Parallel Adaptive Mesh Generation

A. I. Khan† and B.H.V. Topping‡
†Postgraduate Research Student ‡Professor of Structural Engineering
Department of Civil Engineering,
Heriot-Watt University
Riccarton, Edinburgh, EH14 4AS, United Kingdom

received 5th May 1991
revised 25th May 1991

Abstract

Un-structured meshes have proved to be a powerful tool for adaptive remeshing of finite element idealizations. This paper presents a transputer-based parallel algorithm for two dimensional un-structured mesh generation. A conventional mesh generation algorithm for un-structured meshes is reviewed by the authors, and some program modules of sequential C source code are given. The concept of adaptivity in the finite element method is discussed to establish the connection between un-structured mesh generation and adaptive remeshing.

After these primary concepts of un-structured mesh generation and adaptivity have been presented, the scope of the paper is widened to include parallel processing for un-structured mesh generation. The hardware and software used is described and the parallel algorithms are discussed. The Parallel C environment for processor farming is described with reference to the mesh generation problem. The existence of inherent parallelism within the sequential algorithm is identified and a parallel scheme for un-structured mesh generation is formulated. The key parts of the source code for the parallel mesh generation algorithm are given and discussed. Numerical examples giving run times and the consequent "*speed-ups*" for the parallel code when executed on various numbers of transputers are given. Comparisons between sequential and parallel codes are also given. The "*speed-ups*" achieved when compared with the sequential code are significant. The "*speed-up*" achieved when networking further transputers is not always sustained. It is demonstrated that the consequent "*speed-up*" depends on parameters relating to the size of the problem.

1 Introduction

Although the finite element formulation is a powerful tool in structural analysis its accurate use is dependent on human expertise. The solutions obtained are always approximate and if a poor domain discretization is involved then the results may be dangerously different from the true solution.

Research on adaptivity and error estimation [1] aims at removing the human factor and automating the finite element procedure. For any adaptive finite element technique to to be implemented, it is essential that a mesh generator is available to generate meshes in accordance with the information received from the adaptive calculations of the *a posteriori error* estimator [2] using the finite element solution. A limit on the overall error for the finite element problem is fixed and the procedure is repeated until a solution mesh is obtained with the error over the whole domain being within acceptable limits.

An algorithm based upon the advancing front technique of Peraire *et al* [2] has been used in [1, 2, 3] for adaptive remeshing. Taking this mesh generation algorithm as a potent finite element preprocessor the object of the study has been to develop it for use in a parallel computing environment.

The un-structured mesh generator of Peraire *et al* [4] relies upon efficient searching algorithms of Bonet and Peraire [5] for locating the points inside certain regions and determining intersections between geometrical objects [4]. In the two dimensional case the regions are the triangular elements of the previous mesh and the geometric objects are the sides of the triangular elements.

Work has already been undertaken on the parallelisation of the finite element method on transputer networks [6, 7, 8]. Parallel transputer based preprocessors such as the adaptive mesh generator will pave the way for a fully automated finite element method of analysis. With work stations comprising a personal computer at the nucleus and a modest network of transputers, main frame supercomputer performance may be achieved with total hardware costs less than than a small number of good quality personal computers.

2 Case study for the development of a parallel mesh generator

Un-structured mesh generation by the advancing front technique [2] may be divided into two main portions.

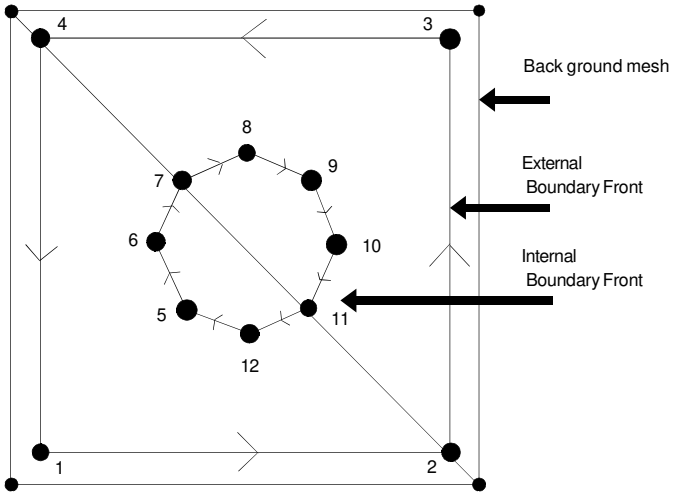

Figure 1: Idealisation of domains with internal boundaries as a boundary front

1. Generation of boundary nodes.
2. Generation of triangles.

2.1 Generation of boundary nodes

The domain to be meshed is discretized using triangular elements and the information on the coarse initial mesh for the node coordinates and the topology of the elements is specified. This initial coarse mesh forms the background mesh. The domain to be remeshed must be contained within the boundaries of the background mesh. If required the boundaries may coincide. The boundary of the domain to be triangulated is idealised as a loop, linking all the specified boundary nodes in counter-clockwise direction. If the domain contains any internal boundaries (openings or cut-outs) then the nodes on these internal boundaries are idealised as a loop linking them in a clockwise direction. This domain boundary discretization is done in accordance with the convention adopted by Lo [9]. The idealization of a square shaped domain containing an arbitrary internal boundary is shown in Fig. 1 this idealization is known as the boundary front. The mesh shown in Fig. 1 comprises two elements. This mesh would be referred to as the background mesh for the first re-meshing of this domain. In a subsequent remeshing the last generated mesh becomes the next background mesh.

The object of assigning a counter-clockwise numbering to external boundary nodes and clockwise numbering to internal boundary nodes is to guarantee that triangular elements generated by the algorithm, using the segments of the front as their base, will always form within the domain to be triangulated. When a boundary segment undergoes element generation, then only the nodes on the left side of the element are considered as probable candidates for forming the apex of the triangle if the afore mentioned numbering conventions are used. The left side of a segment is determined from the nodal coordinates of the first and second nodes of the segment. As the node numbering on the segments is done in a particular order, say counter-clockwise for external boundary segments, therefore a vector having its direction set at 90 degrees counter-clockwise from the direction of the segment will define the space to the left of the external boundary segment. The nodes on the loop formed by the external boundary segments are numbered counter-clockwise so that the elements will form in the region enclosed by the external boundary segments. For the internal boundary segments it is required that the elements should form outside the loop defined by the internal boundary segments and so the node numbering convention is reversed.

For a mesh generator designed for structural analysis where the optimum shape of the mesh elements is that of an equilateral triangle, the mesh parameter for the triangular mesh generator may be taken as the magnitude of the side length of the triangular mesh element to be generated in a particular region of the domain. Thus for each element i of the background mesh a mesh parameter value $\delta(i)$ is assigned. The generation of nodes on each segment of the initial front is done in accordance with the theory presented by Peraire *et al* [2]. The algorithmic details are as follows:

1. For each node the δ values for all associated elements are averaged. These nodal mesh parameter values are termed δ_n. Thus for any node a having a total number of mesh elements connecting with it n_a the associated nodal mesh parameter

may be calculated as

$$\delta_{na} = \frac{\sum_{i=1}^{n_a} \delta(i)}{n_a} \tag{1}$$

2. The intermediate nodes are generated from the end of the segment which has a higher value of δ_n towards the end with the lower value.
   The intermediate node generation may be represented by the following set of simple equations.

$$L_1 = \frac{L_2 * \delta_{nx}}{L_2 + \delta_{nx} - \delta_{n2}} \tag{2}$$

$$(\delta_{nx})_{new} = \frac{[(\delta_{nx} * L_2) + (\delta_{n2} * L_1)]}{L} \tag{3}$$

   Where: the greater nodal parameter value is termed δ_{n1} and the lesser δ_{n2}; L_1 is the distance of each generated node from the first end; and L is the length of the segment. Also δ_{nx} is the interpolated value of the mesh parameter at each generated node location, equal to δ_{n1} at the first end and then decreasing to δ_{n2} at second end of the segment. The distance between the generated node and the second end is defined by L_2. Initially $L_2 = L$ and $\delta_{nx} = \delta_{n1}$.
   This procedure is repeated provided $L_2 > \delta_{n2}$, and the nodes are generated on the segment.
   The node generation stops as soon as the available space to place the next node becomes less than δ_{n2}. The last node generated is usually not in proportion to the local nodal mesh parameter values in that region. In such cases it is therefore necessary to distribute the last L_2 value over the previously generated intermediate segments, in proportion to their lengths. The last generated node is therefore deleted from the node list.
3. By repeating this procedure over all the segments of the initial boundary segments a front may be obtained, comprising straight line segments with the length of each segment in proportion to the local mesh parameter value.

