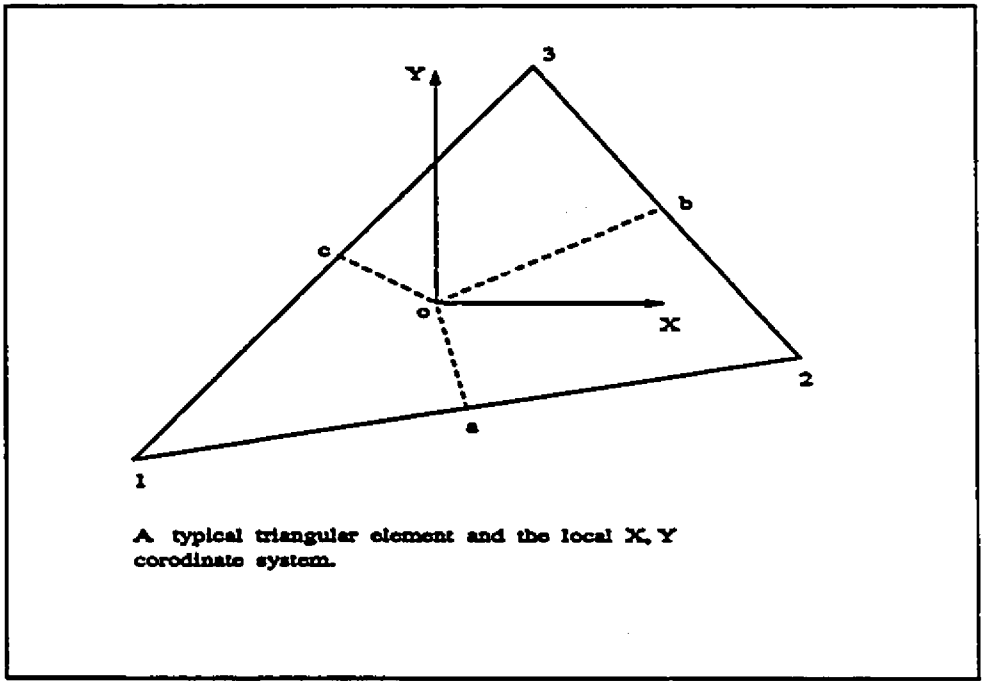
elements. For instance, the dependent variable ϕ is interpolated linearly by:

$$\phi = Ax + By + C \tag{2.11}$$

where A, B and C are constants for a triangular element, and (x, y) is the co-ordinate of the point where ϕ is to be computed, assuming a two dimensional case. Thus, considering the triangular element 123 and its local x-y co-ordinate system with centroid o as the origin as in *Fig. 2.4*, the values of ϕ at the three corners of the triangular element are given by:

$$\phi_i = Ax_i + By_i + C, \qquad i = 1..3 \tag{2.12}$$

where (x_i, y_i), $i = 1..3$, are the local co-ordinates of the three corners of the triangular element.



A typical triangular element and the local X, Y corodinate system.

Figure 2.4: A triangular element

From (2.12), A, B, and C can be computed for the triangular element as:

$$A = [(y_2 - y_3)\phi_1 + (y_3 - y_1)\phi_2 + (y_1 - y_2)\phi_3]/DET \tag{2.13}$$

$$B = [(x_3 - x_2)\phi_1 + (x_1 - x_3)\phi_2 + (x_2 - x_1)\phi_3]/DET \quad (2.14)$$

$$C = [(x_2y_3 - x_3y_2)\phi_1 + (x_3y_1 - x_1y_3)\phi_2 + (x_1y_2 - x_2y_1)\phi_3]/DET \quad (2.15)$$

where $DET = (x_1y_2 + x_2y_3 + x_3y_1 - y_1x_2 - y_2x_3 - y_3x_1)$.

Similarly, in each triangular element, the centroidal value of Γ_ϕ is stored and assumed to prevail over the corresponding element. The source term S_ϕ is assumed to prevail in the following form:

$$S_\phi = S_C + S_P\phi \quad (2.16)$$

Centroidal values of S_C and S_P are stored and they too are assumed to prevail over the corresponding element.

- Step 3: The basic idea is to apply the energy conservation principle to finite control volumes in the calculation domain. Applying the integral conservation equation for (2.10) over a control volume V, as in *Fig. 2.5*,

$$\oint_V (\vec{\nabla}.(\Gamma_\phi \vec{\nabla}\phi) + S_\phi)dV = 0$$

$$\Rightarrow \oint_V \vec{\nabla}.(\Gamma_\phi \vec{\nabla}\phi)dV + \oint_V S_\phi dV = 0 \quad (2.17)$$

Now, by Gauss's divergence theorem:

$$\oint_V \vec{\nabla}.(\Gamma_\phi \vec{\nabla}\phi)dV = \int_S (\Gamma_\phi \vec{\nabla}\phi.\hat{n})dS \quad (2.18)$$

where $\hat{n}$ is the unit normal vector to the surface. Now, from (2.16), we get:

$$\int_S (\Gamma_\phi \vec{\nabla}\phi.\hat{n})dS + \int_V S_\phi dV = 0 \quad (2.19)$$

Now, let us consider a typical node i in the calculation domain; it could be an internal node or a boundary node as in *Fig. 2.3*. Applying (2.18) to the control volume associated with node i in *Fig. 2.4*, we get:

$$(\int_a^o (\Gamma_\phi \vec{\nabla}\phi.\hat{n})dS + \int_o^c (\Gamma_\phi \vec{\nabla}\phi.\hat{n})dS + \int_{iaoc} S_\phi dV) + (\textit{similar contributions from other elements associated with node } i) +$$

$$(\textit{boundary conditions if applicable}) = 0 \quad (2.20)$$

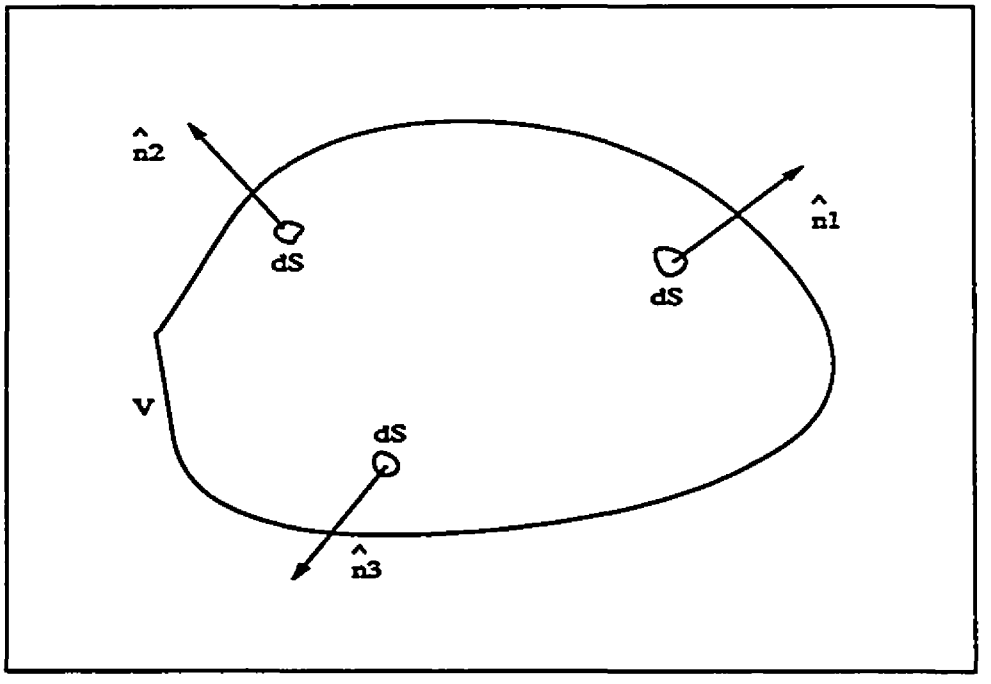


Figure 2.5: An arbitrarily shaped control volume V

From (2.11), we obtain,

$$\begin{aligned} \Gamma_\phi \vec{\nabla}\phi &= \Gamma_\phi(\frac{\partial \phi}{\partial x}\hat{i} + \frac{\partial \phi}{\partial y}\hat{j}) \\ &= \Gamma_\phi A\hat{i} + \Gamma_\phi B\hat{j} \end{aligned} \tag{2.21}$$

With reference to the element 123 in *Fig. 2.4*, and its local x-y co-ordinate system, we have:

$$\vec{oa} = x_a\hat{i} + y_a\hat{j} \tag{2.22}$$

and the unit normal vector to it is given by,

$$\hat{n}_{oa} = \frac{-y_a\hat{i} + x_a\hat{j}}{|\vec{oa}|} \tag{2.23}$$

$$and,\ \hat{n}_{ao} = \frac{y_a\hat{i} - x_a\hat{j}}{|\vec{ao}|} \tag{2.24}$$

Similarly, we can represent $\hat{n}_{ob}$ and $\hat{n}_{oc}$. Combining (2.22) and (2.24), we get:

$$\int_a^\circ (\Gamma_\phi \vec{\nabla}\phi.\hat{n}_{ao})dS = \int_a^\circ (\Gamma_\phi A\hat{i} + \Gamma_\phi B\hat{j}).\frac{y_a\hat{i} - x_a\hat{j}}{|\vec{ao}|}dS$$

$$
\begin{aligned}
&= \int_a^o \frac{\Gamma_\phi A y_a - \Gamma_\phi B x_a}{|\vec{ao}|} dS \\
&= \frac{\Gamma_\phi A y_a - \Gamma_\phi B x_a}{|\vec{ao}|} |\vec{ao}| \\
&= \Gamma_\phi A y_a - \Gamma_\phi B x_a
\end{aligned} \tag{2.25}
$$

In a similar way,

$$\int_o^c (\Gamma_\phi \vec{\nabla}\phi.\hat{n}) dS = -\Gamma_\phi A y_c + \Gamma_\phi B x_c \tag{2.26}$$

The equation involving the source term in (2.20) is approximated as:

$$\oint_{iaoc} S_\phi dV = \frac{A_e}{3} S_C + \frac{A_e}{3} S_P \phi_1 \tag{2.27}$$

where $A_e = |DET|/2$ is the area of the element 123. DET is defined with reference to (2.15). Now, substituting values of A, B and C from (2.15), and then combining (2.20), (2.25), (2.26), and (2.27) we obtain,

$$\int_a^o (\Gamma_\phi \vec{\nabla}\phi.\hat{n}) dS + \int_o^c (\Gamma_\phi \vec{\nabla}\phi.\hat{n}) dS + \int_{iaoc} S_\phi dV = C_1\phi_1 + C_2\phi_2 + C_3\phi_3 + B_1 \tag{2.28}$$

where,

$$C_1 = \frac{\Gamma_\phi}{DET}[(y_a - y_c)(y_2 - y_3) + (x_a - x_c)(x_2 - x_3)] - \frac{A_e}{3} S_P \tag{2.29}$$

$$C_2 = \frac{\Gamma_\phi}{DET}[(y_a - y_c)(y_3 - y_1) + (x_a - x_c)(x_3 - x_1)] \tag{2.30}$$

$$C_3 = \frac{\Gamma_\phi}{DET}[(y_a - y_c)(y_1 - y_2) + (x_a - x_c)(x_1 - x_2)] \tag{2.31}$$

$$B_1 = \frac{A_e}{3} S_C \tag{2.32}$$

Expressions similar to (2.28) can be evaluated for all other triangular elements associated with node i, as in *Fig. 2.3.* Substituting them in (2.20) we get:

$$a_i \phi_i = \sum_j (a_j \phi_j) + b_i \tag{2.33}$$

where the summation is performed over all the neighboring nodes of node i. Here, a_i's and a_j's are coefficients for each node, and are defined in terms of the co-ordinates of the nodes, Γ_ϕ, S_C and S_P. Thus, these coefficients can be

computed beforehand. Equations similar to (2.33) can be derived for all internal nodes in the calculation domain.

For nodes that lie on the boundary of the calculation domain, the discretization equation for any boundary node i is replaced by the following equation:

$$\phi_i = \phi_{specified} \tag{2.34}$$

where $\phi_{specified}$ is the boundary value of that node specified beforehand.

- Step 4: This step involves the solution of the discretization equations specified by (2.33) and (2.34) for all i. These discretization equations form a set of simultaneous algebraic equations. As a direct solution technique, the following iterative procedure can be used to solve these equations:
  1. Guess all unknown values of ϕ in the calculation domain.
  2. Compute the new value of ϕ_i for each node i based on previous values of ϕ_j's for all neighboring nodes j of i, as specified by (2.33).
  3. Repeat step 2 above until all ϕ's converge.

2.3 Parallel solution of the discretization equations

Since each node i in the discretization has to perform similar computations with the data from its neighboring nodes, as specified by (2.33), by performing them in a data parallel way should give a large performance gain. The nodes in the discretization can be mapped to the processors, and computations and communications can then be carried out in parallel over several iterations until the computed values converge. It is clear that communication occurs between nodes and their immediate neighbors. Hence, the farther the nodes and neighbors are mapped, the more is the communication overhead. Here, the distance between a mapped node and one of its mapped neighbors is measured by the minimum number of physical communication edges

from the processor where the node is mapped to the processor where the neighbor is mapped. In a subsequent chapter, we discuss this issue in more detail. The issue of mapping nodes and neighbors close enough comes here, since the communication overhead has to be as low as possible to obtain a good performance gain.

Another issue here is load balancing. At one extreme, we can imagine mapping all the nodes to a single processor, which is equivalent to sequential processing. Obviously, it will give the least performance gain. The key issue here is to keep all the processors equally busy with a well balanced load during the entire process of communication and computation. Since the domain of interest can be of any shape and the number of neighbors of each node can vary from node to node, it is an *irregular problem.*

We repeat the following for convenience to the reader. We use the term *problem graph* to interpret the graph representing the interconnection topology of the nodes in the discretization. Each node in a problem graph will be called a *problem node* or simply a *node.* The term *system graph* will represent the processor interconnection topology of the parallel machine. Each node in the system graph will be termed a *processor node* or simply a *processor.* Hence, the mapping problem is to find some suitable strategy to map the problem nodes to the processors so as to reduce the communication overhead and to maintain a good locality of the nodes, as well as to maintain a well balanced load on all the processors.

Chapter 3

DECmpp12000/Sx: An overview

The general strategies designed were implemented and tested on DECmpp12000/Sx. We first give a brief overview of the machine before describing the different strategies.

DECmpp12000/Sx is a data parallel system, often called a single instruction, multiple data(SIMD) system [6]. In SIMD systems, a single program instruction can be executed simultaneously on many relatively small processors, and on different data. It consists of one instruction fetch unit, and more than one data processors. The instruction stream is processed serially, but all data processors function simultaneously on independent data. With at least 1000 processors, it is a massively parallel system.

Massively parallel data systems can be many times faster than other systems for suitable applications. Applications that make best use of DECmpp12000/Sx system and the language MPL are those in which the same instruction is being executed on thousands of data points at once. Examples of such applications include imaging, fluid mechanics, and thermodynamics, to name only a few. The specific example which we considered in the previous chapter falls in this category. In order to take full advantage of the system, we should use some language, for instance MPL, which

can address parallel data efficiently. We will discuss about the language MPL shortly.

The complete layout of the system is as follows. It consists of a console system, which is either a DECsystem 5900 server or a DECstation 5000/240, and a data parallel unit(DPU). The console system runs the ULTRIX operating system and uses standard I/O. The DPU consists of an array control unit (ACU), an array of at least 1K processor elements(PEs) which can go upto 16K, and PE communications mechanisms. In *fig 3.1*, we show a conceptual diagram of the system. In the next section, we give a brief outline of the different system components.

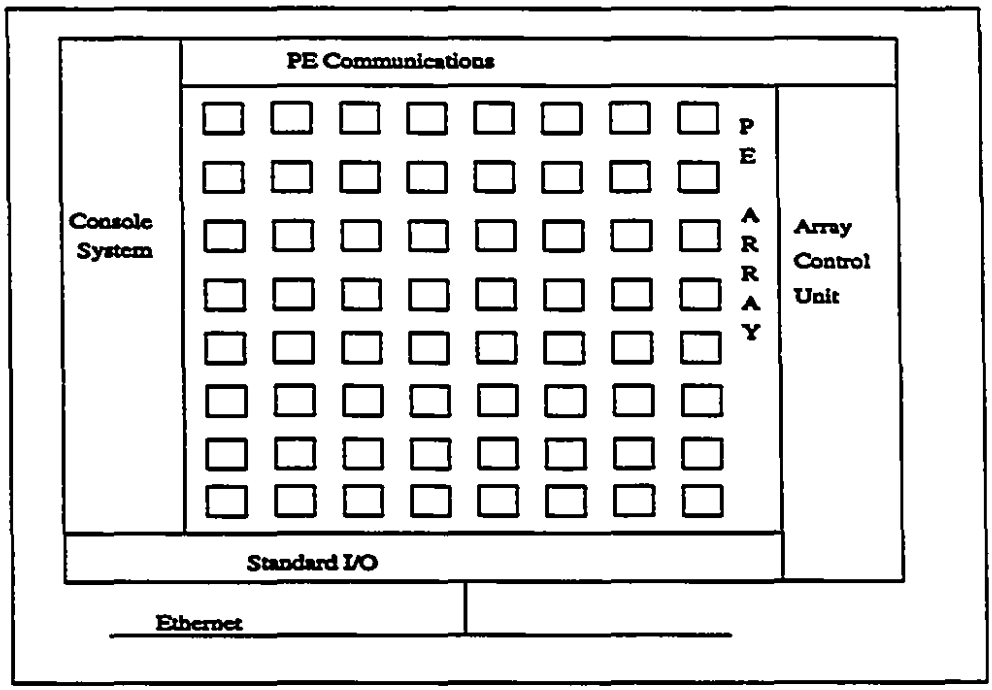


Figure 3.1: A conceptual diagram of DECmpp12000/Sx

3.1 System components

- Console system: This is a scalar processor that runs the ULTRIX operating system and provides standard I/O devices. The console system is either a

DECsystem 5900 or a DECstation 5000/240.