2.2 boundnode

The function **boundnode** generates the nodes on the boundaries of a domain. The structure of the function is as follows:

```
   nsgt = 0;
   for(i=0; i < bsgt; i++)
   {
/* identify nodes */
   nod1= seg[i][0];
   nod2= seg[i][1];
/* assign x, y cords to identified nodes */
   xo1=cord_bg[nod1-1][0];
   yo1=cord_bg[nod1-1][1];
   xo2=cord_bg[nod2-1][0];
   yo2=cord_bg[nod2-1][1];

/* generate boundary nodes proportionally */

   fprop(i,xo1,yo1,xo2,yo2,nod1,nod2,bc[i],cord_bg,nod_bg);
   }
   bdn = bdn1;

/* call triangle generation function */

trian(cord_bg, nod_bg, ei);
return;
```

The total number of segments in the boundary fronts (both external and internal) is termed **bsgt** and the number of segments of the assembled front, after the node generation on the boundary segments, is termed **nsgt**. The number of segments on the assembled front i.e. **nsgt** will always be greater than or equal to the number of segments on the starting boundary front; i.e. **bsgt**. The function **fprop** generates nodes on each boundary segment in accordance with the theory presented in subsection 2.1 and forms the advancing front as the node generation takes place on each boundary segment. If a uniform value of $\delta_i = 50$ is specified for each element of the background mesh in Fig. 2, the overall dimensions of the square being 100 by 100, then **bsgt** remains equal to **4**. For $i = 0$, that is, for the first iteration, **fprop** would generate a node numbered 5 at the center of the first boundary segment having node number 1 and 2 as the first and the second nodes respectively. The value of **nsgt** at this stage would be 5. This process would be repeated until every boundary segment has been divided into two equal parts by placing a node at the middle of each segment. The final number of segments in the assembled front would be **nsgt = 8**. The final topology of the assembled front for the above case is shown in table 1 and the generated nodes are shown in Fig. 3.

After the node generation has been completed on all the boundary segments **bsgt** and the front assembled, the function **trian** is called which starts the generation of the triangular elements on the advancing front assembled by **fprop**.

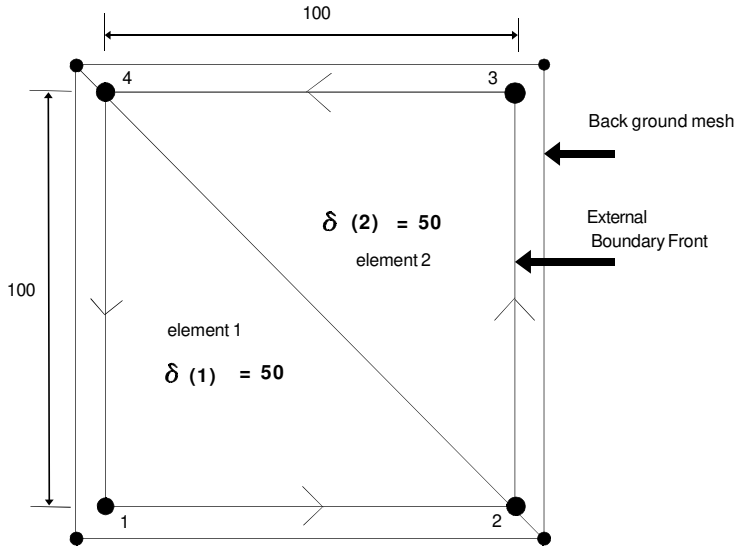

Figure 2: Background mesh and boundary front of a square-shaped domain

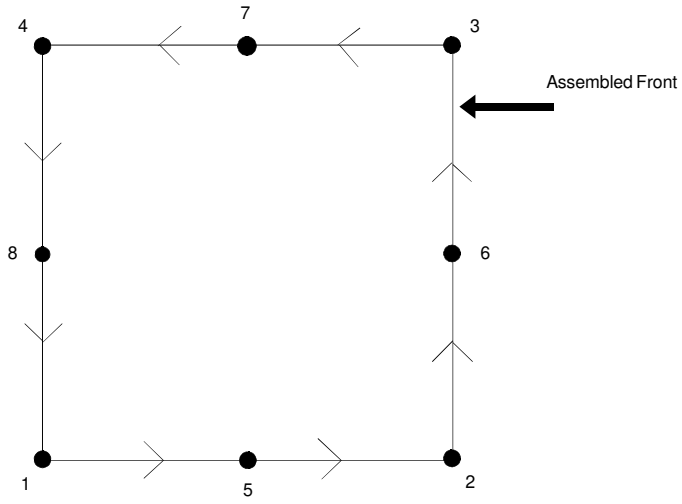

Figure 3: Node generation with a uniform background mesh parameter of $\delta_i = 50$

nsgt	first node	second node
1	1	5
2	5	2
3	2	6
4	6	3
5	3	7
6	7	4
7	4	8
8	8	1

Table 1: The nodal connectivities of the assembled front

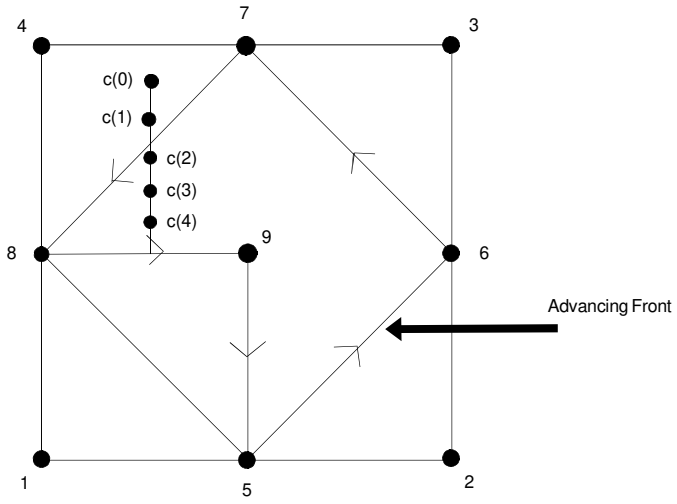

Figure 4: Existing nodes (5,6,7,8,9) and the new nodes (c(0) to c(4))

2.3 Generation of the triangular elements

The triangle generation scheme follows the scheme described by Peraire *et al* [2, 4]. Features similar to those presented by Jin and Wieberg [10] for ensuring the robustness of the algorithm have been included in the authors' code.

2.3.1 Identification of candidate nodes for selection as an apex node

Triangular elements are generated by either identifying suitable existing nodes from the front and/or generating new nodes. The existing and the new nodes are shown in Fig. 4. The existing suitable nodes are stored in a vector **node** and the new nodes, generated along a line perpendicular to the segment, are stored in vector **c**. The node numbers in both these vectors are candidates for the apex node of the triangular element.

As shown in Figures 4 and 5, five equally spaced nodes, c(0) to c(4), are generated along a perpendicular chord from the mid point of the reference segment. The length of this perpendicular chord is $\delta * \sin 60$ where δ is equal to the interpolated value of the mesh parameter at the mid-point of the reference segment.

The nodes in vector **node** include all those existing nodes which lie within a circle of radius δ. These nodes are referred to as the *near nodes*.

2.3.2 Selection of the apex of the triangular element

In references [2, 4] the criteria are laid down for selection of a node to form the apex of the generated triangular element. These criteria are only applicable to nodes which are on the advancing front. The object of these geometrical criteria is to ensure that consistent (equilateral or near equilateral) triangular elements are generated, in any case where an existing node from the advancing front is selected.

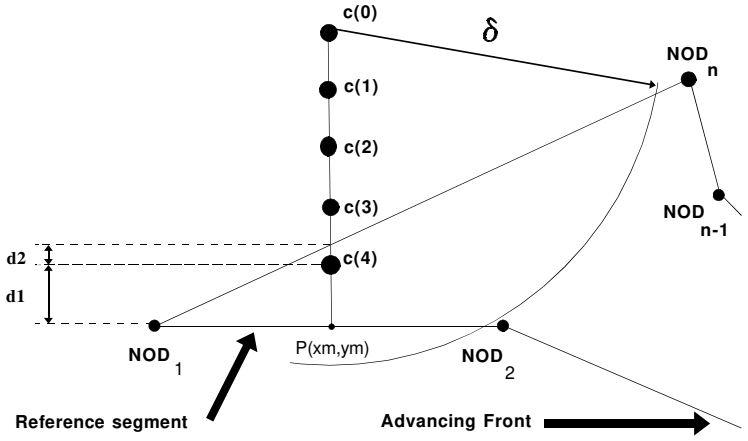

Figure 5: A situation requiring relaxation of node selection criterion; $d_1 > d_2$

It is preferable to use an existing node from the vector **node** rather than to select a new node from the vector **c** since the generation of additional nodes is more likely to result in the formation of ill-formed triangular elements: i.e. elements having a minimum angle of less than 30 degrees. So provided a suitable node can be selected from **node** the generated nodes in vector **c** are not used.

The candidate nodes in vector **c** are listed in the order c(0) to c(4). The listing of nodes in vector **node** commences with the node closest to c(0) and ends with the node farthest from c(0). Ideally an existing node close to c(0) should be selected since this will result in a well-balanced triangle without the generation of an additional node.

The checking of each candidate node for selection as an apex node commences with the nodes from array **node** and then proceeds to array **c**. All nodes considered have to satisfy the condition that the sides of the element thus formed do not intersect any segment of the advancing front. The criteria for the selection of the apex node are as follows:

1. If NOD_n of Fig. 5 is assumed as a *near node* then it shall form the apex of a triangle if line segment

$$\overline{NOD_1NOD_n} < 1.5 * \delta > \overline{NOD_2NOD_n} \tag{4}$$

2. Otherwise a node from the vector **c** will be selected, starting first with c(0) and terminating with c(4).

2.3.3 Robustness

The algorithm presented above is not generally robust because it does not prevent selection of a node from the vector **c** which may be too close to another segment of the front. In due course, when the turn comes of that other segment for triangle generation, then owing to lack of space the algorithm fails to select an apex node and the mesh generation fails at this stage.

A pre-emptive search over the segments of the assembled front is carried out to determine if it is possible to select a new node from the array **c** to form the apex of triangle. The distance of the nearest segment from the reference segment, (i.e. the segment undergoing triangle generation), is monitored to ensure that a prospective candidate node for the triangle apex selected from array **c** is not too close to the nearest segment. If the distance of this candidate node from the nearest segment is less than its distance from the reference segment and yet no existing node has been included in **node**, then the selection criteria pertaining to the existing nodes for the final selection of the apex node are relaxed to ensure that a selection is made from one of these nodes instead of from **c**. This ensures that the apex node will continue to be selected and the mesh generation process will proceed at the expense of the loss in consistency of the mesh during such situations.

From Fig. 5 it can be seen that the no node from the front qualifies as a *near node*; and the last node on the perpendicular chord is located too close to a neighbouring segment. To ensure that a node is always selected situations such as these require relaxation of the selection criterion for *near nodes* and/or apex nodes, as given in sub-sections 2.3.1 and 2.3.2 respectively. When the criterion for the selection of *near nodes* is relaxed then the *near nodes* are taken as up to the first twenty existing nodes closest to c(0). Situations requiring relaxation of the criterion usually result in the formation of elements with undesirable shapes. Nonetheless, from the post-generation processing, the quality of the mesh may generally be restored.

2.4 trian

The structure of the function `trian` for triangular element generation is given below.

```
/* call renum to place the smallest segment in bseg[][]
       as the first segment in seg[][] */
renum();

while(nsgt > 0)
{nod1=seg[0][0];
 nod2=seg[0][1];
 iseg = 0;

 x1=cord[nod1-1][0];
 y1=cord[nod1-1][1];
 x2=cord[nod2-1][0];
 y2=cord[nod2-1][1];

 seglen(x1,y1,x2,y2,&L);
 xm = (x1+x2)/2.;
 ym = (y1+y2)/2.;
 d1 = dels;

 search_elbg1(xm,ym,&d1, cord_bg, nod_bg, ei);

 seldel(d1, L, &seldel1);

 perpcord(x1,y1,x2,y2,seldel1,ccc);

 for(i=0; i < 20; i++)
 node[i] = 0;
 nearnode(nod1,nod2,ccc[0][0], ccc[0][1],node,seldel1, &test_sel);

 test_dis = pt_dist(x1,y1,x2,y2,ccc,ccc[4][0],ccc[4][1]);

 /* test_sel = 1 indicates that no near node has been selected
 test_dis  = 1 indicates that the node from ccc is too close
 to a segment of the front. So a near node is required relaxing
 the condition of node being within a specified radius */
 if(test_sel == 1 && test_dis == 1)
  {
  for(i=0; i < 20; i++)
  node[i] = 0;
  nearnode_ult(nod1,nod2,ccc[0][0], ccc[0][1],node);
  test_sel = -1;
  }

  cri = 1.5;
  selnod=selnode(PER,cri,iseg,seldel1,nod1,nod2,ccc,node,test_sel);

  if(selnod == -1)
  {
  for(i=0; i < 20; i++)
  node[i] = 0;
  nearnode_ult(nod1,nod2,ccc[0][0], ccc[0][1],node);
  selnod=selnode_drs(iseg,nod1,nod2,ccc,node);
  }
  if(selnod == -1)
  {
  break;
  }
  en++;
  prune(selnod,iseg,en_i, en_f, seldel1);
```

```
  if(nsgt == 0)
  break;

  renum();
}

   nn = bdn1;
   ne = en;

return;
```

The function `renum` renumbers the advancing front by making the shortest segment of the front the first segment. The length of this segment is calculated by `seglen` and the coordinates of its mid point are calculated as `xm ym`. The function `search_elbg1` calculates the mesh parameter value δ by locating `(xm,ym)` within an element of the background mesh and then interpolating the value of the mesh parameter from the nodal mesh parameters δ_n of the background mesh element; `perpcord` generates a line from `(xm,ym)` of length $\delta * \sin 60$ to the left of the segment and divides it into five equal parts by placing intermediate nodes whose coordinates are stored in a two dimensional array **ccc**. The function `nearnode` checks all the nodes on the front (except those on the reference segment) with respect to their distances from the tip of the perpendicular chord generated by `perpcord`, the nodes equal to or closer than δ being stored in the vector **node**. The function `pt_dist` identifies when another segment of the front is close to the reference segment, as shown in Fig. 5, and returns a $+1$ value if $d_1 > d_2$. If no *near node* has been selected and `pt_dist` returns '1' then the condition for all nodes less than or equal to δ units away from the tip c(0) of the perpendicular chord is relaxed for the selection of *near nodes* and function `nearnode_ult` is called. This selects the *near nodes* as the first twenty nodes closest to the tip of the perpendicular chord.

The function `selnode` selects a node for the formation of the apex of the triangle from the list of *near nodes*: i.e. **node**, on the front and the nodes generated on the perpendicular chord **ccc**. This function also checks that the *near nodes* are only on the left side of the segment and that the generated sides of the triangle do not intersect any side of the current front. In an extreme case, when the values of the mesh parameter show large variations over a short distance, then it is possible that `selnode` can not select a node. The function then returns a -1 value and the program then calls `selnode_drs` which obliges the program to select a node from the existing front or from new nodes, as the case may be. The only consideration here is that the selected node must not form an element the sides of which intersect with any segment of the front. The element in this case may have an inconsistent shape. The function `prune` removes the inactive sides from the front. The inactive sides on the front are the ones which cannot form sides of a new triangle.

The above steps are repeated until the number of segments in the advancing front, `nsgt`, become equal to zero.

3 Post-processing of the mesh

The post-processing procedures from [4] for restoring the consistency of the mesh generated are given below.

3.0.1 Diagonal exchange

After the generation of the mesh, first post processing is undertaken by performing diagonal exchange between two elements where necessary. Each element of the generated mesh is considered in turn and the element adjacent to its longest side identified. The minimum angle from the angles subtended by the pair of elements is determined. The diagonal is swapped. Again, the minimum angle for the new topologies is calculated and if the minimum angle is found larger in the latter case then the new topology is adopted; otherwise the old topology of the elements is maintained.