- Data parallel unit(DPU): The DPU is where all parallel processing is done. It includes the array control unit(ACU) and the processor elements(PE) array.
- Array control unit(ACU): The ACU is a processor with its own registers and data and instruction memory. It has upto 22 32-bit registers available for user declared register variables, 128 KB of data memory, and 1MB of RAM that expands to 4 GB of virtual instruction memory. It controls the PE array and performs operations on singular data. The ACU sends data and instructions to each PE simultaneously.
- Processor elements(PEs): Each PE is an arithmetic processing element with dedicated registers and RAM. Each PE has 40 32-bit registers, upto 33 of which are available for user-declared register variables. Each PE has either 16 KB or 64 KB of RAM. Each PE receives the same instruction from the ACU, and if it is enabled, it executes the instruction on variables that reside on itself. In MPL, any variable that is declared to be a PE variable is replicated exactly, except for its value, on every PE.
- PE array: It is the two-dimensional representation of all PEs in a system. A system has 1K, 2K, 4K, 8K or 16K PEs, and these are arranged in a matrix that has either an equal number of rows and columns or twice as many columns as rows. A processor in the PE array can be identified by its co-ordinate (x, y), where x and y denote the row and column respectively of the processor. The first row/column is called the 0_{th} row/column.
- PE communication: This constitutes communication between PEs and the ACU and between two PEs in the PE array. Communication between PEs and ACU takes place over a special bus. Communication between two PEs consists of the XNet communication and the global router communication.
- XNet communication: It constitutes the synchronous direct communication between any PE and any other PE that lies on a straight line from the original

PE in one of the following eight directions in the PE array: north, northeast, east, southeast, south, southwest, west, or northwest.

- Global router communication: It constitutes the asynchronous communication between any particular PE and any other PE in the PE array.

3.2 DECmpp12000/Sx programming language

MPL is the lowest level programming language that the DECmpp12000/Sx system supports. The purpose of MPL is to program the DPU, and can be used to encode the appropriate portion of an application in a data parallel way. These MPL subroutines can be called from the scalar program running in the console system, written in either C or Fortran.

MPL is based on ANSI C. All ANSI C language features are supported by the MPL compiler. In addition, keywords, statements, and library functions have been added to support data parallel programming. In general, the additions are:

- A new keyword, *plural*, distinguishes between two independent address spaces. Variables defined using the keyword *plural* are located identically on each PE in the PE array. Variables defined without this keyword are singular variables and are allocated on the ACU.
- All arithmetic and addressing operations are supported for plural data types.
- SIMD communications are implemented. Two explicit constructs, XNet and router, are used for data communication in the PE array.
- SIMD control statement semantics are supported. SIMD control flow also controls the active set, which is the set of PEs that is enabled at any time during execution. The size of this set can be no larger than size of the physical PE array.

Besides, the DECmpp Sx parallel programming environment includes tools to compile and analyze programs. These tools constitute the compilers, debuggers, visualizers, and profilers.

3.3 The communication primitives

There are two types of communication primitives, the synchronous *XNet* communication and the asynchronous *router* communication. There are also library routines which use these two constructs. A detailed discussion regarding the timings with these constructs can be found in [5, 6].

The performance of the *router* construct depends on the number of collisions. Because of this reason, when one PE wants to broadcast some value to all other PEs, it can be slower by a factor of the total number of processors, as compared to the case when one PE communicates with only one other PE. Thus, it is efficient when many PEs communicate with other PEs, and inefficient when every PE communicates with the same PE. In our present experiments, we have adopted no measures to count the number of collisions, or to reduce them.

3.4 The programming model

The recommended model for programming with MPL is:
1. Start with a DEC Fortran or C program.
2. Compile and run the program, and verify that it runs correctly under ULTRIX on the console system.
3. Examine the program to find out what portion(s) of it can be executed in a data parallel way. Also decide which variables should be redefined as *plural* and operated on in a data parallel way.

4. Replace the code identified in step 3 with MPL subroutines, and call these subroutines using *callRequest* MPL library function.

An experienced MPL programmer can straightaway write the routines in C and MPL without following the above model. We show some important constructs of the language through a few examples.

3.4.1 Example 1: Console to DPU communication

This example shows a C program that calls an MPL subroutine. It shows how the MPL subroutine is called by a scalar program and how data is passed to and from the subroutine. First we show the scalar code in C and then the MPL code. The scalar code is in standard C except for the call to the *callRequest* MPL library function.

```
#include <mpl.h>
extern mpl_sub(); /* MPL names must be declared "extern" in C to
                     be used */
main()
{
   double arr1[1024], arr2[1024];
   double f,g;
   int i;
   for (i = 0; i<1024; i++)
      arr2[i] = double(i);
   g = 1.25;

   /* Now call the MPL subroutine, passing the data addresses.
      It is required to pass the addresses of everything that
      one wants to transfer either into or out of the MPL sub-
```

```
      -routine using copyin, copyout or blockin, blockout.
   */

   callRequest(mpl_sub, 16, &f, &g, arr1, arr2);

   /* Above, we call the MPL routine "mpl_sub" from this scalar
      routine. Here 16 is the size in bytes of the remaining
      arguments in the call, four 4-byte pointers.
   */

   printf("arr2[0] = %f, f = %f\n",arr2[0],f);
}
```

To call an MPL subroutine from the scalar program, we have to use the *callRequest* MPL library function, as shown in the above piece of static C program running on the console. In order to pass arguments from the scalar program to the MPL subroutines, we have to use the *callRequest*, *copyIn*, *copyOut*, *blockIn*, and *blockOut* MPL library functions. The third to the n_{th} arguments to *callRequest* are all passed to the subroutine named in the first argument, *mpl_sub* in this case. The second argument is the total number of bytes of the third through n_{th} arguments.

The code for the MPL subroutine, *mpl_sub*, is as follows:

```
#include <mpl.h>

visible mpl_sub(); /* Must be declared "visible" for C in the console
                      to use it */
mpl_sum(fe_f_p, fe_g_p, fe_arr1_p, fe_arr2_p)
   void *fe_f_p;
   void *fe_g_p;
   void *fe_arr1_p;
   void *fe_arr2_p;
```

```
{
   plural double arr1, arr2;
   double f, g;
   copyIn(fe_g_p, &g, sizeof(g)); /* Transfer the value of "g" from
                                     the C routine in console */
   blockIn(fe_arr2_p, &arr2, 0, 0, nxproc, nyproc, sizeof(arr2));
   f = reduceAdd(arr2);
   arr1 = arr2/f + g;
   copyOut(&f, fe_f_p, sizeof(f));
   blockout(&arr2, fe_arr2_p, 0, 0, nxproc, nyproc, sizeof(arr2));
}
```

Here, *copyIn* and *blockIn* routines are used to transfer data from the console to the DPU. Similarly, *copyOut* and *blockOut* are used to transfer data from the DPU to the console. Similar programming model and library functions are used in our implementation of parallel CVFEM, and hence we bring in this example as a concise illustration.

3.4.2 Example 2: Pixel averaging

In this example, there is a rectangular array of image elements, each with an 8-bit intensity measure. We want to smooth out the image by revising each value to represent the average of itself and its eight nearest neighbors. We assume that each pixel element is mapped to a processor element, and there are an equal number of pixel and processor elements. This example basically illustrates the applicability of the Xnet.

```
#include <mpl.h>

/* The following subroutine averages each pixel with the values of
   its 8 neighbors, N, NE, E, SE, S, SW, W, NW. We ignore the PE
```

```
   array edges,i.e. torroidal wraparound is taken into account.
   It turns out that only 4 Xnets are necessary, since the first
   line accumulates the values of NE and NW neighbors to the N
   neighbor, and the second line accumulates the values of SE and
   SW neighbors to the S neighbor.
*/
plural int
average(src)
plural int src;
{
   src += xnetW[1].src + xnetE[1].src;
   src += xnetN[1].src + xnetS[1].src;
   return(src);
}
```

3.4.3 Example 3: Greatest Common Divisor

The following example illustrates how the control flow in the PE array works out. This is a *plural* version of Euclid's classical Greatest Common Divisor algorithm. Values of x and y may be different on each PE. The code works as expected in MPL in that the *while* statement is not finished until all PE's are finished with it, but PEs that finish early are temporarily disabled until all PEs are finished. In the *if* statement, some PEs may execute the *then* clause and some PEs may execute the *else* clause.

```
/* The following routine finds the GCD of two numbers x and y */

plural int GCD(x,y)
   plural int x,y;
{
   while (x != y)
     if (x > y) x = x - y;
```

```
    else y = y - x;
  return (x);
}
```

The example is important because it demonstrates how some processors can be made idle by the proper use of some flag. All processors whose flags are set can be made to execute the *if* part of the statement, while the remaining processors can be made to remain idle. Thus, with proper use of some active flag, we can effectively select an active set of processors, which have to perform some specific operation at some instant of time. The *while* loop above also illustrates why load balancing is important in our application. The *while* statement is not finished until all processors are finished with it. So, in a load-dependent *while* statement, it is essential to distribute loads evenly on all processors in order to keep them equally busy.

Chapter 4

Some initial mapping strategies

Before we start our discussion on different initial mapping strategies, we have to first fix some measure(s) which can suitably predict the *quality* of any mapping strategy. The main objectives of our mapping strategies are to maintain a reduced communication overhead and a good locality of the nodes of the *problem graph*, as well as maintain a well balanced load on all the processors. Accordingly, we should first define one or more *objective functions* which can suitably measure some of these objectives. The next step is to design the mapping scheme which can minimize the values of specific objective function(s). Before defining our objective functions, we introduce a few definitions which will be used in the rest of the discussion.

4.1 Measuring the quality of a mapping strategy

The discussion in this section regarding objective functions follows from the work already described in [3, 4]. We first define a few terms which will be used in the rest of the discussion.

A *problem graph* G_P is formally defined as:

$G_P =< V_P, E_P >$, where
$V_P = (v_{pk} | v_{pk}$ is a problem node)
$E_P = (e_{pij} | e_{pij}$ is a problem edge between problem nodes
v_{pi} and $v_{pj})$

Here, V_P and E_P are the sets of *problem nodes* and *problem edges* respectively.

Let v_{pi} and v_{pj} be any two problem nodes. We say that they are *connected* if there exists a path from one node to the other, where a path denotes a sequence of edges. A path from v_{pi} to v_{pj} can be represented by $e_{pi,l_0,l_1,..,l_n,j}$ where $e_{pil_0}, e_{pl_0l_1}, .., e_{pl_nj} \in E_P$. A problem graph G_P is *connected* if and only if each pair of nodes in V_P are connected. From now on, whenever we consider a problem graph, we will assume that it is connected.

The *nominal distance* between two problem nodes v_{pi} and v_{pj} is the length of the shortest path between the two nodes. This nominal distance is denoted by D_{pij}.

Similarly, we define the terms G_S, V_S, and E_S for the *system graph* representing the interconnection topology of the parallel machine. For convenience, we will denote each system node in V_S as a *processor node* or simply as a *processor*. We denote the *nominal distance* between two processors v_{sk} and v_{sl} by D_{skl}. For instance, in the case of a Hypercube, this *nominal distance* between any two processors is the Hamming distance between them. Similarly, in the case of DECmpp12000SX, which is a two-dimensional mesh, the *nominal distance* between any two processors is the minimal number of hops in the X and Y directions from one processor to the other, taking toroidal wrap-around into account.

Now we are in a position to define one of our criteria for quality measurement. One of the objectives of the mapping problem is to find a suitable surjection $\psi : V_P \rightarrow V_S$, so that the value of the objective function λ is minimized. We define λ as follows:

$$\lambda = \sum_{e_{pij}} D_{skl}, \quad e_{pij} \in E_P \tag{4.1}$$

where, $\psi(v_{pi}) = v_{sk}$, and $\psi(v_{pj}) = v_{sl}$.

Thus we are trying to minimize the sum of the message distances between mapped nodes and neighbors. It is important to note here that in our practical applications, it is not necessary that message between two processors will always follow the minimal path between the processors. Still, the above measure gives an idea of the proximity of the nodes after being mapped. At a first glance, it looks as if it should be the only criterion for optimization, but we should be slightly careful at this point. It should be noted that reduction of this λ alone does not mean speedup in actual computation. In fact, speedup depends on some other factors as well. For instance,

- *Load balancing* is one important factor to be taken into consideration. In one extreme, we can make λ equal zero by mapping all the nodes to a single processor, but this will make things highly sequential. If we don't take load balancing into account, making λ small might result in an imbalanced load on all the processors. This can make things equally worse, because some processors might have to wait idle, while some others might do most of the work, and the net effect is the time taken by the busiest processor.

- Another issue is the number of *collisions* in the message paths. For instance, in the case of DECmpp12000/Sx, communications are through the Xnet and the router. In the case of *router* communication, the communication time depends on the number of collisions in the communication paths. When we compute λ for mapping onto this machine, we assume that all communication takes place exclusively through the Xnet. But in practice, this is not possible due to the irregular nature of the problem. So the entire communication has to be divided between the Xnet and the router. So, a reduced λ, which is computed on the basis of exclusive Xnet communication, may not always give a reduced time in the solver. Still, a measurement of λ gives some idea about the closeness of the mapped nodes and neighbors, and hence it can suitably predict the quality.

Taking the above issues into consideration, we adopt the following criteria for a

quality mapping: *a reduced λ together with a well balanced load on all the processors.* Presently, we don't take the issue of collisions into consideration.

All the general strategies designed were implemented and tested on DECmpp12000/Sx. They were compared in the following way.

Whenever we have to compare two different mapping strategies, we take the same problem graph and then apply both strategies to map this problem graph to the system graph under identical conditions. The mappings are such that a well balanced load is maintained on all the processors. Then we compare the corresponding λs. The mapping strategy giving the lower value of λ consistently for several graphs is definitely the better strategy. We also compare the timings in the solver, which are identical for both the mapping strategies. It is found that a lower value of λ with a well balanced load gives better timing, with very few exceptions which can be attributed to collisions in router communication as described before.

Now we are in a position to discuss the different strategies.

4.2 A bad *Random Mapping* strategy

The next important issue is to compare the performance of any mapping heuristic with some other *bad* mapping scheme, which does not follow any heuristic. This way we will be able to know the benefit we derive by applying the heuristic. That *bad* strategy may be the sequential counterpart of the problem, but it is not a sensible idea to compare the performance of two implementations on two different architectures. So, we adopt a *random mapping* strategy, which simply scans through the nodes and then maps nodes to the processors in a random fashion. Since any node can go to any processor depending on the order it is entered, it is in a sense *random* and it has a tendency to completely destroy the locality of the nodes in the problem graph. It simply goes as follows.

Let N be the total number nodes in the problem graph, and let n_{proc} be the total number of processors. Let us also assume that the processors are numbered from 0 to

$(nproc-1)$ in any arbitrary fashion. Then $n = \lceil \frac{N}{n_{proc}} \rceil$ is the average number of nodes that should go to each processor. Nodes $0..(n-1)$ are mapped to processor 0, nodes $n..(2n-1)$ are mapped to processor 1, and so on. In general, nodes $kn..((k+1)n-1)$ are mapped to processor k, where $k = 0..(\lfloor \frac{N}{n} \rfloor - 1)$. Fewer than n nodes will go to the remaining processors depending on whether N is a proper multiple of n_{proc} or not. Everything else is the same as in the other heuristic. Then we compare the results, i.e. the measures of λ as well the computational time in the solver with experimental domains. The results for DECmpp12000/Sx are illustrated in a subsequent chapter.

4.3 A *Greedy close-neighbor* heuristic

We start with a naive algorithm to map the nodes to the processors. It is based on the simple requirement that whenever we map a node, we see that it is mapped as close as possible to its neighbors. First we give the *restricted* algorithm, which assumes that the number of problem nodes is at most equal to the number of processors. Subsequently we generalize it for any number of problem nodes. We discuss the heuristic next.