3.1 Smoothing

For each interior node of the generated mesh all the nodes connecting through an element side are identified. The x and y coordinates of all these nodes are averaged and these averaged nodal coordinates are assigned as the new coordinates for the node under consideration. This process is repeated a number of times for all the interior nodes of the mesh. In the examples presented in this paper the smoothing was done five times for each new mesh.

4 Adaptivity

4.1 Introduction

In the case of finite element idealizations based upon simple constant-strain triangular elements the accuracy of the results depends upon the topology of the mesh chosen for the finite element analysis. Adaptive remeshing can predict an efficient mesh by taking into account the domain error for a uniformly graded mesh. An efficient mesh is defined as the one in which the error is equally distributed over the domain.

Through adaptive remeshing the domain error is reduced as well as uniformly distributed over the domain. The analyses are carried out until the domain error becomes less than a pre-defined error value [1].

4.2 Domain Error

For a constant-strain triangular element the elemental stresses obtained from a finite element analysis are generally defined as the stress resultants at the centroid of the element which satisfy the equilibrium and compatibility requirements. This element therefore gives no further information on the distribution of the stresses within the element. This leads to stress discontinuities at the boundary with adjacent elements whereas in a real continuum there is no such sudden variation of stresses at arbitrarily chosen points within the continuum.

From these and other considerations it is clear that the results obtained from the finite element analysis can, at best, be a close approximation to the actual stresses in the material.

To achieve a finite element solution close to the actual response of the material it is assumed that if the continuum is idealized using a large number of elements, such that the size of each individual element is very small in comparison to the dimensions of the continuum, then the actual element stresses will tend to be constant over the sub-domain of each infinitesimal element and the solution of the finite element method may be regarded as accurate for all practical purposes.

This strategy has the disadvantage that such an approach would be computationally very expensive and time-consuming. So graded meshes comprising variable element sizes are used. The object is to provide more elements in the regions where high variation of stresses is anticipated. Although this approach is workable but it requires considerable effort and skill in the design of accurate finite element meshes.

4.3 Smoothed stresses from nodal averaging

In the process of nodal averaging, the element stresses $\underline{\sigma}$ from all the elements common to the node are averaged and this is repeated for all the nodes of the mesh. The vector $\underline{\sigma}$ consists of the element stresses $\{\sigma_{xx}, \sigma_{yy}, \sigma_{xy}\}^T$. Then for each element a smoothed value, $\hat{\sigma}$, is determined by averaging the three nodal stress values. This can be expressed as

$$\underline{\sigma}_n = \sum_{nn} \frac{\sum_{i=1}^{ns} |\underline{\sigma}|}{ns} \tag{5}$$

$$\underline{\hat{\sigma}} = \sum_{ne} \frac{\sum_{i=1}^{3} \underline{\sigma}_n}{3} \tag{6}$$

where nn is the number of nodes in the mesh, ns the number of elements connecting to a node and ne the number of elements in the mesh.

4.4 Equations of adaptive remeshing h-refinement

The standard finite element problem may be summarised by the following equation:

$$\mathbf{Ku} - \mathbf{f} = 0 \tag{7}$$

where

$$\mathbf{K} = \int_\Omega \mathbf{B}^T \mathbf{D} \mathbf{B} \tag{8}$$

$\mathbf{f}$ = applied load vector.

$\mathbf{u}$ = nodal displacement vector.

In adaptive calculations the displacement error is:

$$\mathbf{e} = \mathbf{u} - \hat{\mathbf{u}} \tag{9}$$

The error in the stresses:

$$\mathbf{e}_\sigma = \underline{\sigma} - \underline{\hat{\sigma}} \tag{10}$$

where $\underline{\sigma}$ and $\underline{\hat{\sigma}}$ are the element stresses from finite element analysis and nodal averaging, respectively. The error in the strain may be determined from the error in the stresses:

$$\mathbf{e}_\epsilon = \mathbf{D}^{-1} \mathbf{e}_\sigma \tag{11}$$

The error energy norm is defined to be

$$\|e\| = \left(\int_{\Omega} (\mathbf{Be})^T \mathbf{D} (\mathbf{Be}) d\Omega \right)^{\frac{1}{2}} \tag{12}$$

Substituting equation (11) in equation (12) we obtain the following expression for the energy norm:

$$\|e\| = \left(\int_{\Omega} \mathbf{e}_{\sigma}^T \mathbf{D}^{-1} \mathbf{e}_{\sigma} d\Omega \right)^{\frac{1}{2}} \tag{13}$$

The total energy norm is given as

$$\|u\| = \left(\int_{\Omega} \underline{\hat{\sigma}}^T \mathbf{D}^{-1} \underline{\hat{\sigma}} d\Omega \right)^{\frac{1}{2}} \tag{14}$$

All these norms have been defined for the whole domain. In practice the norms for each individual element of the mesh are calculated and summed:

$$\|e\| = \sum_{i=1}^{ne} \|e\|_i{}^2 \tag{15}$$

An adaptivity control parameter η is defined to quantify percentage error

$$\eta = \frac{\|e\|}{\|u\|} * 100 \tag{16}$$

If $\overline{\eta}$ is the limit on error then, while

$$\eta \quad > \quad \overline{\eta} \tag{17}$$

the mesh element refinement parameter is defined as

$$\xi_i = \frac{\|e\|_i}{e_m} \tag{18}$$

where

$$e_m = \overline{\eta} \left(\frac{\|u\|^2}{ne} \right)^{\frac{1}{2}} \tag{19}$$

The mesh is refined according to

$$h_{new} = \frac{h_i}{\xi_i} \tag{20}$$

until

$$\eta \leq \overline{\eta} \tag{21}$$

where h_i and h_{new} are the previous and the new element sizes determined from the adaptive analysis. The symbol h_{new} corresponds to $\delta(i)$ in sub-section 2.1.

4.5 Discussion on some approaches to adaptivity

The adaptive remeshing method shown above may be divided into three parts; namely, (i) calculation of smoothed stresses, (ii) calculation of norms for representing the domain errors and (iii) transformation of the domain errors to mesh parameters, for remeshing.

For the first part (i) Sibia and Hinton [3] have used the method of nodal averaging as described previously. Zienkiewicz and Zhu[1] have used a more sophisticated approach wherein the smoothing is done over the sub-domain of each element by assigning the same interpolation function as the element displacement function. This is shown in equations (16), (17) and (18) of reference [1].

With regard to the second part (ii), the error energy norm $\|e\|$ is determined in the same way in both the papers, [1, 3], and the notation used is also the same. However in the paper by Zienkiewicz and Zhu [1] the total energy norm $\|u\|$ has not been clearly defined. It is, however, assumed that it carries the same meaning as $\|w^*\|$ in reference [3]. The norm $\|w^*\|$ of reference [3] is synonymous with $\|u\|$ in this text.

For the third part (iii) different approaches have been adopted in [3] and [1] for evaluating the term e_m. In equation (8) of Sibia and Hinton [3] this term is called $\overline{\rho}$ and is taken as $\overline{\rho} = \overline{\eta}\|w^*\|/n^{\frac{1}{2}}$ where n is the number of elements in the domain.

Zienkiewicz and Zhu [1] have calculated this term on a "Root Mean Square" basis as shown in equation (24) of reference [1] and equation (19) of this text. Both these approaches have been tested for constant strain triangular element and it appears that the approach in [1] gives better smoothness for the mesh generation parameters. It may be pointed out that if non–smooth values of mesh parameters, exhibiting large variations in parameter values in local domains, are encountered then this may lead to the breakdown of the mesh regeneration algorithm.

Component specifications	No. of units
IMST800 transputer	12
TTM18 TRAM with 8MB dynamic RAM	1
TTM3 TRAM with 1 MB dynamic RAM	11
Transtech mother board TMB08	1
Transtech mother board TMB12	1
Transrack for housing TMB12	1

Table 2:

5 Hardware configuration

The description of the hardware used in the development and running of the parallel adaptive mesh generator is given in Table 2. All the daughter boards TTM18 and TTM3 support IMST800 transputers. The TTM18 and TTM3 daughter boards have 8 MBytes and 1 MBytes of on-board memory respectively. The TTM18 daughter board was plugged into the TMB08 mother board and acted as the `root`. The TMB08 mother board was fixed into the expansion card slot of a PC-AT compatible. The Down, ConfigDown and the PipeTail [11, 12] from the TMB08 were connected to the UP, ConfigUP and PipeHead of the TMB12. The TMB12 was populated with eleven TTM3 TRAMS acting as slaves to the `root` transputer. The INMOS Link speeds [13] on both the mother boards were set to high i.e. 20 Mbits/second. The TMB12 was mounted in a Transrack. The Transrack has provision for ten large slots for TMB12 mother boards. Each board may be populated with up to 16 TTM3 daughter boards. In theory, therefore, the Transrack may support 160 TTM3 daughter boards.

The transputers are connected in series with link 2 of the first transputer connected to link 1 of the second transputer and so on. This linear serial configuration is present in hard-wired form on the TMB08 and TMB12 mother boards. Further information on the default pipelines of TMB08 and TMB12 mother boards may be obtained from references [11, 12].

6 Parallel Compiler

OCCAM [14] is a language developed for supporting concurrent processes. Transputer systems may be designed and programmed using OCCAM which allows setting up of concurrent processes, for applications, communicating through channels. Since OCCAM was developed for the transputer environment it would therefore be reasonable to assume that OCCAM would be a strong candidate for writing a parallel transputer-based code. However, with the availability of parallel compilers for popular languages like C and FORTRAN, which support most of the standard features of their sequential counterparts, it requires a great deal less effort to create a parallel version if the sequential code has already been developed in C or FORTRAN.

The sequential code for un-structured mesh generation was written in C. The 3L Parallel C compiler [16] was therefore selected for the development of the parallel code.

Since Parallel C carries nearly all the features of ANSI C and the functions for parallel operations are in addition to the standard features, it is possible to develop components of the parallel code on a main-frame or a mini-computer using the superior debugging facilities that are usually available on such systems.

Debugged and tested segments of the code may therefore then be assembled using the parallel features of the language. In this way the use of the parallel debugger Tbug [15], which runs in the PC environment, may be limited to the checking of parallel features of the code only. This leads to a great deal of saving of the programer's time as Tbug operates at much lower speed and carries limited facilities as compared to sequential debuggers.

7 Parallel Algorithms

7.1 A Brief Introduction to parallel algorithms

As the subject of parallel algorithms is in its infancy, general classifications are difficult to form. However, successfully implemented parallel algorithms on Multiple Instructions Multiple Data machines such as transputers, may be classified broadly according to [14].

1. **Processor Farming** in which each processor in the network carries the same code and executes it in isolation from all the other processors. This is also termed **Independent Task / Event Parallelism**.
2. **Geometric Parallelism** in which each processor executes the same code on **data corresponding to a sub-region of the system** being simulated and communicates boundary data to neighbouring processors handling neighbouring sub-regions.
3. **Algorithmic Parallelism** in which each processor is responsible for **part of the algorithm**, and all the data passes through each processor.

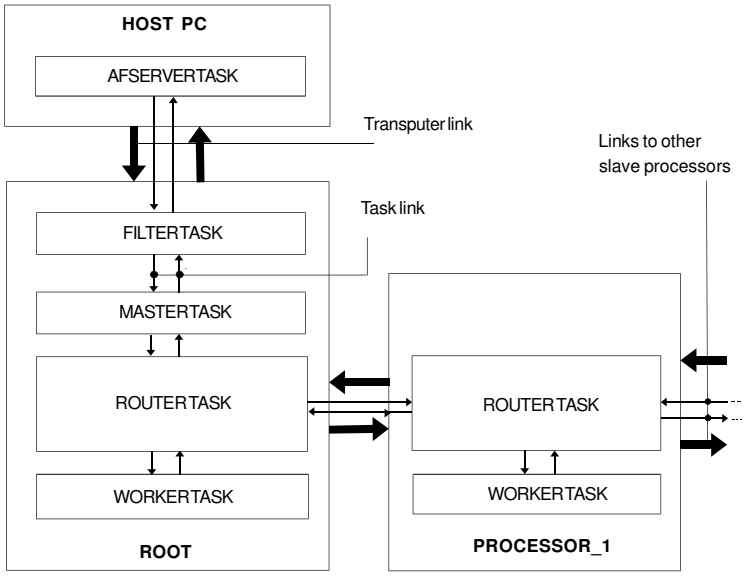

Figure 6: A flood-fill representation

7.2 Processor farming in 3L Parallel C

The processor farming environment in Parallel C [16] is generated by the flood-fill configurer. The flood-fill configurer provided with 3L Parallel C compiler enables the programmer to implement a processor farming application which is completely portable over transputer networks having different number of transputers and/or topologies. Any flood-fill application must have the following program structure:

1. A `master` task to divide the job into independent work packets;
2. A `worker` task which is loaded on to each node (processor) of the network; and
3. A configuration file describing the memory requirements and other attributes of the tasks.

The flood-fill configurer creates the processes for the `master` task and a `worker` task on the root transputer. The remaining transputers of the network are each loaded with the `worker` task image. The connections between the `master` and `worker` tasks are made through interfacing `router` tasks, as shown in Fig. 6.

The flood-fill application may be run on different transputer topologies without making any changes to the configuration file or re-configuring the bootable file [16]. Thus the source codes produced using the flood-fill configurer are completely portable to different transputer configuration topologies.

It is also possible to build up a configured application around a flood-fill code. In such a case a configuration file giving the transputer network topology and the placing of the tasks on different transputers, must be specified. These configured applications have to be compiled using the Parallel C general purpose configurer instead of the flood-fill configurer. This is particularly useful when fewer transputers in the network are to be configured for running the application, without physically disengaging them from the network.

In Parallel C terminology the processes which may function in parallel are either threads or tasks. The threads of a single task run on the same processor. They may share data, being on the same processor, and may communicate either by using channels as tasks, or shared memory. The tasks on the other hand have their own code and data areas and runtime library functions. The tasks communicate only through the specified channels and do not share data or runtime library functions even when placed on the same processors.

8 Identification of parallelism within the algorithm

The advancing front method, as described in section 2, relies heavily on the searches made over the background mesh and the advancing front. As the number of elements of the background mesh and the relative size of the domain increase this searching process becomes computationally expensive. It is also noted that mesh generation may be carried out on different sub-domains independently. If the domain to be remeshed is considered as comprising loops of external and internal boundary segments, then each sub-domain may be visualized as an external boundary loop occurring completely within the remeshing region of the domain. The area enclosed by the external and internal boundary loops of the domain is called the remeshing region.

These sub-domains fully cover the remeshing zone of the domain and the boundary segments of these sub-domains do not intersect with one another.

Computational advantages may be derived through parallelism in the following ways:

(a) by adopting parallel searching techniques; or

(b) by dividing the domain into suitable sub-domains and mapping these sub-domains onto different processors.

Scheme (a) would retain all the characteristics of the original algorithm but the actual process of boundary discretization and that of element generation would remain sequential. In scheme (b) the actual mesh generation process would be done in parallel and the algorithm would not require communication between slave processors, as each individual slave processor would be entrusted with a separate sub-domain. However, the generation of elements larger than the specified sub-domain would not be possible in this case.

For the implementation of this scheme account was taken of the following points:

1. The discretized background mesh, representing the domain, has to be divided into separate sub-domains in order to be mapped on to different processors. The size and shape of each sub-domain should be such as to ensure adequate load balancing of the processors for optimizing efficiency.
2. The integrity of the boundary connectivities between the meshes generated on separate processors has to be guaranteed.
3. After the parallel processing of the sub-domains the assembly of the sub-domain meshes to form the full domain mesh must be considered

As mentioned above, the execution rate of the advancing front technique depends heavily on the searches made over the background mesh elements and the segments of the advancing front. So if scheme (b) were to be selected then the computational advantage would be two-fold :

1. Owing to the division of the domain into smaller sub-domains the advancing fronts of each sub-domain would have smaller number of segments as compared to the case if the whole domain was idealized as a single loop. Thus the geometric searching effort required to generate an element within a sub-domain would be reduced in proportion to the ratio between the size of the sub-domain and the domain.

   From the theory presented in section 2, the mesh parameter δ is specified for each element of the background mesh. If the division of the background mesh was done element-wise, meaning that each element of the background mesh was treated as a separate sub-domain, then for all such triangular sub-domains the δ and δ_n values would be known at all times. Thus, the requirement to search over the background mesh, to determine the local values of mesh parameters, would be eliminated.
2. As each sub-domain undergoes independent generation of elements, the actual process of element generation would occur simultaneously, in different regions of the domain.