4.3.1 The restricted algorithm

In the following discussion, we use the term *close vicinity* of any processor v_{sk} in the system graph. In simple terms, it implies any available neighboring processor of v_{sk}, to which at least one more node can be mapped. We can have different criteria to find this *close vicinity*, depending on the processor interconnection topology. We first describe the algorithm before elaborating it further:

1. Reset *active* flag on all processors.
2. **Repeat**
   2.1. for each node v_{pi} in V_P
   do

2.1.1. if v_{pi} is the first node to be mapped
then
2.1.1.1. map it arbitrarily to any processor v_{sk} in V_S.
2.1.1.2. set *active* flag on v_{sk}.
2.1.1.3. for each unmapped neighbor v_{pj} of v_{pi}
do
2.1.1.3.1. map v_{pj} to some *inactive* processor v_{sl}
in *close vicinity* of v_{sk}.
2.1.1.3.2. set *active* flag on v_{sl}.
od
else
2.1.1.4. if v_{pi} is already mapped to some processor
v_{sm}(as neighbor of some other node)
then
2.1.1.4.1. *for* each unmapped neighbor v_{pj} of v_{pi}
do
2.1.1.4.1.1. map v_{pj} to some inactive
processor v_{sn} in *close vicinity* of v_{sm}.
2.1.1.4.1.2. set *active* flag on v_{sn}.
od
else
2.1.1.4.2. if some of its neighbor v_{pk} is already
mapped to some processor v_{sp}
then
2.1.1.4.2.1. map v_{pi} to some processor v_{sq} in
close vicinity of v_{sp}
2.1.1.4.2.2. set *active* flag on v_{sq}.
2.1.1.4.2.3. for each unmapped neighbor v_{pj}
of v_{pi}
do
2.1.1.4.2.3.1. map v_{pj} to some inactive

processor v_{sr} in *close vicinity* of v_{sq}.
2.1.1.4.2.3.2. set *active* flag on v_{sr}.
od
else
(don't map it now)
od
until all nodes in V_P are mapped.

All nodes should be mapped in a single iteration of the *Repeat* loop, provided they are sorted in proper order. This is found to be the general requirement for all experiments conducted.

Assuming that the number of nodes is less than or equal to the number of processors, at most one node should be mapped per processor. We declare a parallel flag called *active* on each processor. Initially, all processors are made *inactive*, i.e. their *active* flags are reset. As soon as a node is mapped to a processor, it's *active* flag is set. We can have different criteria in selecting an available *inactive* processor in the *close vicinity* depending on the processor interconnection topology.

For instance, in case of DECmpp12000/Sx, which is a two dimensional mesh with toroidal wraparound, each processor has eight immediate neighbors at a distance one, sixteen neighbors at a distance two, and so on. Whenever we search the *close vicinity* of a processor for an *inactive* neighboring processor, we first search the eight immediate neighbors at distance one in a clockwise/counterclockwise fashion, then the sixteen neighbors at distance two, and so on. The search continues until we can find the first available processor. The node is mapped to this processor, and its *active* flag is set. Since our algorithm grabs the first available processor without thinking about the future, it is *greedy*.

4.3.2 The generalized algorithm

To generalize the above algorithm, we should have a previous knowledge of the total number of nodes and the total number of processors so that we can find the average

number of nodes that should go to each processor. Let this average be n, as computed in the previous section. The changes in the implementation are that, first each processor has to be partitioned to hold more than one node and second, there is a different strategy in finding the next available processor in the *close vicinity*. Now, we do not use an *active* flag. Instead, a processor is selected for mapping if the total number of nodes already mapped to it is less than the average n. The rest of the procedure remains unaltered. We try to maintain a fair load on all the processors by not allowing to map more than the average number of nodes that should go to each processor.

Each mapped node should contain information about where (i.e. processor id and partition index on that processor) all its neighbors are mapped so that it can communicate with them in the parallel solver. This information is set up in the same scan or in a subsequent scan through the nodes in V_P. We discuss about the implementation details in a subsequent chapter.

4.3.3 Setting up communication

One important issue is to set up communication between mapped nodes and neighbors in the parallel solver part. As mentioned earlier, each mapped node should contain information about where(i.e. processor id and partition index on that processor) all its neighbors are mapped so that it can communicate with them. In the parallel solver, computation and communication proceeds in a data parallel way over a fixed number of iterations, which is large enough so that computed values are guaranteed to converge. We keep this number of iterations fixed so that we can compare the performance of different mapping strategies.

Setting up communication primitives depends entirely on the underlying machine, but the basic idea is the same. We slightly elaborate this communication issue for our implementation on DECmpp12000/Sx.

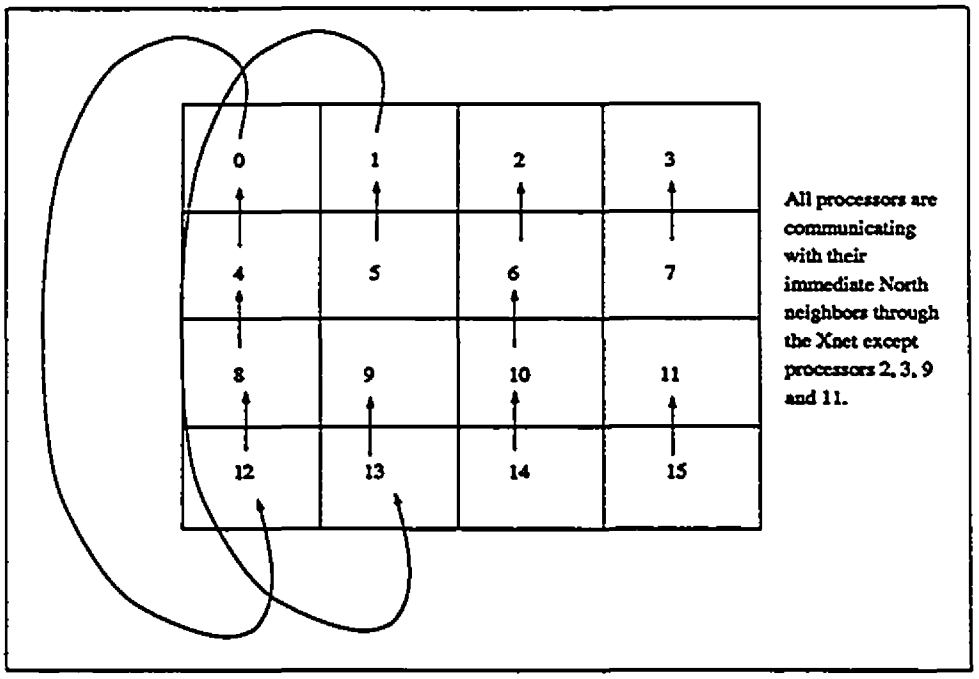


Figure 4.1: Synchronous Xnet communication

There are two types of communication in DECmpp12000/Sx, one is the asynchronous *router* communication, and the other is the synchronous *Xnet* communication. The Xnet is much faster as compared with the router. But in using the Xnet, at any instant of the global clock, all *communicating* processors have to communicate with their neighbors in an identical way, i.e. in the same direction and distance. We can keep some parallel flag so that a processor communicates with its neighbor if and only if its corresponding flag is set, i.e. it is a *communicating* processor at that instant. It should be noted that, in the case of an irregular problem, at some instant some processor might want to communicate with some neighboring processor in some specific direction and distance, but another processor might have to remain idle because none of the neighboring problem nodes are mapped to its corresponding neighboring processor. This is illustrated in *Fig. 4.1.*

In the above figure, at some instant of the Xnet communication with the immediate North neighbors, processors v_{s2}, v_{s3}, v_{s9} and v_{s11} are idle. Processor v_{s9} is made idle at that instant by proper use of some flag because (i)either none of the neighbors of

the problem node v_{pi} mapped to processor v_{s9} are mapped to its immediate North neighbor v_{s5} or (ii)no node is mapped to processor v_{s9}. For convenience, we have assumed that at most one node is mapped per processor. The same holds true for the other 3 processors. Thus, it is evident that an exclusive use of the Xnet might degrade performance.

Instead, we do a compromise between the Xnet and router communication. We divide the entire communication pattern into two parts, one through the Xnet and the other through the router. The Xnet communication is restricted only to a distance of one, because anything more than one is found to degrade performance. The rest of the communication is done through the router.

4.3.4 Limitations

One obvious limitation of the above scheme is that it is *greedy*, and hence the mapping obtained and the corresponding λ are not necessarily optimal. It is clear that some nodes and neighbors in the discretization may not remain neighbors or even remain close to one another after being mapped. This is because we have destroyed locality by transforming a multi dimensional problem graph into a one dimensional mapping problem. The strategy has to be modified so that locality is not destroyed.

In the next chapter, we discuss quad-tree based and binary decomposition based mapping strategies for 2-dimensional problem graphs. They can be generalized for any arbitrary d dimension.

4.4 A *layer by layer* scheme

In the *layer by layer* scheme, we pick up an arbitrary node and put it in a queue. Then we pick up all its neighboring nodes and queue them. We proceed along the queue, and keep on queueing neighbors until all nodes are exhausted. Thus we proceed in a layer-by-layer fashion, as illustrated in *Fig. 4.2*, starting with an arbitrary node

which is the *layer 0* node. Its immediate neighbors constitute the *layer 1* nodes, the neighbors of the *layer 1* nodes constitute the *layer 2* nodes, and this way the layering continues in a breadth first fashion. It can be put in the following form:

1. Pick up an arbitrary node v_{pi} from V_P.
2. Queue v_{pi} and make *next-queue* pointer point to it.
3. Assign *layer* of $v_{pi} = 0$.
4. **Repeat**
   - 4.1. Pick up next node v_{pj} pointed by *next-queue* pointer.
   - 4.2. For each unqueued neighbor v_{pk} of v_{pj}

   do
     - 4.2.1. Queue v_{pk}.
     - 4.2.2. Assign layer of v_{pk} = layer of $v_{pj} + 1$.

   od
   - 4.3. Advance *next-queue* pointer.

   until end of queue(*next-queue* pointer points to *NULL*).

After arranging the nodes in this layer by layer fashion, we map them in such a way that the adjacent layers are mapped close to one another, i.e. *layer i* nodes are adjacent to *layer (i-1)* nodes as well as *layer (i+1)* nodes.

For this, we compute the average n of nodes that should go to each processor, as before. Then we scan along the queue. The first n nodes in the queue are mapped to an arbitrary processor v_{si}, the next n nodes are mapped to an neighboring processor v_{sj} of v_{si}, the subsequent n nodes are mapped to a processor v_{sk} adjacent to both v_{si} and v_{sj}. In general, let v_{sk} denote the processor where nodes $kn..(k+1)n-1$ are mapped. The numbering of nodes starts with 0 at the head of the queue. Then, processor v_{s0} is selected arbitrarily. v_{s1} is selected as a neighboring processor of v_{s0}. Processor v_{sl} is selected adjacent to both processors $v_{s(l-1)}$ and $v_{s(l-2)}$ for $l = 2..(\lfloor \frac{N}{n} \rfloor - 1)$.

Obviously it is not a comfortable mapping scheme, because it has a tendency to destroy locality of the nodes as in the previous scheme. However, our experiments on

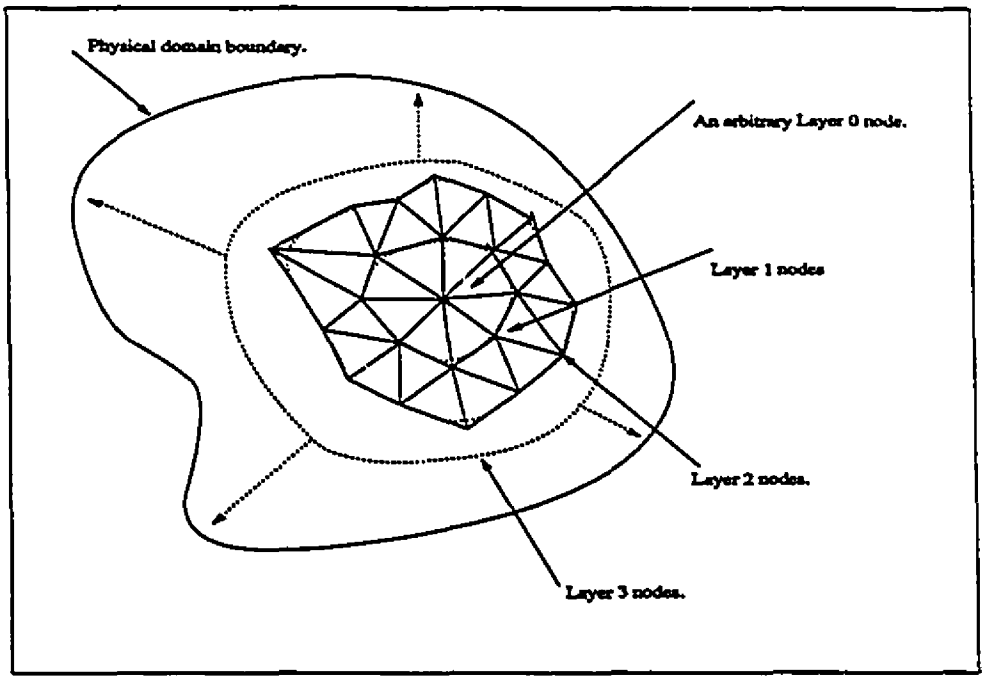


Figure 4.2: Layering of nodes

DECmpp12000/Sx show it to be a better scheme than the *random mapping* scheme. The results are illustrated in a subsequent chapter.

Chapter 5

A Quad-Tree based mapping strategy

We initially discuss a quad-tree based mapping strategy based on geometrical co-ordinates of the nodes. In this scheme, we try to cluster geometrically close nodes together and map them to the same or neighboring processors. We presume that maintaining geometrical closeness will automatically preserve the topology, which comes from the node-neighbor relationship in the discretization. It is not necessarily true, because two geometrically close nodes may not be neighbors in the actual discretization. We try to resolve this problem in subsequent modifications. In the following discussion, we will assume a 2-dimensional problem graph.

The basic idea is to partition nodes to different branches of a quad-tree based on their geometrical co-ordinates. The root of the quad-tree contains all the nodes of the problem graph. Then they are routed to the four branches of the quad-tree based on their X and Y co-ordinate values. We call each intermediate node of the quad-tree as a quadrant. This partitioning process continues recursively with nodes belonging to each of the quadrants, and stops at a depth which we are going to discuss shortly. Each leaf of the quad-tree contains a bunch of nodes geometrically close to one another. The mapping problem now focuses on mapping these leaves to

the processors in the system graph so that neighboring leaves are not separated by a large distance. It works as follows:

- Step 1: Find the four extreme co-ordinates of the nodes, i.e. leftmost X co-ordinate x_{left}, rightmost X co-ordinate x_{right}, topmost Y co-ordinate y_{top}, and bottommost Y co-ordinate y_{bottom}. Construct a rectangle through the corner nodes (x_{left}, y_{top}), (x_{left}, y_{bottom}), (x_{right}, y_{top}), and (x_{right}, y_{bottom}). All nodes fall on or within this rectangle. This rectangle corresponds to the root of the quad-tree.

- Step 2: Find the average co-ordinates $x_{av} = (x_{left} + x_{right})/2$ and $y_{av} = (y_{top} + y_{bottom})/2$, and then divide the rectangle into four equal quadrants by drawing lines through these averages. These quadrants are numbered from 0 to 3 as illustrated in *Fig. 5.1.* They correspond to the four branches of the corresponding quad-tree at level 1. Problem nodes are partitioned to different quadrants based on their X and Y co-ordinate values. For instance, a node v_{pi} with co-ordinate (x, y) will be routed to quadrant 0 if $x \geq x_{av}$ and $y \geq y_{av}$. Similarly, it will be routed to quadrant 1 if $x \leq x_{av}$ and $y \geq y_{av}$, and so on. Note that each quadrant will contain a variable number of nodes, i.e. load is not balanced across quadrants. From now on, we will use the terms *branch of a quad-tree*, and *quadrant* interchangeably. We call each problem graph quadrant simply as a *problem quadrant.*

  Simultaneously partition the system graph into four equal *system quadrants.* Conveniently map each *problem quadrant* to a *system quadrant* so that neighboring *problem quadrants* do not go far. In practice, each *problem quadrant* is assigned an id which uniquely identifies the *system quadrant* to which it is mapped.

- Step 3: For the purpose of load-balancing among the processors, adjust the sizes of the quadrants, i.e. exchange nodes among the quadrants *respecting geometrical closeness of the nodes*, so that almost an equal number of nodes

goes to each quadrant. This can be achieved as follows. First exchange nodes between quadrants 0 and 3, and quadrants 1 and 2. This can be termed as the *exchange along the Y dimension.* Then exchange nodes between quadrants 0 and 1, and quadrants 2 and 3. This is the *exchange along the X dimension.* Thus each quadrant will finally contain an almost equal number of nodes.

- Step 4: Recursively follow steps 1..4 with each of the four *problem quadrants* and the corresponding *system quadrants.*
- Step 5: Stop when the depth of recursion is $k-1$, where $k = \log_4(no\_of\_processors)$. We assume that the *no_of_processors* is such that we get an integral value for k. Note that this k is the height of the corresponding quad-tree.

At the bottom level of the quad-tree, we have a number of leaves, each containing 0 or more nodes. We assume that the number of leaves is equal to the number of processors. Each leaf is assigned with an id, which uniquely identifies a processor to which all nodes belonging to this leaf are to be mapped.

A simple mapping can be on a two-dimensional square mesh, as in the case of DECmpp12000/Sx. In the next section, we discuss a quad-tree based mapping strategy for such a square mesh. In case the mesh is not a square, the length of the longer dimension is an integral multiple of the length of the shorter dimension. Hence, we can divide the mesh into an integral number of square meshes, and then apply the same algorithm with some modifications.