Scheme (b) therefore provides computational advantages owing to the smaller number of segments of the advancing front per sub-domain and Independent Task / Event Parallelism. As the scheme does not require inter-processor communication a processor farming environment generated by the flood-fill configurer [16] was selected for the development of the scheme. This also added to the simplicity and portability of the code.

9 Sub-domain discretization

A two dimensional domain is divided into triangular elements of similar size. This initial mesh is termed the background mesh. Fig. 7 shows a coarse background mesh comprising only two elements for remeshing of a square domain. For each element of this background mesh a mesh parameter value δ is specified. From equation (1) the nodal mesh parameters $\underline{\delta_n}$ are calculated.

As each sub-domain is to be meshed individually on a separate processor and the topology of each sub-domain is known to be triangular, the boundary nodes in all the sub-domains may be numbered identically and in the same order, as shown in Fig. 7. This eliminates the need for the transmission of the information on the number and the connectivities of the boundary segments of each sub-domain. Each processor is "aware" of the fact that each sub-domain which it will receive will comprise three nodes and it accordingly assigns the boundary connectivities in a counterclockwise manner. In order to initiate mesh generation on any processor, only the nodal coordinates and the nodal mesh parameter values need to be transmitted from the `master` task. If the actual node numbers and the topologies of each sub-domain had been sent to the processors, then inter-processor communication would be required every time a new node was created within any sub-domain. The simple representation shown in Fig. 7 allows independent generation of nodes and elements within each sub-domain without the restriction of an overall node numbering protocol. All the nodes within each sub-domain are numbered from '1' onwards. It becomes the task of the overall mesh assembler to successfully knit together the elements received at random from the `worker` tasks, so as to form a coherent mesh. This is done by scanning the nodal coordinates of each generated element and comparing them with those of the previously received and compiled elements.

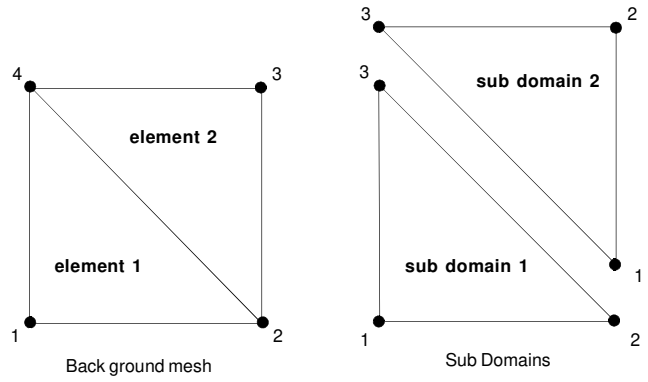

Figure 7: Representation of a background mesh and triangular sub-domains

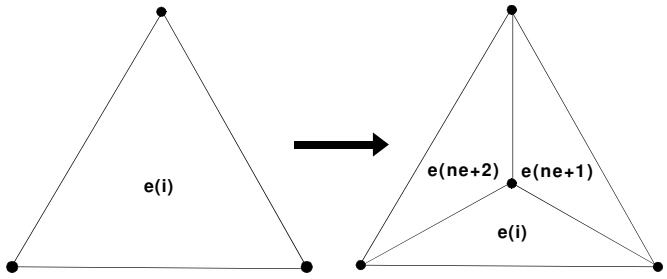

Figure 8: Sub division of an element $e(i)$ of a background mesh comprising ne elements (sub-domains)

10 Load balancing for the triangular sub-domains

The *flood-fill* configurer is designed on the assumption that the number of *messages* sent is very large compared to the number of slave processors. Information concerning a single sub-domain constitutes a message. If the `router` tasks at each node of the network receives two or more than two messages, one is immediately handed over to the associated `worker` task and the other is buffered pending completion of the processing of the first message by the `worker`. The remaining messages are sent to the next `router` task for similar action. Hence if the number of messages are less than twice the number of slave processors then it is not possible to load and initiate all the processors in the network.

In the case of the first remeshing, the background mesh is usually specified manually. In this case the number of messages and the aspect ratios, which may be defined as the ratio of the average δ_n and the average length of the sides of the background element under reference, may be quite low. This may lead to the following undesirable situations.

1. It may not be possible to initiate all the available processors.
2. Few processors may be performing a sufficiently large quantum of calculations owing to low aspect ratio for the sub-domains transmitted to them.

Therefore to achieve load balancing, steps are required to ensure that:

1. the number of processors is less than half the number of sub-domains; and
2. the computational load (time) of each sub-domain should not be trivial in comparison with the inter-processor communication time.

Owing to the specification of the sub-domains, the mesh generation algorithm dispenses with the requirement of background mesh searching for δ_n values; but it still requires to scan the active sides of the advancing front for geometric intersections between the sides of the generated elements and the active sides of the front. It is therefore computationally advantageous to keep the number of segments within a front of a sub-domain to the minimum. On the other hand, if only a single element were to be generated, within a sub-domain, then it would be a waste of time to send such a sub-domain for remeshing to a `worker` task. Ideally the `worker` tasks should therefore receive sub-domains which would generate an equal number of elements and the elements which thus result within each sub-domain would form an appropriate fraction of the overall generated mesh. For instance, if it is known before hand that a mesh of 500 elements was to be generated and that there were 5 `worker` tasks, then the ideal domain discretization would be to divide that domain into 50 sub-domains and size of each sub-domain so that it generates exactly 10 elements. However, when dealing with meshes resulting from *a posteriori* error estimates, it is not possible to know the exact number of elements that will be generated. A compromise therefore has to be made between ideal load balancing and the extent of load balancing that may be achieved by not adding excessive computational overheads.

The load balancing techniques outlined in this section target the sub-domains having either the ability to generate large number of elements or the inability to generate any elements at all. The prolific elements are sub-divided subject to certain geometric preconditions and the non-generative elements are are sent straight away for compilation without being sent to the `worker` tasks.

In order to alleviate the load imbalancing situations discussed above, the background mesh is processed through two mesh filters. These filters check the background mesh (sub-domains) for their ability to generate large number of elements or no elements at all, within each sub-domain. The filters have therefore been termed "Prolific Element Filter" or PE filter and "Non Generative Element Filter" or NGE filter, respectively.

The PE filter locates the sub-domains exhibiting the potential for generating relatively large number of elements and performs sub division as shown in Fig. 8.

An arbitrary value for the aspect ratio, say Fac, depending upon the size, shape and the nodal mesh parameter values for the sub-domain, is fixed at the beginning of the procedure. A useful trial value is 0.2 which has been taken from the range of $0.1 - 0.3$. The user can however select a different value depending upon the magnitudes of the nodal mesh parameter values, size and conditioning of the sub-domain.

The average nodal mesh parameter value $(\delta_n)_{avg}$ and the average length of the sides L_{avg} for the triangular sub-domain are calculated. The minimum angle for the sub-domain θ_{min} is determined.

$$ratio = \frac{(\delta_n)_{avg}}{L_{avg}} \tag{22}$$

$$Fac_1 = Fac(\frac{\theta_{min}}{30} - 1) \tag{23}$$

If

$$ratio \leq Fac_1 \tag{24}$$

then the sub-domain is divided into three equal area sub-domains as shown in fig 8. This subdivision is accomplished by generating a node at the centroid of the triangular sub-domain. One of these three sub-domains is copied in place of the original sub-domain in the background mesh and the remaining two sub-domains are added to the background mesh as shown in Fig. 8. Thus for each subdivision of the element of the background mesh, the total number of elements of the background mesh increases by two.

Equation (23) ensures that no subdivision will take place if the minimum angle of the sub-domain is less than or equal to 30 degrees. Equation (23) also linearly and progressively reduces the right hand side of the inequality of equation (24). In other

words, the subdivision of a sub-domain is discouraged for θ_{min} values less than 60 degrees. This is done to avoid the formation of very flat triangular sub-domains (generally having minimum angles less than the 15-20 degree range), within the sub-domain under consideration.

The NGE filter is designed to locate sub-domains which will not generate mesh elements owing to their relatively higher nodal mesh parameter values as compared to their sub-domain sizes.

For each sub-domain (element) of the background mesh all three sides of the triangular sub-domain are checked for the nodal mesh parameter values at each end. The lesser of the two end values for each side of the triangular sub-domains are stored as $(\underline{\delta_n})_{min}$.