5.1 Implementation on DECmpp12000/Sx

In a DECmpp12000/Sx processor mesh, processors are numbered as in *Fig. 5.2.* We design and implement a quad-tree based mapping strategy for this machine. The algorithm is illustrated below in a C like pseudo code. As we can see, at each level of recursion, each quadrant is labeled with an *id*, called *quadn-id* for $n = 0..3$, which

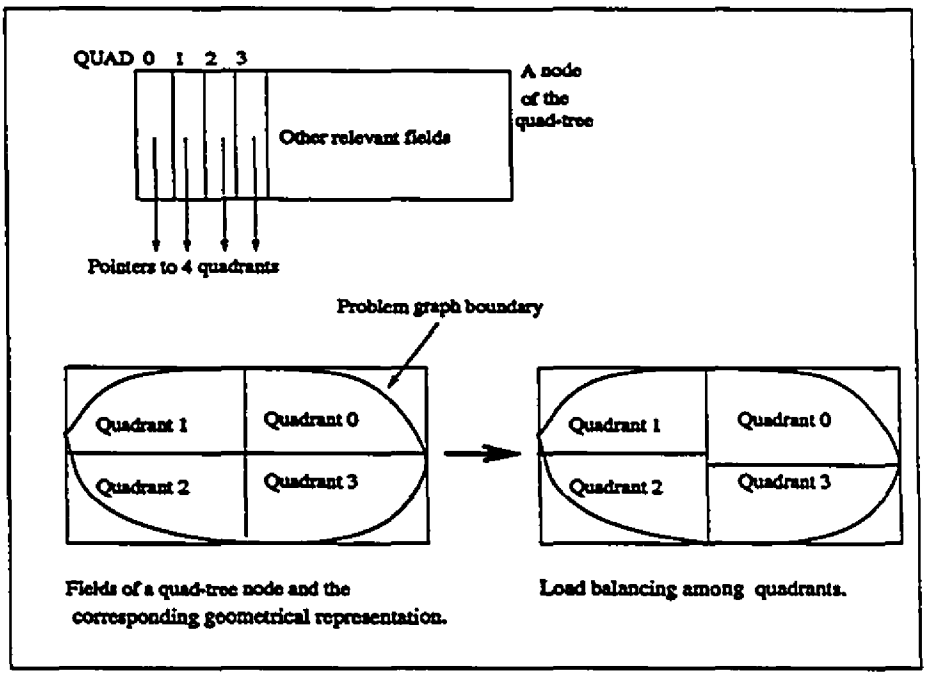


Figure 5.1: A Quad-tree based strategy

is passed as a parameter. An *id* associated with a quadrant identifies the processor partition to which it is to be mapped. At the lowest level of recursion, when we reach the leaves, these *ids* become the corresponding processor numbers. All nodes belonging to a leaf are mapped to the corresponding processor. Below we summarize the implementation in a C like pseudo code:

```
/* The following algorithm is applicable only when log4(no_of_processors)
   is a proper integer, and the processor mesh is a perfect square.
*/
quad-tree *build-tree(original-problem-node-list, level, id, x, y)
begin
    /* Some variable declarations */
    quad-tree *new-node;
    problem-node-list *quad0-list, *quad1-list;
    problem-node-list *quad2-list, *quad3-list;
```

```
int quad0-id, quad1-id;
int quad2-id, quad3-id;
int k;
.
.
/* And other declarations */
.
.
k = log4(no_of_processors); /* An integral value */
new-node:=(quad-tree *) create-new-node-for-quad-tree();
if (level=k) then /* We have reached a leaf of the tree */
begin
   step 1:
      new-node->problem-node-list:=original-problem-node-list;
   step 2:
      Assign id to all problem nodes in original-problem-node-list;
   step 3:
      new-node->quad0:=NULL;
      new-node->quad1:=NULL;
      new-node->quad2:=NULL;
      new-node->quad3:=NULL;
   step 4:
      Set appropriate values to other fields of new-node;
   Step 5:
      return(new-node);
end
else
begin
   step 1:
      x:=x/2;
      y:=y/2;
```

```
        step 2:
            quad0-id:=id + x;
            quad1-id:=id;
            quad2-id:=id + y;
            quad3-id:=id + x + y;
        step 3:
            /* The following lists are sorted as described in next
               subsection */
            quad0-list:=the list of nodes from original-problem-node-list which
                        go to quadrant 0;
            quad1-list:=...
            .
            .
        step 4:
            /* The following is for load-balancing, and is discussed
               in the next subsection.*/
            Exchange problem nodes among quadrants in such a
            way that each quadrant contains almost an equal number
            of them.
        step 5:
            new-node->quad0:=build-tree(quad0-list, level + 1,
                        quad0-id, x, y);
            new-node->quad1:=build-tree(quad1-list, level + 1,
                        quad1-id, x, y);
            .
            .
        step 6:
            return(new-node);
    end; /* Of else part */
end; /* Of build-tree() */
```

```
main() /* Call from the main routine */
begin
        int x, y;
        quad-tree *root;
        .
        .
        /* nxproc = number of processors in X-direction */
        /* nyproc = number of processors in Y-direction */
        .
        .
        if ((nxproc = nyproc) and (log4(no_of_processors) is a proper
                        integer)) then
        begin
           x:=nxproc;
           y:=nxproc * nyproc;
           root:=build-tree(original-problem-node-list, 0, 0, x, y);
        end;
        else
        begin
           /* Some other scheme to be designed */
        end;
end. /* Of Main */
```

Until now, we have assumed that the processor mesh is a perfect square and $k = \log_4(no\_of\_processors)$ is a proper integer. The next step may be to design a general algorithm with a proper numbering scheme that can be applied to any processor mesh. In DECmpp12000/Sx, either $nxproc = nyproc$ or $nxproc = 2 * nyproc$. We can think of the following possibilities:

- When $\log_4(no\_of\_processors)$ is not an integral value, we can add some *dummy* processors in order to make it a perfect integer, and then redesign the above

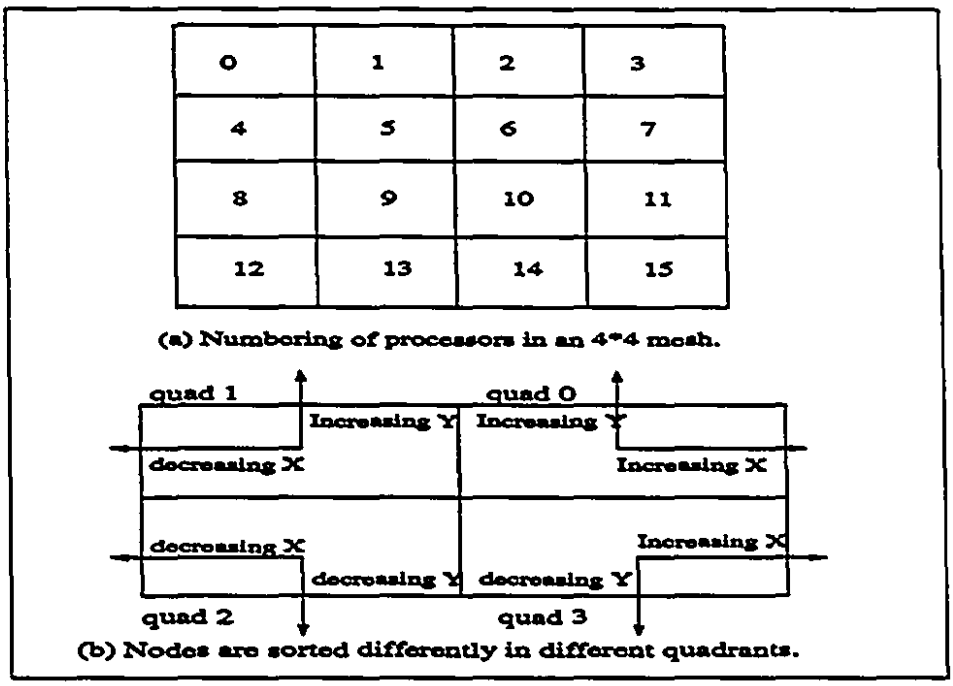


Figure 5.2: An implementation on DECmpp12000/Sx

scheme so that it takes care of the *dummy* processors, and does not assign any leaf to them.

- Another possibility may be that we take a suitable subset of the processor mesh to make k integral. In this case, some processors will not be assigned any nodes, i.e.they will be *inactive*. It can be taken care of, if after the initial assignment we perform a second phase of redistribution to move some of the problem nodes to the *inactive* processors.

- When $nyproc = m * nxproc$, or vice versa for some integer m, we can divide the processor mesh into m equal sub-meshes, each of size $nxprox * (nyproc/m)$. Similarly, we can divide the problem graph into m sub-graphs, each containing almost an equal number of nodes to take care of load-balancing, and then apply the same algorithm to map a sub-graph to a sub-mesh.

5.1.1 Load Balancing

In *step 4* of the above implementation, we mentioned exchanging nodes amongst quadrants so that almost an equal number of nodes goes to each quadrant. Whenever nodes are exchanged amongst quadrants, the geometry of the problem graph has to be respected so that geometrical locality is not destroyed, i.e. nodes cannot be exchanged arbitrarily. In order to achieve this, nodes are sorted into X and Y sorted lists in each quadrant depending on their geometrical X and Y co-ordinate values, instead of sorting them into just one list called *quadn-list*, $n = 0..3$, as described in *step 3* of the algorithm.

In each quadrant, nodes are sorted in different ways. For instance, in quadrant 0 they are sorted in increasing X and Y sorted lists, while in quadrant 1 they are sorted in decreasing X and increasing Y sorted lists. This is illustrated in *Fig. 5.2.* The reason for this will be clear when we visualize the exchange of nodes between any two adjacent non-diagonal quadrants at any level of recursion.

For instance, let us suppose that we have to move some nodes from quadrant 1 to quadrant 0 so that both quadrants contain an equal number of nodes. This can be achieved simply by scanning through the decreasing X sorted list of quadrant 1, removing nodes from its head, and adding them to the head of increasing X sorted list in quadrant 0. Whenever a node is removed from the X sorted list in quadrant 1, it is also removed from the Y sorted list in the same quadrant. To achieve this without adding any extra complexity in the search process, we maintain a *side link* between the two lists so that following this link from one list, we can obtain the position of the node in the other list. Similarly, whenever we add a node in the head of the increasing X sorted list in quadrant 0, we add it in the appropriate position of the Y sorted list for the same quadrant. In this case, we have to search through the Y sorted list from the head, until we reach the appropriate position.

5.1.2 Limitations

The above scheme has a number of limitations which we rectify in subsequent modifications. One limitation is that, we have used geometrical co-ordinates of the nodes for partitioning them to different quadrants. We have assumed that maintaining geometrical proximity of nodes will automatically preserve their topology, which comes from their node-neighbor relationship in the discretization. But this is not necessarily true. This is because two nodes may be geometrically close to each other, but they may not be neighbors in the actual discretization. In some cases, two nodes may be geometrically far off, but they may be neighbors in the actual discretization.

To handle this, we device at least two schemes to assign topological co-ordinates to nodes, and then apply the same quad-tree based mapping strategy to partition them to quadrants based on their topological co-ordinates. We discuss it in a subsequent section.

Another limitation is that, while routing a node to a particular quadrant, we have to insert it in the proper position in a sorted list. We use insertion sort for this purpose, which has a time complexity of $O(N^2)$, where N is the number of nodes in the problem graph. This is quite high when the number of nodes is very large. We modify the algorithm so that we can use some other less expensive sorting techniques like merge sort instead of using the insertion sort. It improves the time complexity of the sorting to $O(Nlog_2N)$. In this modified algorithm, the number of processors has to be an integral power of 2, instead of being an integral power of 4 as in this case. We discuss it next.

5.1.3 A binary-decomposition based scheme

We implement a binary-decomposition scheme, somewhat similar to the one described in [10]. The problem graph is alternately divided in horizontal and vertical partitions in a recursive manner, while load-balancing is taken into account. This recursive

partitioning process stops when the number of leaves is equal to the number of processors, assuming that the number of processors is an integral power of 2. If k is the height of the corresponding binary tree, then $k = log_2(no\_of\_processors)$.

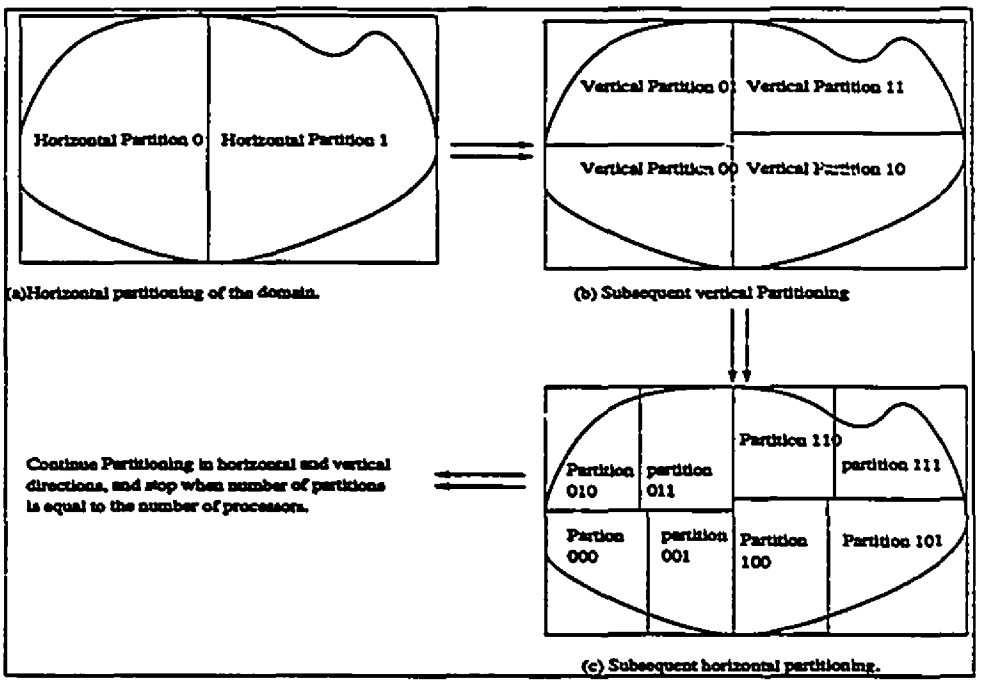


Figure 5.3: A binary-decomposition based scheme

This is illustrated in *Fig. 5.3*. The problem graph is first divided by a vertical line into horizontal partitions 0 and 1. Each of these two partitions contains an approximately equal number of nodes. Then, each of these two partitions is divided into equally weighed vertical partitions by horizontal lines, these partitions are numbered 00, 01 and 10, 11 for partitions 0 and 1 respectively. This process of binary-decomposition of the problem graph continues recursively and stops when the number of leaves is equal to the number of processors. The mapping problem now focuses on mapping the leaves, each containing 0 or more nodes, to the processors. It works as follows:

- Step 1: Assign $flag = 0$.
- Step 2: If *flag* is even then sort all nodes in the X direction, otherwise sort them

in the Y direction into an array. We can use merge sort for this purpose.

- Step 3: Divide this sorted array in two equal partitions by dividing it exactly in the middle. So each partition contains an equal number of nodes. We name these two partitions as *problem partitions.*

  Simultaneously partition the system graph into two equal and adjacent *system partitions.* Map each *problem partition* to a *system partition.* In practice, each *problem partition* is assigned an id which uniquely identifies a *system partition* to which it is mapped.

- Step 4: Assign $flag = flag + 1$.

- Step 5: Repeat steps 2..5 recursively with nodes belonging to each *problem partition.* Also simultaneously divide the corresponding *system partitions.*

- Step 6: Stop when the depth of recursion is $k-1$, where $k = log_2(no\_of\_processors)$.

Thus, as before, each leaf is assigned an id which uniquely identifies the processor to which it is to be mapped. All nodes belonging to that leaf are mapped to that processor.

Speedup in pre-processing time over the previous quad-tree based mapping strategy is one obvious advantage of this scheme. Another advantage is that, in this case the number of processors has to be an integral power of 2, which is a more general case than being an integral power of 4 as in the previous scheme. Both these schemes do partition nodes based on their geometrical co-ordinates, which need not always preserve the topology of the problem graph.