If each entry in the $(\underline{\delta_n})_{min}$ vector is greater than 51% of the length of the corresponding side of the sub-domain, then from the theory presented in sub-section 2.1 it may be concluded that no intermediate node generation will take place on any of the boundary sides of the sub- domain. Hence no element generation would take place within the sub-domain. The NGE filter guarantees that the selected element will not undergo mesh generation within its sub-domain. It marks such an element and stores it in the location where the new mesh would be assembled. This filter does not however eliminate the possibility that some other elements which have passed through the filter unmarked, shall not undergo mesh generation.

11 Compatibility between adjacent sub-domains

Since the adjacent elements (sub-domains) of the background mesh share the same nodal mesh parameter values δ_n at the common nodes, and the discretization of the boundary segments of the sub-domains is carried out from the higher value of δ_n to its lower value in equation (2), the common boundaries of the adjacent sub-domains will always have the same number and distribution of the intermediate generated nodes. Thus the connectivity of the adjacent sub-domains is guaranteed even when they are processed in isolation from each other.

12 Master Task

In accordance with the flood-fill configuration requirements the parallel mesh generator is written in the form of a `master` task and a `worker` task. The `master` task performs the following functions.

1. The function `main` of the `master` task creates two parallel processes (threads), arbitrarily named as `bound_send` and `bound_receive` for handling the communication with the `worker` tasks. These threads are synchronized using a semaphore [17].
2. The thread `main` of the `master` task applies the PE and NGE filters to the background mesh and this is called the *preprocessing* procedure.
3. The thread `bound_send` contains the function `net_broadcast` [1] which transmits the information packet pertaining to each sub-domain. Only the sub-domains which have remained unmarked while passing through NGE filter are transmitted. The `router` tasks distribute these packets among the idle `worker` tasks. Whenever the network of `worker` tasks becomes fully engaged the `net_broadcast` is blocked until a `router` task with an empty buffer becomes available.

   The information on the nodal coordinates and the nodal mesh parameters δ_n of a sub-domain is read into a *structure* type buffer by the thread. The function `net_broadcast` relays the information held in the pointed buffer in packets of 1024 bytes. The total length of the message in this case works out as less than 100 bytes hence each message is carried within a single packet. Thus the terms *message* and *packet* have been used in a synonymous sense.
4. The thread `bound_receive` contains the function `net_receive`. The results from the `worker` tasks are received and `net_receive` waits until the next packet from the `worker` task becomes available. The `net_receive` is performed within a never ending loop.
5. In the case of a break down of the mesh generation algorithm within a `worker` task, an error message is received and displayed by the `master` task.
6. The `main` executes the function `thread_deschedule` within a loop while the worker tasks are active and are receiving and sending the information packets through `net_broadcast` and `net_send` to the `main`. This function causes `main` to be momentarily unable to execute (usually for one timer tick); causing `main` to be de-scheduled from the processor, thus allowing some other thread to resume execution in its place. This prevents `main` from hogging the processor to the detriment of the other two threads. In effect a very low priority level is achieved for the thread `main` and the benefit of processor scheduling is given to the `bound_send` and `bound_receive` threads.
7. The information packets received from the `worker`s are fitted into the overall mesh by `bound_receive`.
8. The post processing of the mesh by diagonal exchange and smoothing, as described in [4], is undertaken by `main` of the `master` task after all the sub-domains have been received back from the `worker` tasks.

[1] The function `net_broadcast` is a new prototype version of the `net_send` function currently available with the 3L Parallel C Version 2.1 compiler

The function net_broadcast and net_receive are performed in parallel with the thread main of the master task. This prevents the incoming result packets being blocked by the net_broadcast waiting for a worker to become free, or net_broadcast being blocked by net_receive waiting for output from the worker tasks.

12.1 Representation of the function main of the master task

```
sema_init(&green,0);
/* begin time count for preprocessing */
time(&tima);
/* average the bg mesh para at each node */
node_para();

store_bg(&n_iln); /* store bg mesh */
signal = 0;
/* count time at the end of pre- process  & starts
        for parallel processing */
time(&timb);

thread_create(bound_send,1000,1,bc);
thread_create(bound_receive,1000,0);

sema_signal(&green);

/* thread bound_send and bound_receive are activated here */
/* wait here till results are received */
while(signal < (ne_bg - knt_bg) ) /* less the els already p_stack */
{thread_deschedule();

if(ck_fail < 0)
{printf("Packet no %d failed at elem no = %d\n",id_num,-1*ck_fail);
ne_bg = ne1; knt_bg = 0; write_info();
exit(1);}
}
/* count time at end of parallel process */
time(&tim2);

un_load_st();

interdiag();

search_node();
for(i = 0; i < 5; i++)
smooth1();
```

The threads are synchronized by the semaphore green. The function sema_init assigns a starting value to the semaphore green as '0'. A thread may only execute through a semaphore when the semaphore value is greater than zero. The function time returns the number of seconds elapsed since a datum which is taken as 00:00:00 GMT on 1st of January 1970 by the Parallel C compiler, according to the host system clock. The function node_para calculates the δ_n using equation (1). The function store_bg stores the background mesh and also applies the PE and NGE filters. The integer variable knt_bg is equal to the number of the elements of the background mesh marked by the NGE filter. The integer variable signal is increased by one each time a worker task indicates that it has completed the remeshing of the sub-domain with which it was entrusted. The function thread_create establishes two threads for sending and receiving the packets to and from the worker tasks. As all necessary procedures stand completed at this stage sema_signal is called and bound_send which was waiting due to the '0' value of green, is activated. The bound_send sends the information packets to the worker tasks and the bound_receive becomes active as soon it receives an output from a worker. It then increments the signal by one and fits the received element into the mesh being compiled. During this period, when bound_send and bound_receive are active, the main loops over thread_deschedule. This looping operation has to be carefully designed to ensure that the main comes out of the loop as soon as the last information packet is received from a worker task. The functions interdiag and smooth1 perform the diagonal exchange and the smoothing operations for the compiled mesh.

The flow chart for the master task is shown in Figure 9

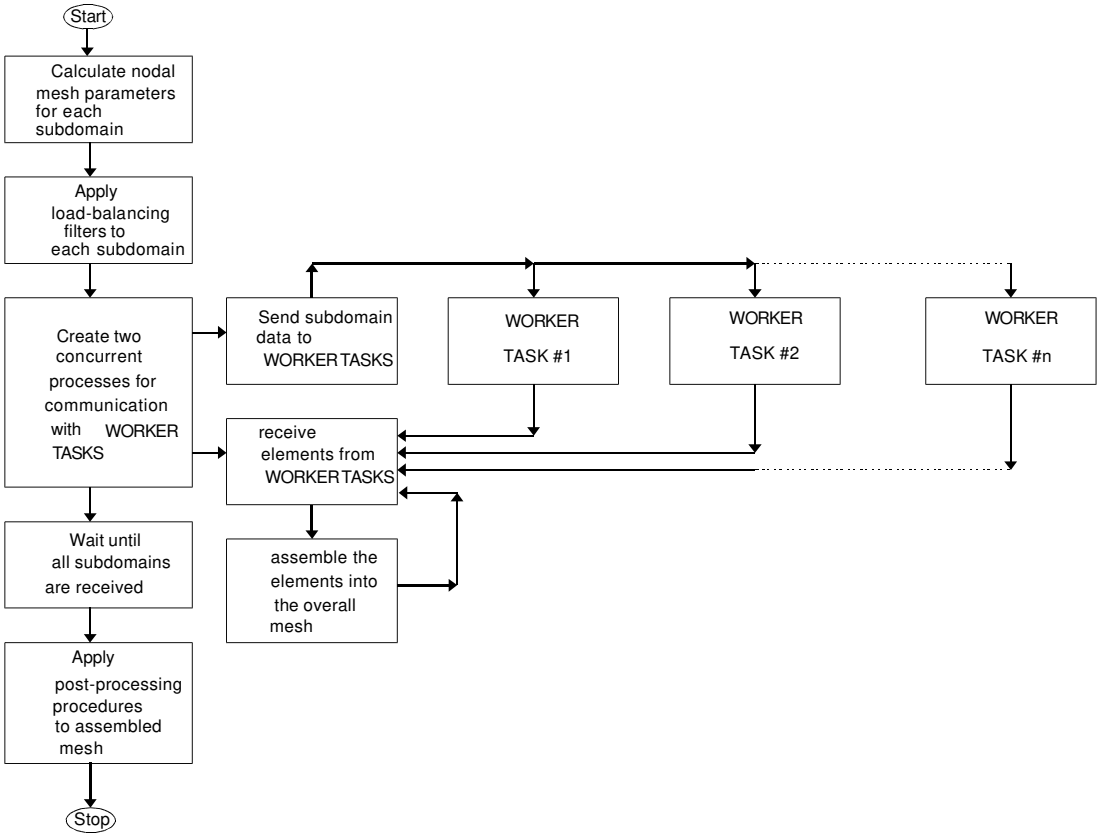

Figure 9: Master task for Adaptive Remeshing

Boundary segment no.	first node	second node
1	1	2
2	2	3
3	3	1

Table 3: Protocol for the representation of the boundary segments of a sub-domain within a `worker` task

13 Worker Task

The `worker` task in the flood-fill configuration is replicated at all the nodes of the transputer network. These tasks have only one two way communication channel, Fig. 6, and communicate with the `master` task through the `router` task or tasks. Inter-processor communication is not permitted in this type of configuration, and so the mesh generation is carried out at each node of the network without reference to the state of mesh generation in all the other processors.