5.1.4 An implementation for DECmpp12000/Sx

Below we describe a binary-decomposition based mapping scheme for DECmpp12000/Sx in a C like pseudo code:

```
/* The following routine is applicable only when log2(no_of_processors)
   is a proper integer, and either the processor mesh is a perfect square
   or nxproc = 2*nyproc, where nxproc and nyproc are number columns
   and rows respectively in the processor mesh.
*/
bin-tree *build-tree(original-problem-node-list, level, id, flag, x, y)
begin
    /* Some variable declarations */
    bin-tree *new-node;
    problem-node-list *partition-0-list, *partition-1-list;
    int part0-id, part1-id;
    int k;
    .
    .
    /* And other declarations */
    .
    .
    k = log2(no_of_processors); /* An integral value */
    new-node:=(bin-tree *) create-new-node-for-bin-tree();
    if (level=k) then /* We have reached a leaf of the tree */
    begin
      step 1:
        new-node->problem-node-list:=original-problem-node-list;
      step 2:
        Assign id to all problem nodes in original-problem-node-list;
      step 3:
        new-node->partition-0:=NULL;
        new-node->partition-1:=NULL;
      step 4:
        Set appropriate values to other fields of new-node;
```

```
    Step 5:
      return(new-node);
end
else
begin
    Step 1:
      flag:=flag mod 2;
    if (flag = 0) then /* Divide into horizontal partitions */
    begin
      Step 2:
        Sort nodes in original-problem-node-list in increasing X-direction.
      Step 3:
        partition-0-list:=first half of sorted list;
        partition-1-list:=second half of sorted list;
      Step 4:
        x:=x/2;
      Step 5:
        part0-id:=id;
        part1-id:=id + x;
    end
    else /* Divide into vertical partitions */
    begin
      Step 2:
        Sort nodes in original-problem-node-list in increasing Y-direction.
      Step 3:
        partition-0-list:=first half of sorted list;
        partition-1-list:=second half of sorted list;
      Step 4:
        y:=y/2;
      Step 5:
        part0-id:=id + y;
```

```
            part1-id:=id;
        end
        Step 6:
            new-node->partition-0:=build-tree(partition-0-list, level+1,
                        part0-id, flag + 1, x, y);
            new-node->partition-1:=build-tree(partition-1-list, level+1,
                        part1-id, flag + 1, x, y);
        step 7:
            return(new-node);
    end; /* Of else part */
end; /* Of build-tree() */

main() /* Call from the main routine */
begin
        int x, y;
        bin-tree *root;
        .
        .
        /* nxproc = number of processors in X-direction */
        /* nyproc = number of processors in Y-direction */
        .
        .
        if (((nxproc = nyproc) or (nxproc = 2*nyproc) and
            (log2(no_of_processors) is a proper integer)) then
        begin
            x:=nxproc;
            y:=nxproc * nyproc;
            root:=build-tree(original-problem-node-list, 0, 0, 0, x, y);
        end;
        else
```

```
        begin
          /* Some other scheme to be designed */
        end;
    end. /* Of Main */
```

Here, *partn-id*, for $n = 0..1$, specifies the processor partition where a particular problem graph partition is to be mapped. At the bottommost level of recursion, this id identifies a unique processor to which a set of nodes, belonging to *original-problem-node-list*, are to be mapped. This is done in *Step 2* of the *then* part of the outermost *if* statement.

Speedup over the previous quad-tree based mapping strategy is one obvious advantage of this scheme. Another advantage is that, in this case the number of processors has to be an integral power of 2, which is a more general case than being an integral power of 4 as in the previous scheme. Both these schemes do partitioning of nodes based on their geometrical co-ordinates, which need not always preserve the topology of the problem graph.

In the next section, we will discuss a number of schemes for assigning topological co-ordinates to nodes.

5.2 Assigning topological co-ordinates to nodes

Topological co-ordinates are assigned to nodes for the purpose of capturing the topology information of the problem graph, which comes from the node-neighbor interconnection. After assigning topological co-ordinates, we can apply the same quad-tree and binary-decomposition based mapping strategies, as discussed above, for partitioning nodes to processors.

Topological co-ordinates are assigned to nodes, based on their topological distances from certain reference sets(s). We design and implement at least two different schemes

for assigning topological co-ordinates to nodes. We fix one or more reference sets, and then assign co-ordinates to all nodes relative to nodes in these sets. First we discuss a scheme based on one reference set, and subsequently rectify its limitations in a modified scheme.

5.2.1 A *one reference set* based scheme

We consider a 2-D, closed and connected domain as in *Fig. 5.4*. After performing domain discretization, we construct a reference set consisting of some connected boundary nodes. We call this set R_B. Let V_B denote the set of all boundary nodes. Other terms are already defined in *chapter 4*. The set R_B satisfies the following properties:

1. $|R_B| \geq 2$.
2. $v_{pi} \in R_B \Rightarrow v_{pi} \in V_B$.
3. $\forall v_{pi} \in R_B$, $\exists v_{pj} \in R_B$ such that $D_{pij} = 1$.
4. We call a node $v_{pi} \in R_B$ an *extreme node* iff there exists one and only one node $v_{pj} \in R_B$ so that $D_{pij} = 1$. There exists at most two such *extreme nodes* in R_B.

It would be clear that for each node $v_{pi} \in R_B$, there can be at most two other nodes satisfying *property 3*. Together with *property 4*, it ensures the continuity of the nodes in the set. We illustrate such a reference set in *Fig. 5.4*. Two extreme nodes v_{pl_0} and v_{pl_1} are shown in the same figure. Note that if $R_B = V_B$ then it does not contain any *extreme nodes*.

Though there is no fixed rule regarding how to select this set and how big it should be, we followed the following criterion for our implementation on DECmpp12000/Sx. We first compute the average number of nodes that should go to each processor, let it be n. We start with an arbitrary boundary node, which becomes the first element to enter set R_B. We then keep on adding nodes to R_B satisfying the above properties

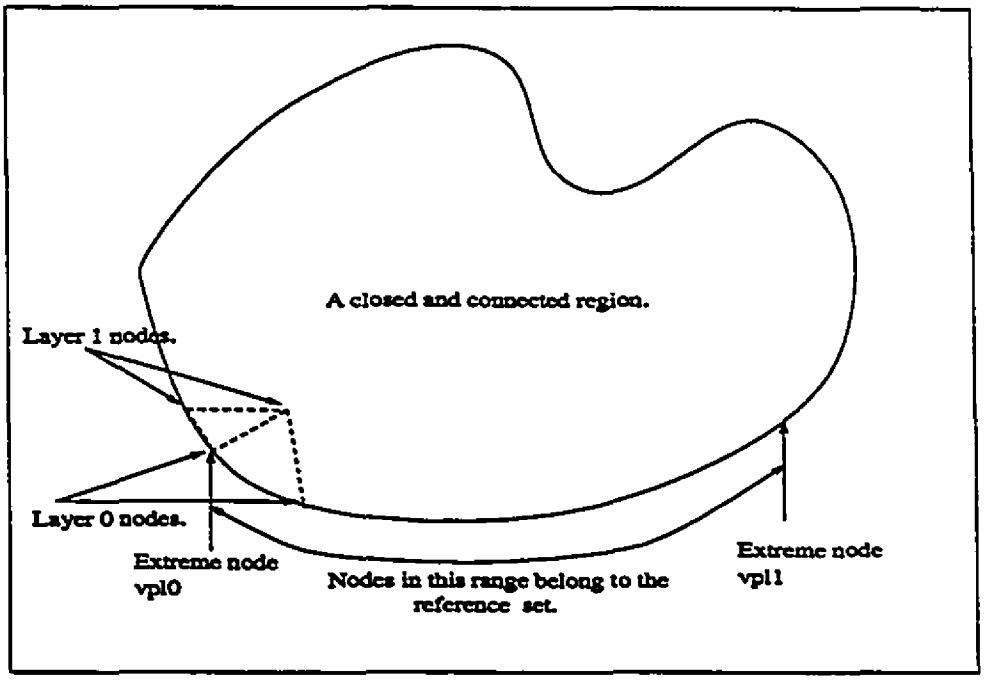


Figure 5.4: Assigning topological co-ordinates to nodes

until either all the boundary nodes are exhausted or the number of nodes in the set is exactly $n * nxproc$, where $nxproc$ is the number of columns in the processor array.

The next issue is to assign topological co-ordinates to nodes relative to this reference set. We follow the following steps:

Step 1:

Arrange all the nodes in R_B along the X axis, i.e. they are assigned Y co-ordinate=0.

Step 2:

case 1:$R_B \neq V_B$

1. Select one *extreme node* from R_B, queue it, assign its X co-ordinate=0, and make *next-queue pointer* point to this first and only entry in the queue.
2. *Repeat*

2.1. Pick the next node v_{pi} from queue as pointed by *next-queue pointer.*

2.2. Select $v_{pj} \in R_B$ so that $D_{pij} = 1$.

2.3. Queue v_{pj}, and assign X co-ordinate of v_{pj}=X co-ordinate of $v_{pi} + 1$.

2.4. Advance *next-queue pointer.*

until all nodes in R_B are queued.

case 2:$R_B = V_B$.

1. Select any arbitrary node from R_B, queue it, assign its X co-ordinate=0, and make *next-queue pointer* point to this first and only entry in queue.
2. Follow the same steps in 2 as in the previous case.

Step 3:

1. Make *next-queue pointer* point to the head of the queue.
2. *Repeat*

2.1. Pick the next node v_{pi} from the queue as pointed by *next-queue pointer.*

2.2. Let $v_{pi_1}, .., v_{pi_n}$ be its n neighbors so that $D_{pii_k} = 1$, $k = 1..n$, and none of them are already queued.

2.3. *for* $k = 1..n$ *do*

begin

2.3.1. Queue v_{pi_k}.

2.3.2. Assign Y co-ordinate of v_{pi_k}=Y co-ordinate of $v_{pi} + 1$.

2.3.3. Assign X co-ordinate of v_{pi_k}=X co-ordinate of $v_{pi} + (k-1)$.

end

2.4. Advance *next-queue pointer.*

until all nodes in queue are exhausted(i.e. *next-queue pointer* points to NULL).

Here, we have performed a breadth first search in assigning topological co-ordinates to nodes. The nodes in the initial set R_B are laid along the X axis and can be termed as the 0_{th} layer nodes. All other nodes which are immediate neighbors of these 0_{th} layer nodes are the 1_{st} layer nodes and are assigned Y co-ordinate 1, and this way the layering continues in a breadth first fashion. X co-ordinates are assigned to nodes based on the X co-ordinates of their immediate neighbors.

The next step is to map these nodes based on their topological co-ordinates. We apply the same quad-tree or binary-decomposition based mapping strategies to partition nodes to different processors as described in the previous section. The only difference is that, this time we use the topological co-ordinates of the nodes, rather than using their geometrical co-ordinates. The results are illustrated in the next chapter.

5.2.2 Limitation

One obvious limitation is that more than one node might get assigned the same topological co-ordinates. In that case, it is not fully successful in preserving the topology information, and hence partitioning of nodes to processors may not always cluster neighboring nodes together. We design another scheme, which should resolve this problem. We discuss it in the next subsection, and name the corresponding co-ordinates as *polar topological co-ordinates* of nodes because of their similarities to the polar co-ordinate system.

5.2.3 Another scheme for assigning topological co-ordinates

In this scheme, nodes are assigned co-ordinates (r, θ), somewhat similar to the polar co-ordinate system. The angle co-ordinate θ is the same angle co-ordinate as in the polar co-ordinate system, and thus it captures the geometrical information of the nodes. The difference is that, the distance co-ordinate r here does not specify the distance from the origin as in the polar system. Instead, it preserves the topological

information of the problem graph, and is assigned in a similar layer-by-layer fashion as described previously. Thus, one co-ordinate captures the geometry and the other captures the topology information of the problem graph. It is illustrated in *Fig. 5.5.*

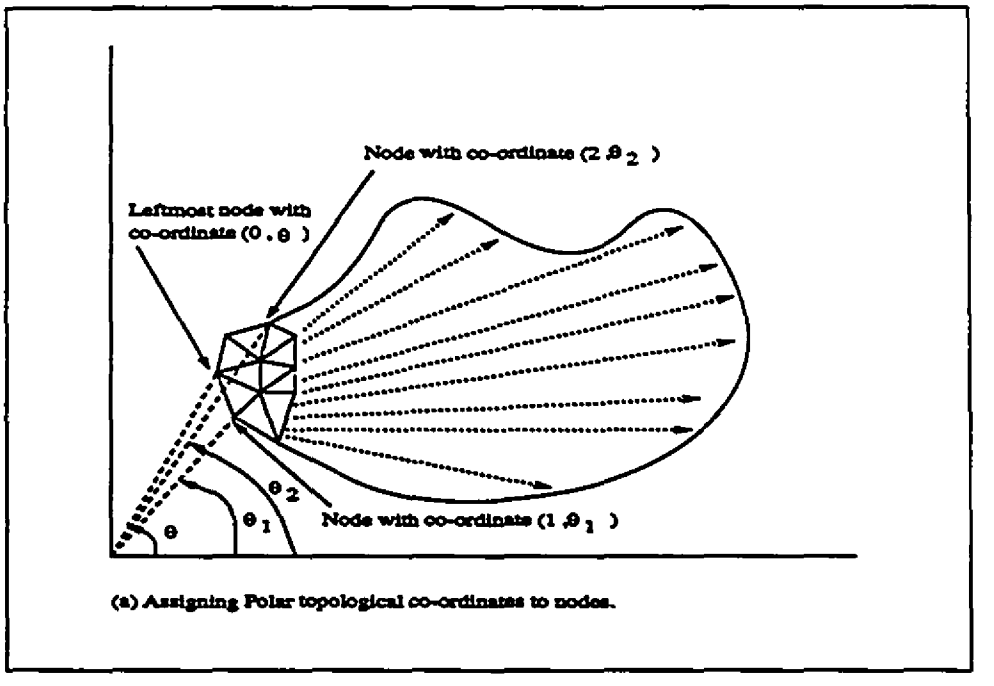


Figure 5.5: Assigning Polar topological co-ordinates to nodes

One possible way in which two nodes might get the same co-ordinates is illustrated in *Fig. 5.6.* In reality, such a situation will rarely occur.

Below, we discuss the scheme in some detail.

1. Select one leftmost node v_{p_i} from problem graph, as in *Fig. 5.5.*
2. Assign its **r co-ordinate** = 0.
3. Assign its θ **co-ordinate** = θ_i, which is the corresponding polar angle.
4. Queue v_{p_i} and make *next-in-queue pointer* point to it.
5. **while** there is some node in queue **do**
   begin
   5.1. Pick up next node v_{pj} as pointed by *next-in-queue pointer.*
   5.2. Let $v_{pj_1}, .., v_{pj_n}$ be its n neighbors so that $D_{pjj_k} = 1$, $k = 1..n$,

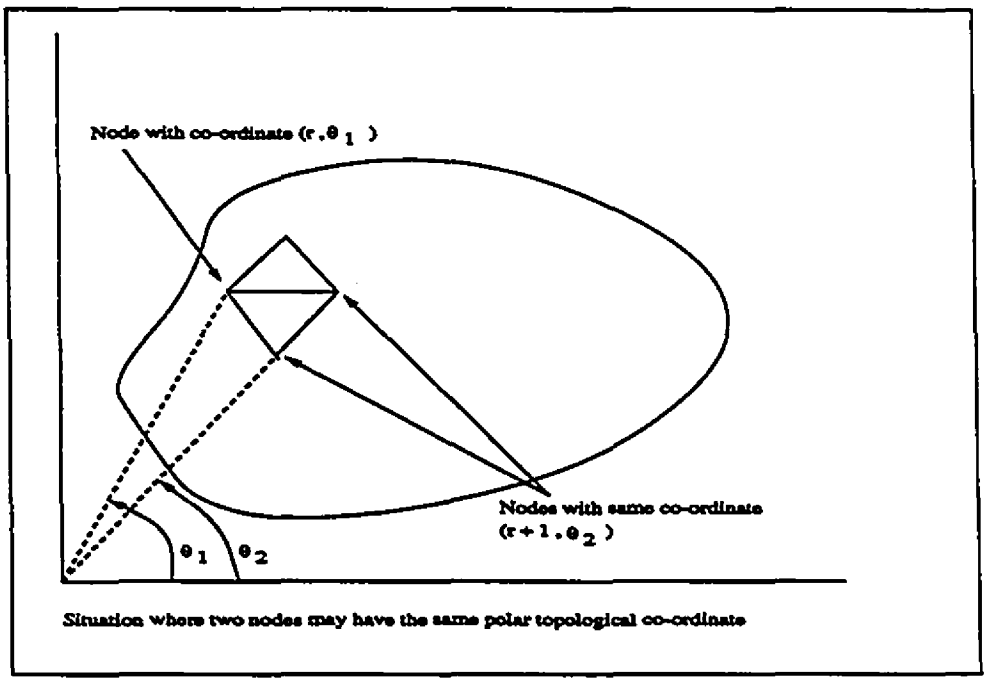


Figure 5.6: A possible conflict

and none of them are already queued.

for $k = 1..n$ **do**

begin

5.2.1. Queue v_{pj_k}.

5.2.2. Assign its **r co-ordinate = r co-ordinate** of $v_{pj} + 1$.

5.2.3. Assign its θ **co-ordinate** = θ_{jk} which is the corresponding polar angle.

end.

5.3. Advance *next-in-queue* pointer.

end.

After nodes are assigned their *polar topological co-ordinates*, they are partitioned in a similar way as before to different processors, either by the quad-tree based or the binary tree based mapping strategies. The results are illustrated in the next chapter.

Chapter 6

Implementation details and results

The general strategies discussed in the previous chapters were implemented and tested on DECmpp12000/Sx for experimental domains. The languages of implementation are C and MPL. In the following discussion, we first give a brief overview of the implementation issues and then discuss the results.

The complete implementation can be divided into two major components. One is the *sequential component* which runs in the front-end *console system* and is written in the C language. The other is the *parallel component* which runs in the back-end *data parallel unit*(DPU) and is written in MPL. More details about the system components can be found in *chapter 3*. The sequential component is composed of routines for data input/output, domain discretization, coefficient assembly and portions of the specific mapping strategy. The parallel component can be further divided into two sub-components. One component runs in the *array control unit*(ACU) and is composed of sequential mapping related routines. The other component is composed of data parallel routine(s) for the parallel solver and it runs on the PE array. The back-end MPL routines are called from the front-end C routines through the *callRequest* MPL library routine. Different components of the implementation are illustrated in *Fig. 6.1*.

The solver is the routine which takes the maximum percentage of the total program execution time. Hence, an efficient implementation of the solver and some good

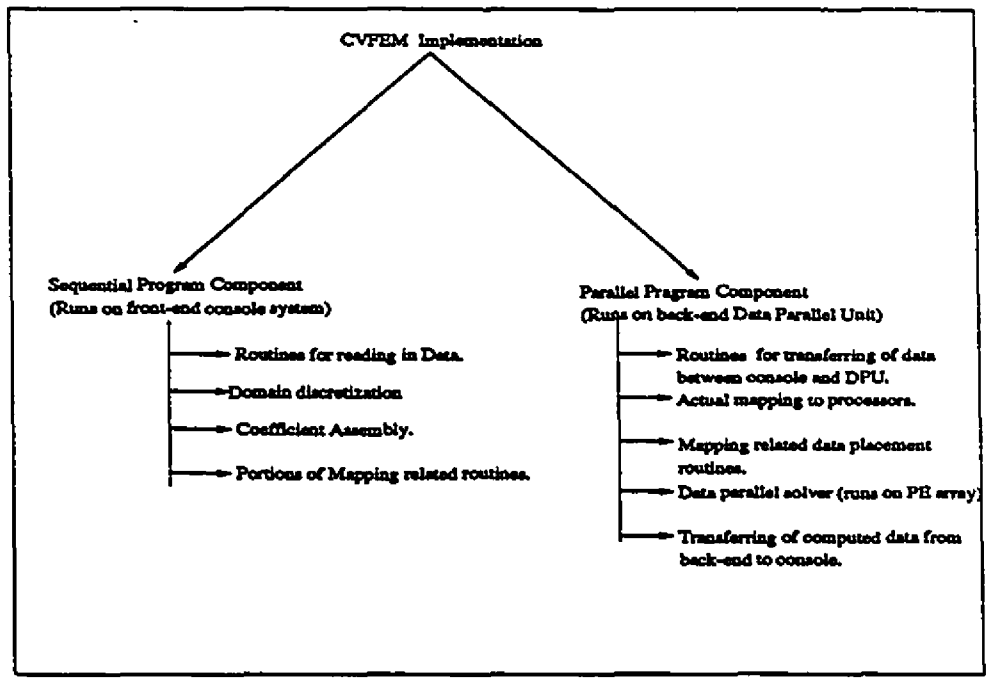


Figure 6.1: CVFEM program components

mapping strategy are the key issues for speedup. The total time spent in the mapping related routines, which we can call the *pre-processing time*, is very small compared to the time in the solver. If the mapper can do some good job in data placement, the pre processing time can be masked by the additional speedup obtained in the solver.

We discuss below the important issues related to the sequential and parallel program components. We start with a discussion on the sequential component.

6.1 The sequential component

The sequential component comprises of routines for input/output of data, domain discretization (routines related to delaunay triangulation of the domain), coefficient assembly and portions of mapping related routines (for example, in the case of quad-tree based mapping strategy based on topological co-ordinates, it may involve routines

for assigning topological co-ordinates to nodes, and routines for building the tree and simultaneous processor partitioning). The data here maybe in two different formats, either as *nodes* which specify a discrete domain without the interconnection pattern, or as *triangles* and *nodes* which completely specify the discrete domain and the problem graph. In the second case, the phase of delaunay triangulation is to be skipped. We first discuss about the data portion, mainly the representation of nodes and triangles.

6.1.1 Representation of the discrete domain

Most of the following work on the sequential component part was done by Clark Verbrugge who is presently working for his doctoral degree at McGill University. Some modifications were made for the parallel machine in order to link with the parallel component(s) and to take care of the memory constraints.

Given a domain of interest, we pick up a set of boundary and internal points, which we call nodes. Each node is specified by its X and Y co-ordinate values and a flag to indicate whether or not it is a boundary node. For simplicity, the X and Y co-ordinate values are all taken to be positive without any harm, i.e. the domain sits in the first quadrant of the cartesian co-ordinate system. The nodes are read in from a file called *nodedata* file, and are arranged in such a way that all boundary nodes come first and then come the internal nodes.

In the internal representation, nodes are arranged in a *linked list* of *records* as they are read in sequentially from the *nodedata* file. Each node is a *record* with fields: its X and Y co-ordinate values; its topological co-ordinate values; a flag to indicate whether or not it is a boundary node, and whether or not it is part of the triangulation yet; another flag to indicate whether or not it is mapped to any processor yet; if so then the identity of that processor; its ϕ value and other coefficient values; pointers to the head and tail of all the triangles with which this node is associated; a pointer to the next record in the list; and other fields for ease of programming.

While performing *delaunay triangulation* of the domain, new nodes are added to the list, and nodes, which are not part of the triangulation, are removed. Finally what we get is a set of nodes and a set of triangles, which completely specify the discrete domain and the triangulation information. This information of nodes and triangles is stored in another file called *xxx.tri*, where *xxx* is the identity of the triangulation file. Whenever we run the same program on the same domain another time, the *delaunay triangulation* part can be skipped by directly reading in data from the triangulation file.

Similar to the representation of nodes, triangles are also arranged in a linked list of *records*. In the case of *delaunay triangulation*, this list is built dynamically with the triangulation process. In the case when the triangulation information is read in from a file, this list is built dynamically as data is read in from the file. In both cases, the final internal representations of the nodes and the triangles are identical.

Each triangle is a *record* with fields: the three corners of the triangle; a flag to indicate some specific status information; and a pointer to the next triangle in the list. Each corner of a triangle is a *record* with fields: pointer to the specific node associated with this corner; coefficients of the other two corner nodes relative to this corner; and pointers to the next and previous triangles associated with this corner which form a doubly linked list of triangles. We discuss it next.

Given any node, we should be able to find out all the triangles associated with this node, i.e.all triangles which have this node as a corner. By knowing about all its associated triangles, we will be able to find out all its neighboring nodes. Similarly, given a triangle we should be able to obtain information about all its corner nodes. The internal representations of nodes and triangles are such that we can obtain this information easily.

The following method can be adopted to find the triangulation and neighborhood information for a node. This is illustrated in *Fig. 6.2*. Given any node v_{pi}, we want to find all its neighboring nodes. Each node has pointers, called *thead* and *ttail* respectively, pointing to the head and tail of the triangles associated with this node.

By following the *thead* pointer of v_{pi}, we reach the first triangle 1 as in the figure. The three corners of this triangle are marked as A, B and C respectively, of which A is the corner associated with this node. By following the pointer to the next triangle from corner A, we reach triangle 2. Corner D of triangle 2 is associated with this node. We follow the pointer from corner D to the next triangle 3, and proceed in a similar way until we finally reach the last triangle numbered as 6, which is also the triangle pointed to by *ttail*. As we know about all triangles associated with this node v_{pi}, we can now easily know about all its neighboring nodes just by looking at the other corners of these triangles.

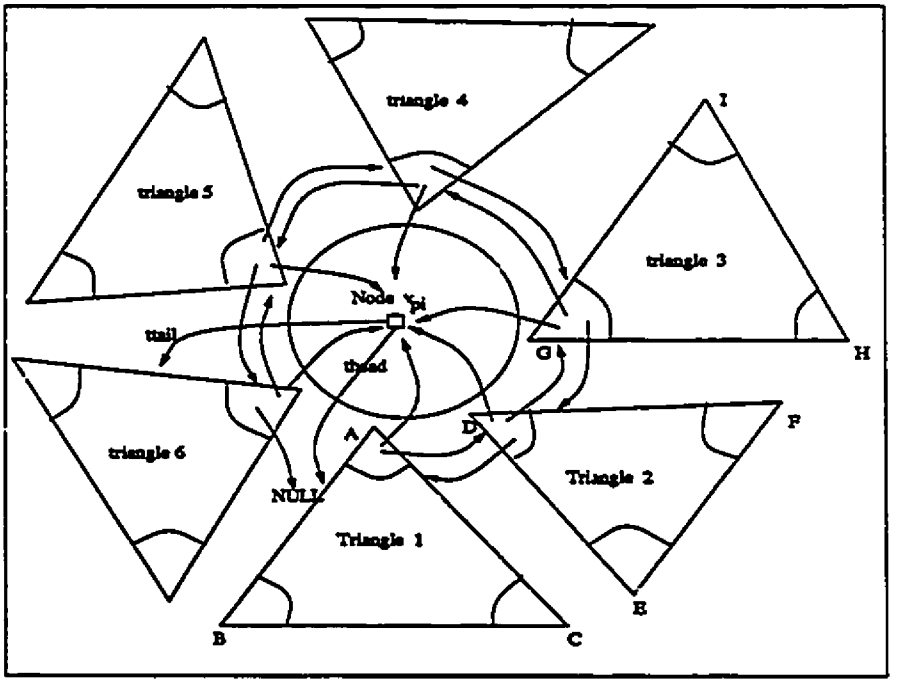


Figure 6.2: Neighborhood information of a node.

After reading in data from the file, the next sequence of operations involve the delaunay triangulation of the domain, in case it is not already performed, and the coefficient assembly. We are not going to discuss here the routines for delaunay triangulation and coefficient assembly, since they are not directly related to our mapping issue. A description on coefficient assembly can be found in *chapter 2*. More about delaunay triangulation can be found in [2, 11]. Instead, we will briefly discuss the

front-end portion of the implementation related to the specific mapping strategy. The following discussion assumes a quad-tree based mapping strategy based on topological co-ordinates. So we will resume our discussion from the point where we already have the nodes and the triangles.

6.1.2 Mapping related routines

For a quad-tree based mapping strategy based on topological co-ordinates, it involves routines for assigning topological co-ordinates to nodes, and routines for building the tree and simultaneous partitioning of the processor mesh. We have a choice between running these routines in the front-end console system or in the back-end ACU. We decided to run them in the console because of memory constraints of the ACU. The drawback is that the user has to inform the console about the present configuration of the back-end PE array, mainly the number of rows and columns. This can be taken care of by allowing the console read in this information from the ACU.

The process of assigning topological co-ordinates to nodes by one of the two schemes already described involves the scanning of the node list and queueing the nodes in a layer-by-layer fashion. For the purpose of queueing nodes, we keep another linked list, each entry of which contains a pointer to a node. Each node contains a flag to indicate whether or not it is already queued. This is necessary because each node may be a neighbor of many other nodes and thus there is a very high possibility that a node will be multiply queued by all its neighboring nodes. Thus it will be assigned multiple co-ordinates leading to a conflicting situation. To avoid this, a node is allowed to be queued only once as the neighbor of only one of all its neighboring nodes.

The next issue is the building of the quad-tree or the binary tree, as described in the previous chapter, on top of the existing data structures for nodes and triangles. One important point to note here is that, in any case we are only interested in the information at the level of the leaves and hence the intermediate nodes of the tree can be freed in the building process, thus saving memory. At the end, the whole tree

can be discarded as soon as each problem node is marked with the processor id to which it is to be mapped. The next step is to scan through the linked list of nodes and then map each node to the processor where it is destined. We discuss it next.

6.2 The parallel component

As discussed previously, the parallel component consists of routine(s) for transferring data between the console and the DPU, mapping related routines for data placement, the data parallel solver, and routine(s) for transferring computed data back to the console. All these routines are invoked from the single procedure called *mpl_sub*, which runs on the ACU. The mpl library routine *callRequest* acts as a bridge between the front-end console and the back-end DPU. The back-end routine *mpl_sub* is invoked from the front-end console process by the *callRequest* library routine. This is illustrated in *Fig. 6.3*. The console process resumes its execution as soon as the the call to the back-end system, i.e. the *mpl_sub* routine, completes.

6.2.1 Data transfer between the console and the DPU

There are four MPL library routines, called *copyIn*, *copyOut*, *blockIn* and *blockOut*, for transferring of data between the front-end console and the back-end DPU. The last two routines are for direct block transfer of data between the console and the PE array, and are not useful in our application. Instead, in our case we have to transfer data in an element by element fashion from the linked list of nodes in console to the ACU, and then place each node in its destined processor. Thus it is a two step process, firstly to read in data from the console to the ACU and secondly to transfer data from the ACU to the PE array. We use the *copyIn* and *copyOut* library routines for the first step, and the *proc* library routine for the second.

It works as follows. Two of the parameters for the call to *callRequest* library routine from console are the starting addresses of the linked lists of nodes and triangles. Let

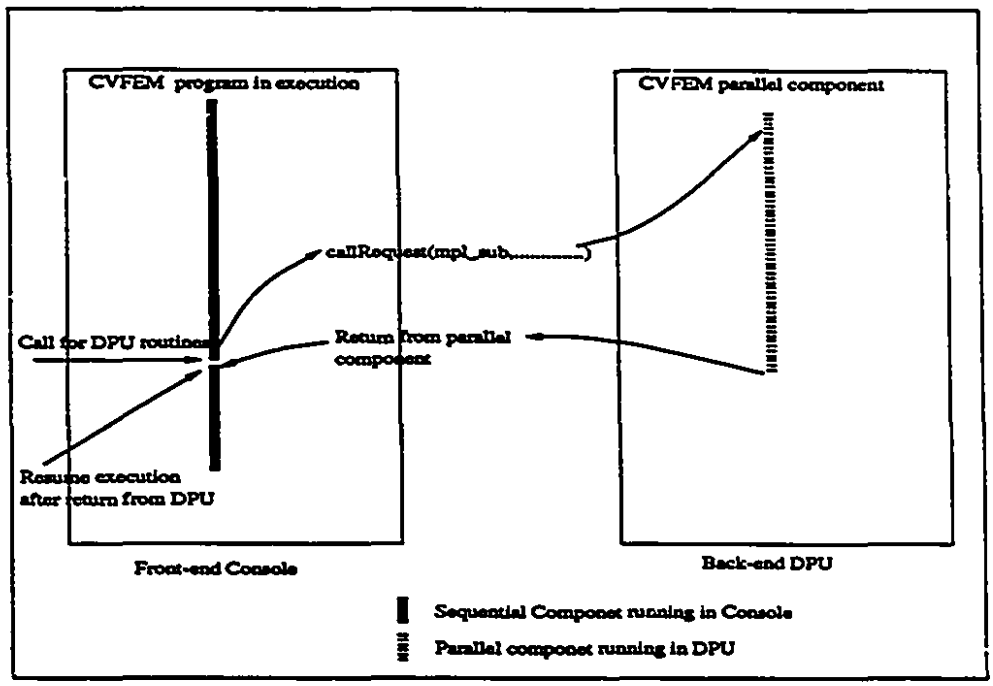


Figure 6.3: The CVFEM program model

them be *fe_nhead_p* and *fe_thead_p* respectively, where *fe* stands for front-end. Let us suppose that the ACU wants to read in the nodes one by one from the linked list in front-end console, and then map them to the PE array. The ACU has its local copy of a node record, of the same type as in the console. Let the address of this record be *be_node_p*, where *be* stands for back-end. In an iterative loop, the ACU calls the *copyIn* library routine. The call to *copyIn* has its parameters: *fe_nhead_p*, *be_node_p* and *nodesize*, which is the size in bytes of a node record. In the first iteration, it copies the first node record from the console to the local copy in ACU pointed by *be_node_p*. The ACU reads appropriate fields from this record including the processor id to which it is to be mapped, and then places data from these fields to the appropriate processor, which we will discuss shortly. It also reads the address of the next record in the front-end linked list, as specified by the *next* field, and modifies the *fe_nhead_p* accordingly. So, in the next iteration, it reads in the next node from the new address of *fe_nhead_p* to the same local copy in the ACU. It continues iterating until *fe_nhead_p* points to NULL. In actual implementation, things may vary slightly, but the basic idea is the

same.

We do not want to create a replica of the front-end linked list in the ACU, obviously for its memory limitations. Instead, we copy one node at a time from the console to the ACU, and then map each node to the destination processor. The temporary copy of the node in the ACU is erased as soon as the next node is read in from the front-end linked list. So the ACU acts as a base camp for each node on its way to its final destination in the PE array.

As we read in a node from the console to the ACU, we already know the identity of the processor to which it is to be mapped, since it is already decided by the front-end quad-tree. What we do not know is the specific partition on that processor to which it will be mapped, since it is decided only dynamically. We consider it in the next section.

6.2.2 Dynamic mapping to the PE array

Theoretically, each processor in the PE array is partitioned into as many parts as the average number of nodes n that should go to each processor, as discussed in the previous chapter. In actual implementation, we have an array of records of length n allocated to each processor. Each element of this array holds information about a particular node, which is mapped to that processor. All n entries of this array hold information about all n nodes that are mapped to that processor. Let us call this array the *node-array*.

Each mapped node is uniquely identified by two parameters: the processor id to which it is mapped, and the index of its entry into *node-array* on that processor. The processor id is determined by the quad-tree or some other mapping strategy. But the index is determined dynamically as the node is actually mapped to that processor. This index information is stored back in the node record in the console so that the mapped node can be correctly identified in subsequent operations such as, communicating with neighbors and retrieving computed data back.

Each entry of *node-array* has fields for storing: ϕ value of the node, its coefficient values, and an array of records called *neighbor-array*. Each entry of this *neighbor-array* contains information about where one particular neighbor is mapped, identified uniquely by the processor id and index on that processor, together with some other field related to coefficient assembly. This *neighbor-array* for each mapped node is filled in a subsequent scan through the node list, after all nodes are mapped and their partition index information on the corresponding processors are known. With these information, each node can correctly communicate with all its neighbors, take their old ϕ values from the previous iteration, and compute its new ϕ value, as is discussed in *chapter 2*. The partitioning of a processor in the PE array is shown in *Fig. 6.4*. Actually this *neighbor-array* contains information about all neighbors which are to be communicated through the router, and it will be clear in a short while.

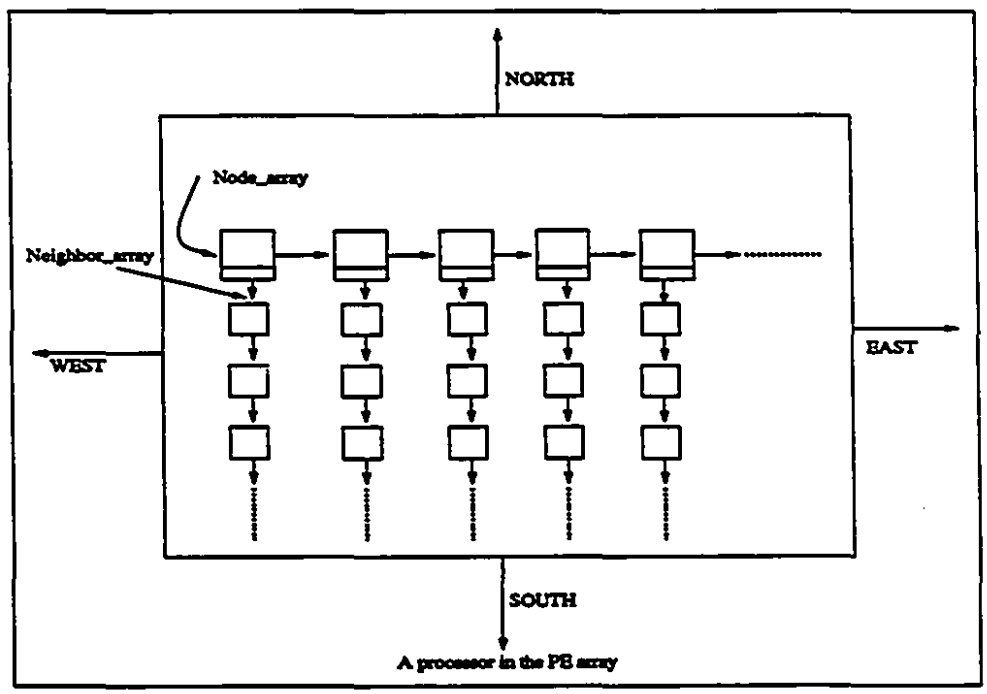


Figure 6.4: Partitioning of a processor in the PE array

Finally, on the average n nodes are mapped to each processor in the PE array. Each mapped node is uniquely identified by its processor id and partition index on that processor, and this information is stored in the front-end console so that

data retrieval and communication proceed correctly. Each mapped node contains information about its ϕ value and other coefficient values, as discussed in *chapter 2*. Each mapped node also contains information about where all its neighbors are mapped. Each neighbor is identified by its processor id and partition index on that processor. The next issue is to set up the communication primitives so that each node can communicate with its neighbors, exchange data, and perform computations in the parallel solver. We discuss it next.

6.2.3 Setting up communication

As discussed in a previous chapter, there are two communication primitives: one is the *Xnet* for fast and synchronous close neighbor communication, and the other is the *router* for asynchronous communication. We divide the entire node-neighbor communication pattern into two parts: one is through the Xnet and the other is through the router or some other MPL library routine using router depending on whichever is more convenient.

The problem with the Xnet communication is that, all active processors are required to communicate in an identical way at any instant of communication. What we mean by identical is that, each communicating processor has to communicate with another processor in the same direction and distance as all other communicating processors do at that instant. In case of an irregular problem, it is very difficult to bring in this harmony. Still, if by some trick we can bring this harmony, we pay some price as some processors might have to remain idle at any instant of communication through the Xnet. This will be clear through an example. Let us consider an $8 * 8$ processor mesh, as in *Fig. 6.5*. Here, at some instant of communication through the Xnet, each active processor is communicating with its immediate north neighbor. Processors 0 and 9 are inactive, i.e not taking part in communication, at that instant. It is because, either none of the nodes mapped to these two processors have its neighboring nodes mapped to its immediate north processor or, at this instant no node mapped to these two processors is required to communicate with its north neighbors.

Whatever is the case, these two processors have to remain idle. How many processors have to remain idle at an instant is a factor in deciding the speedup. The lower is this number, the better is the speedup. The rest of the communication, which could not be done through the Xnet, is to be completed through the router or some other MPL library routine using router. Below, we elaborate it further.

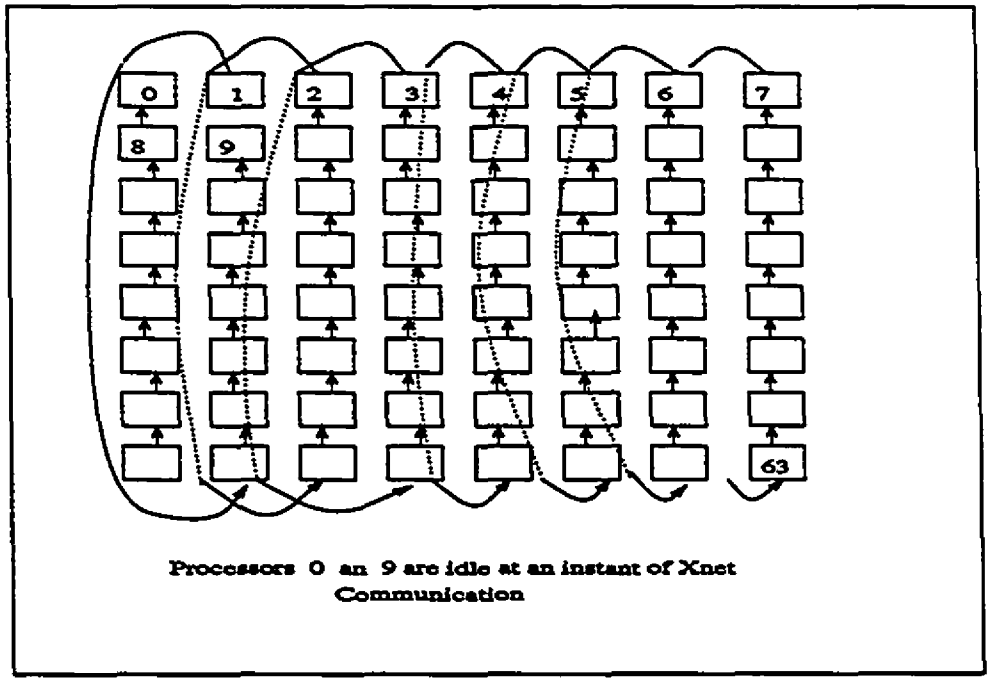


Figure 6.5: An instant of Xnet communication

Each processor has an array called *Xnet-com*, which determines whether a processor should remain active or inactive at some instant of Xnet communication. The length of this array is a multiple of 8. This is because, there are 8 directions of Xnet communication, namely *north*, *south*, *east*, *west*, *north-east*, *north-west*, *south-east* and *south-west*. The first 8 entries determine Xnet communication with processors in these 8 directions at distance one, the next 8 entries determine Xnet communication with processors at distance two, and so on. Let us say that the length of this array on each processor is $8 * XDIST$, where *XDIST* is the distance to which we want to communicate through the Xnet. Obviously, *XDIST* is a parameter which determines speedup. We have to fix some suitable value for *XDIST* for obtaining the optimal

speedup. There is also another parameter, which determines exactly what is the maximum number of times any two processors can be allowed to communicate through the Xnet. We discuss it next.

Each entry of *Xnet-com* is itself an array, called *Indx-array*, whose length *Xindxsize* is another adjustable parameter. Each entry of *Indx-array* is a flag, together with two other fields called *source-index* and *dest-index*. If the flag is set, it indicates that the source processor has to communicate with the destination processor at that instant of iteration, and *source-index* and *dest-index* specify the indices of a node-neighbor pair in the source and destination processor respectively. If the flag is not set, the source processor does not have to communicate with the destination processor at that iteration, i.e. it has to remain inactive(or, we can say idle), and hence we do not have to refer to the other two fields. In *Fig. 6.6*, we illustrate the *Xnet-com* array on an arbitrary processor. The length of this array is 8 indicating that we want to communicate through the Xnet with the immediate 8 neighbors at distance one.

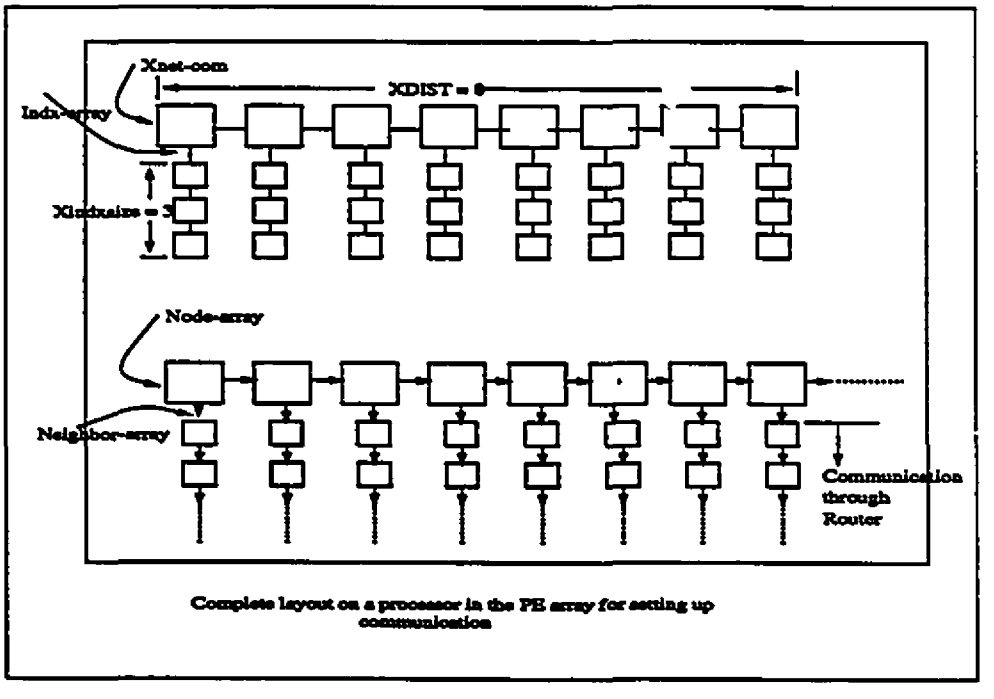


Figure 6.6: Communication layout on a processor

What *Xindxsize* actually determines is the maximum number of times each processor can communicate with another processor through the Xnet. How many times each processor has to communicate with another processor is actually determined by how many node-neighbor pairs are mapped to these two processors. The problem is that, *Xindxsize* has to be identical on all the processors, whereas this number of node-neighbor pairs varies from processor to processor. We fix a value of *Xindxsize* depending on the size of the problem. As a result, some processor might communicate through the Xnet with another processor *Xindxsize* times, while some other processor might do the same less then *Xindxsize* times, and remain idle for the rest of the time. It is also possible that more than *Xindxsize* times Xnet communications may be necessary between some processor pairs, though it is not allowed. Thus, this *Xindxsize* is a parameter which is to be selected carefully depending on the problem size, as it determines the number of processors that have to remain active or inactive at an instant of Xnet communication.

The rest of the communication, which could not be done through the Xnet, is to be completed through the router, or some other library routine using the router. We have already discussed about *neighbor-array* in the previous subsection. Each valid entry of *neighbor-array* actually contains information about a neighboring node which has to be communicated through the router. If a node can communicate with all its neighboring nodes through the Xnet, all entries of *neighbor-array* for this particular node would be empty(or, we can say invalid).

Finally, we have two adjustable parameters, namely *XDIST* and *Xindxsize*, which determine the speedup of the solver. In all applications, an *XDIST* value of 1 is found to be the optimal. *Xindxsize* actually depends on the problem size, and has to be fixed accordingly.

Now we are in a position to discuss the parallel solver.

6.2.4 The parallel solver

The solver works as follows. Each node in the discretization smoothens its ϕ value based on previous ϕ values of itself and all its neighbors. This smoothing operation continues over several iterations until computed values converge for all nodes. This is already discussed in detail in *chapter 2.* We can imagine a big outermost loop, which goes upto 1000 or more iterations. In every iteration of this loop, each node communicates with all its neighboring nodes, take their old ϕ values from the previous iteration, performs some computations with them, and modifies its own ϕ value, as is already discussed.

In the sequential solver, in each iteration of the outermost loop, one has to scan through the entire node list, and subsequently scan through all neighbors of each node. This operation continues over a large number of iterations until all computed values converge. Thus the solver consumes the maximum amount of the whole program execution time. In our data parallel solver, each processor in the PE array works with its own share of nodes in every iteration of the outermost loop, and thus it speeds up computation. We stop when ϕ values on all nodes converge. Alternately, in order to compare the performances of the different mapping schemes, we keep some fixed value for this number of iterations, say around 1000. This big value generally guarantees convergence. Below, we elaborate the solver.

In each iteration of the outermost loop, each processor in the PE array performs three types of operations: (1)it completes all Xnet communication by scanning through the array called *Xnet-com* ; (2) after completing Xnet communication, it scans through its own *node-array* whose each entry contains information about a particular node. For each node in *node-array*, it scans through the valid entries of *neighbor-array* and performs communication with these neighbors through the router. Finally, (3) after communication and data transfer with all neighbors is complete, each node computes its new ϕ value before moving to the next entry in the *node-array.* All these operations within a big outermost loop are illustrated in *Fig. 6.7.*

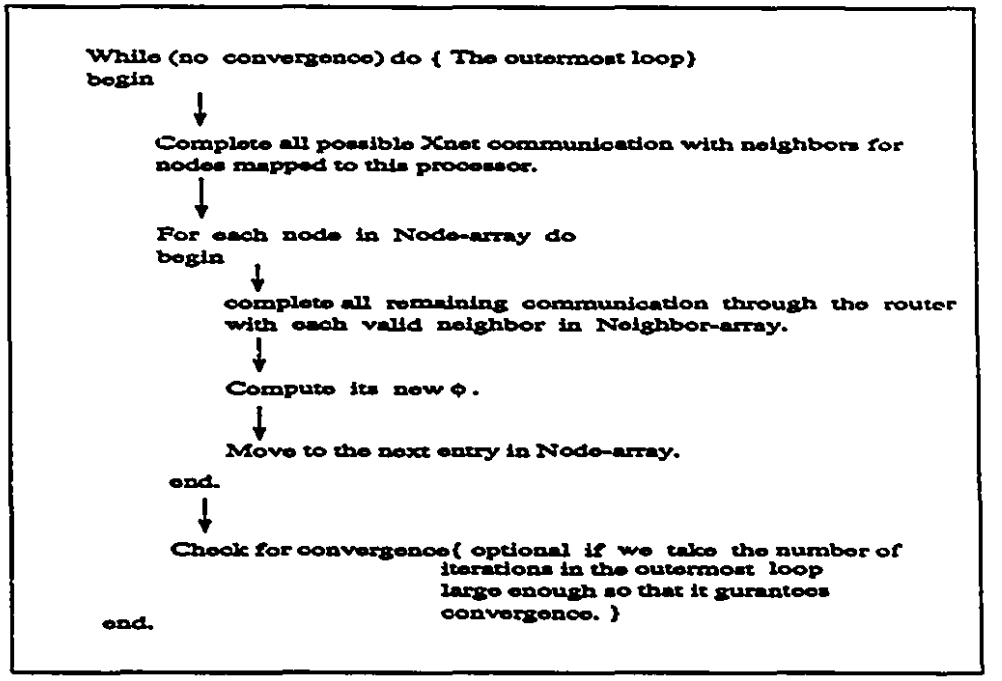


Figure 6.7: A model of the parallel solver on each processor

After all ϕ's converge, the computed values are transferred back to the front-end console. As before, this data transfer operation takes place in two steps where the ACU acts as the intermediate rest camp. Since each mapped node is uniquely identified by its $(processor, index)$ pair, and this information is already stored in the console, this transfer operation can proceed correctly.

6.3 Computing λ

As already discussed in *chapter 4*, λ is some measure which, together with proper load balancing, can judge the quality of a specific mapping strategy. For the measurement of λ, we assume that messages between two processors always travel by the shortest path connecting them. In reality this may not be the case, as message passing is highly machine dependent. Still, this measurement of λ gives a good information about locality and thus, irrespective of the machine, it gives some good idea about the quality of the mapping strategy, which is of particular concern.

In case of the DECmpp12000/Sx, we assume that all messages between any two processors travel by the minimal distance path connecting them, as in the case of Xnet communication. However we have already seen that all communication cannot be completed exclusively through the Xnet. Thus the computed value of λ may not always give the precise information regarding the message distances. Still this information is reliable enough to give some idea about how good the mapping is in preserving the locality.

If $[x_1, y_1]$ and $[x_2, y_2]$ are the locations of two processors v_{si} and v_{sj} in the PE array, then the minimal distance between them, denoted by D_{sij} as already discussed in *chapter 4*, can be given by:

$$D_{sij} = min_x + min_y \tag{6.1}$$

where $min_x = min(|(x_1 - x_2)|, nxproc - |(x_1 - x_2)|)$, $min_y = min(|(y_1 - y_2)|, nyproc - |(y_1 - y_2)|)$, $min(x, y) =$ minimum of x and y, and $nxproc$ and $nyproc$ are the number of rows and columns respectively in the PE array.

In the actual implementation, after knowing about the mapping information of all nodes due to a specific mapping strategy, it is straightforward to compute D_{sij} for any two processor pairs where a node-neighbor pair is mapped. For each node-neighbor pair (v_{pi}, v_{pj}), we find the corresponding processor pair (v_{sk}, v_{sl}) where they are mapped, and then compute D_{skl} as above. The summation of all D_{skl} for all node-neighbor pairs gives the value of λ, as is already discussed in a previous chapter. For simplicity, we treat (v_{pi}, v_{pj}) and (v_{pj}, v_{pi}) as distinct pairs, and hence distances are counted twice. Obviously this does not affect the final judgment.

6.4 Results

All the strategies discussed previously were implemented and tested on the DECmpp12000/Sx. Some of the results with these strategies on different irregular problem domains are illustrated below:

(a)*Problem graph 1*: Wrench (253 nodes):

Different Strategies	λ	Time in solver (in Sec.)
Random mapping	22852	6.09
General Close-neighbor heuristic	15539	5.34
Quad-tree (Geometric)	10636	2.68
Binary-decomposition (Geometric)	11792	2.62
Binary-decomposition (Polar topological)	13528	2.56

(b)*Problem graph 1*: Irregular plate as in *Fig. 6.8*(553 nodes):

Different Strategies	λ	Time in solver (in Sec.)
Random mapping	42156	7.21
General Close-neighbor heuristic	25636	6.56
Quad-tree (Geometric)	15040	4.66
Binary-decomposition (Geometric)	15380	4.56
Binary-decomposition (Polar topological)	19552	4.82

(c)*problem graph 2*: Arc of a circle (4194 nodes and 8062 triangles):

Different Strategies	λ	Time in solver (in Sec.)
Random mapping	351908	33.8
General Close-neighbor heuristic	124296	31.1
Quad-tree (Geometrical)	65564	26.8
Binary-decomposition (Geometric)	63176	26.7
Binary-decomposition (Polar topological)	55892	26.1

(d)*problem graph 3*: Wrench (49238 nodes and 96384 triangles):

Different Strategies	λ	Time in solver (in Sec.)
Random mapping	5828126	
General Close-neighbor heuristic	1765148	322.9
Binary-decomposition (Geometric)	161084	286.8
Binary-decomposition (Polar topological)	237452	290.4

The timings above are the average timings with at least three observations. The timings vary slightly depending on system load.

The discussion about timings is not complete unless we give some idea about the pre-processing times. In ths case of the quad-tree based and the binary decomposition based mapping strategies, these pre-processing times are the times required to construct the corresponding trees. In the case of the topological co-ordinate based schemes, it also includes the extra time in assigning topological co-ordinates to the nodes. Below we give some illustrations of pre-processing times for some of the experimental domains:

(a) Irregular plate as in *Fig. 6.8*(553 nodes):

Different Strategies	Pre-processing time (in sec.)
Quad-tree	0.08
Binary-decomposition(geometric)	0.06
Binary-decomposition(polar topological)	0.08

(b) Arc of a circle (4194 nodes and 8062 triangles):

Different Strategies	Pre-processing time (in sec.)
Quad-tree	6.99
Binary-decomposition(geometric)	0.59
Binary-decomposition(polar topological)	0.83

(c) Wrench (7917 nodes and 14265 triangles):

Different Strategies	Pre-processing time (in sec.)
Quad-tree	26.96
Binary-decomposition(geometric)	1.22
Binary-decomposition(polar topological)	1.63

(d) Wrench (49238 nodes and 96384 triangles):

Different Strategies	Pre-processing time (in sec.)
Quad-tree	1456.8
Binary-decomposition(geometric)	11.48
Binary-decomposition(polar topological)	43.33

As it is evident from above, with small domains, all three strategies take comparable pre-processing times. But, with larger domains, the quad-tree based strategy becomes very expensive in terms of the pre-processing time. Since all three strategies take almost an equal amount of time in the solver, the quad-tree based strategy is obviously inferior to the other two for larger domains.

The parallel solvers with different mapping strategies are all identical. We measure the time in the solver after a fixed number of iterations, say 1000, which is large enough for convergence. It is evident that a reduction in λ does not always guarantee a reduction in time. The reason for this has already been discussed previously. When we compute λ, we assume that all communication is done through the Xnet, and it follows the shortest message paths. But in actual implementation, some communication has to be completed through the router. Router communication can give variable timing depending on the number of collisions in the communication path. Once again, we stress that this issue is highly machine dependent.

Some of the experimental irregular domains are illustrated in *Figs. 6.8, 6.9, and 6.10.*

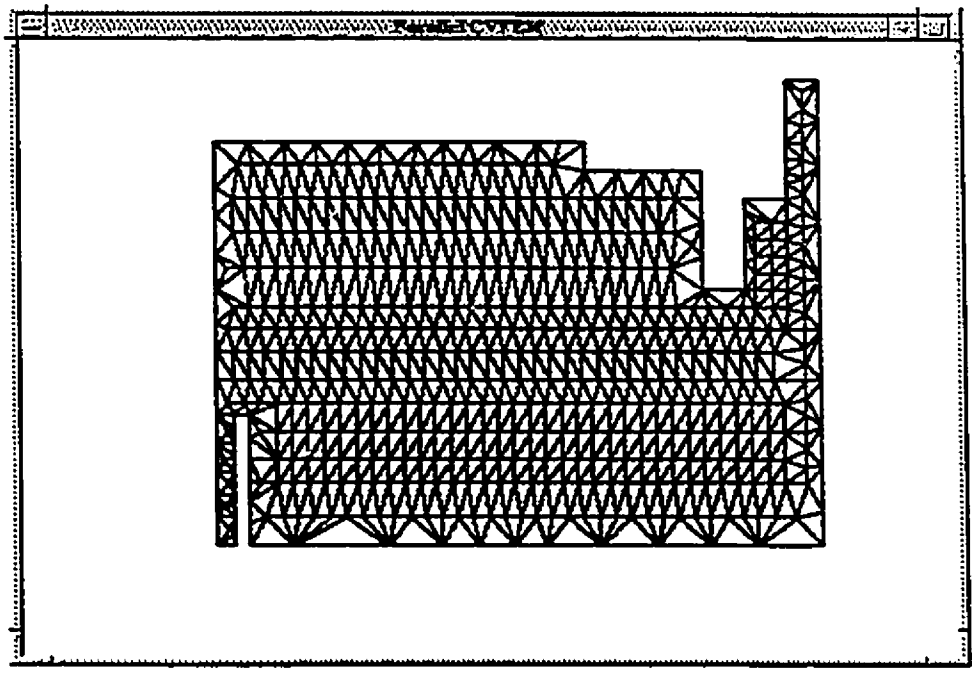


Figure 6.8: An irregular domain

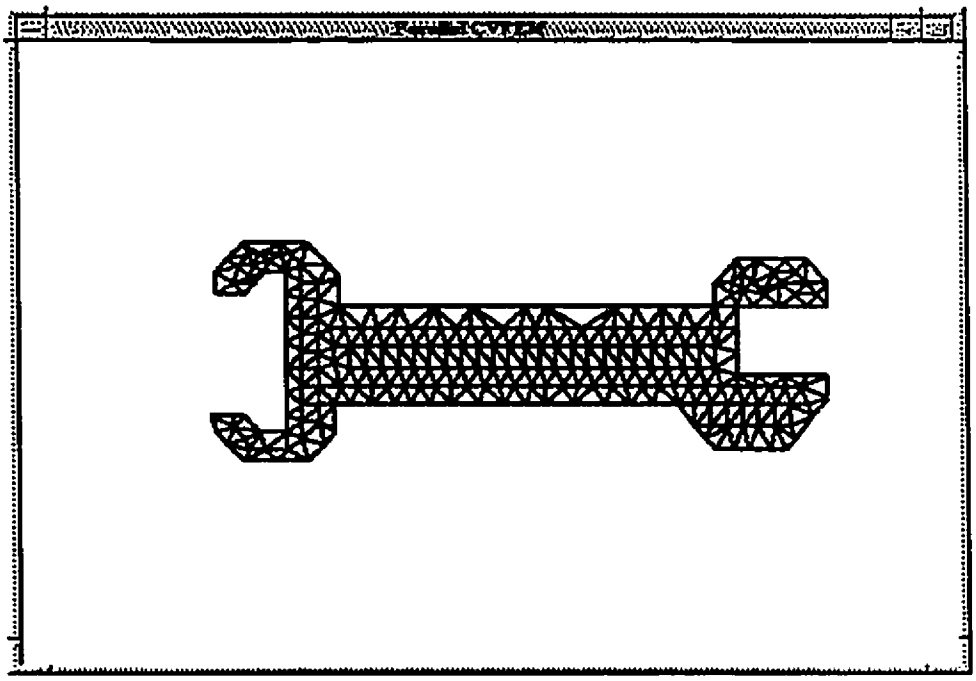


Figure 6.9: A wrench shaped domain

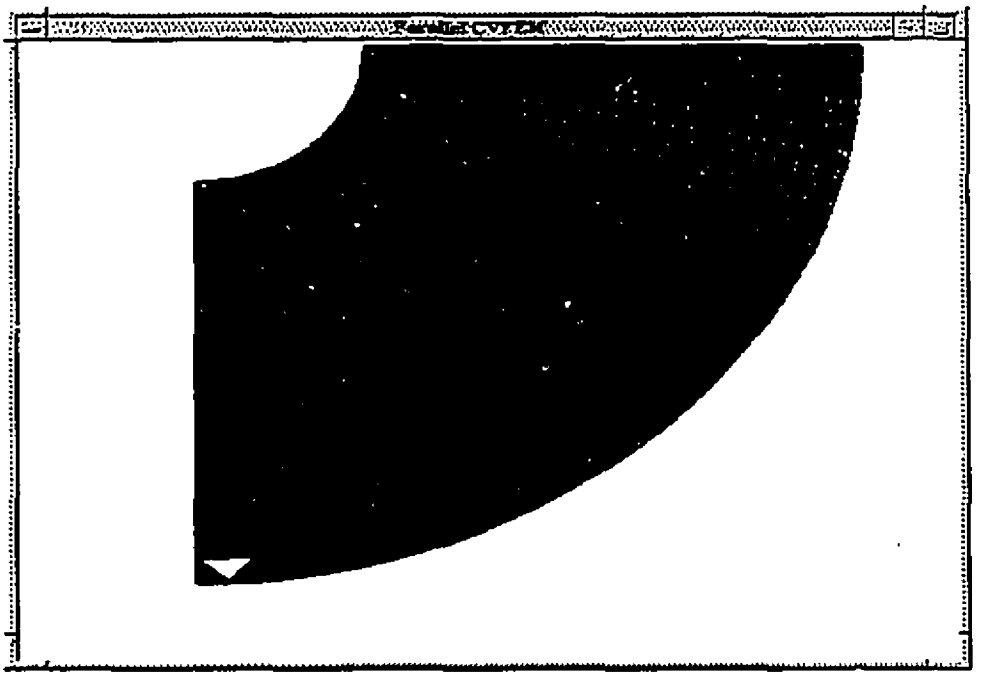


Figure 6.10: Arc of a circle

6.5 Conclusion and future work

This work is part of EPPP(environment for portable parallel programs) project jointly sponsored by Centre de recherche informatique de Montreal(CRIM), IBM Canada, Digital Equipment Canada, Alex Parallel Computers, and Industry Canada. The project focuses on the development of a portable parallel programming environment which can be used over a wide variety of architectures.

The present trend is on the implementation of some of these algorithms in HPC, which is a data parallel language under development as part of the EPPP project.

In HPC, the user can define a virtual machine suitable to his needs. This virtual machine can be a linear array of virtual processors, or a two dimensional mesh similar to DECmpp12000/Sx, or maybe something else. He can apply the same algorithms described previously to map the irregular problem graph to these virtual processors. A second phase of mapping is to be performed by the compiler to map these virtual

processors to the underlying physical processors. The underlying machine need not necessarily be an SIMD machine. For instance, the present underlying physical configuration is a cluster of RS6000 connected by a high speed IBM switch. Data references across processors are transformed into corresponding *send* and *receive* messages by the compiler. This mechanism is transparent to the user.

Mapping of the virtual processors to the physical system plays a crucial rule in communication, and is itself an active area of research. Present emphasis is on the study of the communication overhead under these situations and how to improve them.

Bibliography

[1] B.R. Baliga and S.V. Patankar,*Elliptic Systems: Finite-Element Method II*, Handbook of numerical heat transfer, Editors: W.J.Minkowycz, E.M.Sparrow, G.E.Schneider, R.H.Pletcher, Published by John Wiley and Sons, Inc., 1988.

[2] T.J. Baker,*Automatic Mesh Generation for Complex Three-Dimensional Regions using a Constrained Delaunay Triangulation*, Engineering with computers, Vol. 5, 161-175, 1989.

[3] S.W. Hammond and R. Schreiber,*Mapping unstructured grid problems to the connection machine*, RIACS Technical Report, 90.22, 1990.

[4] S. Lee and J.K. Aggarwal, *A mapping strategy for parallel processing*, IEEE transactions on Computers, vol.C-36, No.4, pp. 433-442, April 1987.

[5] *DECmpp Sx programming language user's guide*, Digital Equipment Corporation, Maynard, Massachusetts.

[6] *DECmpp Sx architecture specification*, Digital Equipment Corporation, Maynard, Massachusetts.

[7] B.W. Jones, M.G.Everett and M.Cross, *Mapping Unstructured Mesh CFD Codes onto Local Memory Parallel Architectures*, Parallel Computational Fluid Dynamics '92, Editors:R.B.Pelz, A.Ecer and J.Hauser.

[8] P. Sadayappan and F. Ercal, *Nearest-neighbor Mapping of Finite-element Graphs onto Processor Meshes*, IEEE transactions on Computers, vol.C-36(1987), pp. 1408-1424.

[9] C. Pommerell, M. Annaratone and W. Fichtner, *A Set of New Mapping and Coloring Heuristics for Distributed-memory Parallel Processors*, SIAM J. Sci. Stat. Comput., Vol. 13, No. 1, pp. 194-226, January 1992.

[10] M.J. Berger and S.H. Bokhari, *A Partitioning Strategy for Nonuniform Problems on Multiprocessors*, IEEE transactions on Computers, Vol. C-36, No.5, pp. 570-580, May 1987.

[11] J. Boissonnat and M. Teillaud, *On the Randomized Construction of the Delaunay Tree*, Theoretical Computer Science 112 (1993) 339-354.

[12] Z. Cvetanovic, *The Effects of Problem Partitioning, Allocation, and Granularity on the Performance of Multiple-Processor Systems*, IEEE transactions on Computers, Vol. C-36, No.4, April 1987.

[13] C. Aykanat, F. Ozguner, F. Ercal and P. Sadayappan, *Iterative Algorithms for Solution of Large Sparse Systems of Linear Equations on Hypercubes*, IEEE transactions on Computers, Vol. 37, No. 12, pp. 1554-1568, 1988.

[14] T.M. Shih, *Numerical Heat Transfer*, Hemisphere Pub. Corp., Washington, (c1984).

[15] J.J. Modi, *Parallel Algorithms and Matrix Computation*, Clarendon Press, Oxford, 1988.

[16] P.R. Woodward, *Interactive Scientific Visualization of Fluid Flow*, IEEE Computer, October 1993.

[17] T. Tezduyar, S. Aliabadi, M. Behr, A. Johnson, S. Mittal, *Parallel Finite-Element Computation of 3D Flows*, IEEE Computer, October 1993.

[18] A.K. Noor and S.L. Venneri, *A Perspective on Computational Structures Technology*, IEEE Computer, October 1993.

[19] P. Sadayappan, F. Ercal, and J. Ramanujam, *Cluster Partitioning Approaches to Mapping Parallel Programs onto a Hypercube*, Parallel Computing, 13, 1-16(1990).

[20] C. Farhat, N. Sobh and K.C. Park, *Transient Finite Element Computations on* 65536 *Processors: The Connection Machine*, International Journal for Numerical Methods in Engineering, Vol.30, 27-55(1990).

[21] Z. Johan, T.J.R. Hughes, K.K. Mathur and S.L. Johnsson, *A Data Parallel Finite Element Method for Computational Fluid Dynamics on the Connection Machine System*, Computer Methods in Applied Mathematics and Engineering, 99, 113-134(1992).