The `worker` tasks of the parallel mesh generator perform the following duties.

1. The information on the nodal coordinates and the nodal mesh parameters is received through `net_receive`. These values are unpacked and copied into the data structure of the task. As it already stands determined that each sub-domain will be triangular in shape the task assembles the boundary front as shown in Table 3.
2. The generation of boundary nodes, the assembling of the advancing front and the generation of triangular elements within the sub-domain are carried out sequentially in accordance with the theory presented in section 2.
3. As soon as the node coordinates of the generated triangular element become available they are loaded into a *structure* type buffer and transmitted by the task using `net_send`. This strategy has proved to be superior to compilation of the mesh for the complete sub-domain followed by transmission of the mesh data in bulk. As in the former case there is a relatively longer time interval between each transmission of data, owing to the computational time taken between the generation of the mesh elements, the possibility of clogging of the net_work is prevented. In the second case the network may become clogged by transmission of a large number of packets in rapid succession.
4. When the number of active sides in the advancing front becomes equal to zero then the `worker` task communicates with the `master` task making it aware that the `worker` has successfully meshed the sub-domain, by sending an integer value of "1" which was previously held to zero during the mesh generation process. The `master` task counts these unit signals and proceeds to the next stage of mesh post-processing when it has received unit signals equal to the number of the sub-domains.
5. The `worker` task after signalling the completion returns to the receiving stage of the code to receive the next sub-domain.

14 Representation of the `worker` task

```
for(;;)
{

vali = net_receive(&ab, &ready);
/* load the information received */
vali = bsgt = 3;
vali = bdn = 3;
vali = mats = ab.mats;

j = 0;
for(i = 0; i < 3; i++)
{
j = i + 1; if(j > 2) j = 0;
vali = nod[0][i] = nod_bg[0][i] = i + 1;
vali = seg[i][0] = i + 1;
vali = seg[i][1] = j + 1;
val  = cord[i][0] = cord_bg[i][0] = ab.cords[i][0];
val  = cord[i][1] = cord_bg[i][1] = ab.cords[i][1];
val = hhi[i] = ab.hhis[i];
}
vali = nod_bg[0][3] = ab.mats;
boundnode(bc,cord_bg,nod_bg,i);
}
```

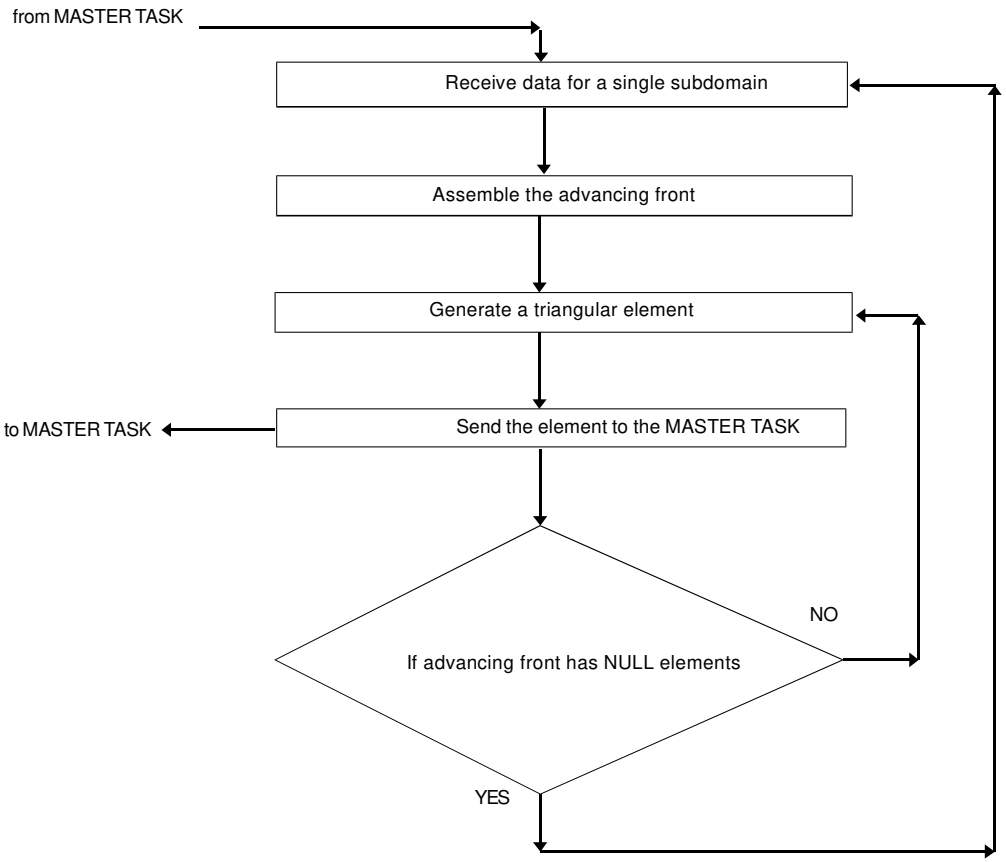

Figure 10: Worker task for Adaptive Remeshing

The `worker` receives the structure `ab` containing the node coordinates and the nodal mesh parameter δ_n of a sub-domain through `net_receive`. As already mentioned it assigns the boundary segment connectivities and numbering for a triangular domain. The boundary segment connectivities are kept in array `seg` which stores the node numbers for the first and the second ends of the boundary segment following the counter-clockwise numbering convention. The δ_n values for the sub-domain are un-packed from the structure `ab` and loaded into `hhi`. The function `boundnode` performs exactly the same operations as described in sub-section 2.2 with the exception that the search over the background mesh to find the local values of δ_n is not required. The function `trian`, called from within `boundnode`, contains `net_send` and every time a new element is generated within the sub-domain its nodal coordinates and the appropriate value of `signal` are loaded into a structure `cd` and transmitted to the `master` task.

The flow chart for the worker task is shown in Figure 10

15 Examples

Example 1, 2 and 3 show the application of the parallel mesh generator to adaptive finite element problems. The "*speed-ups*" defined as the ratio of the time taken by the parallel code to execute on a single transputer to the time taken to execute on n number of transputers, were plotted for each problem to record the response of the mesh generator to different remeshing problems.

Examples 4 and 5 show the comparisons of run times and the generated meshes, from parallel mesh generator (executed on four transputers) and a sequential version based upon the advancing front technique [2, 4]. The run times may vary for different types of PC `host` units.

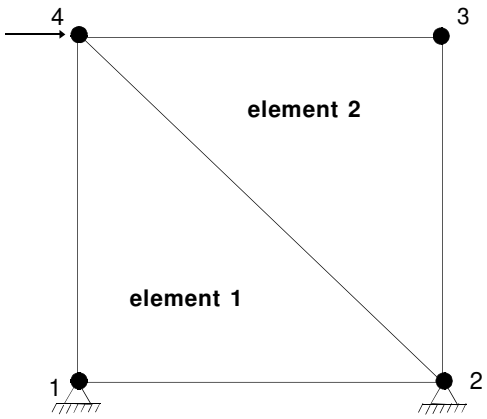

Figure 11: A square shear wall carrying a horizontal concentrated load

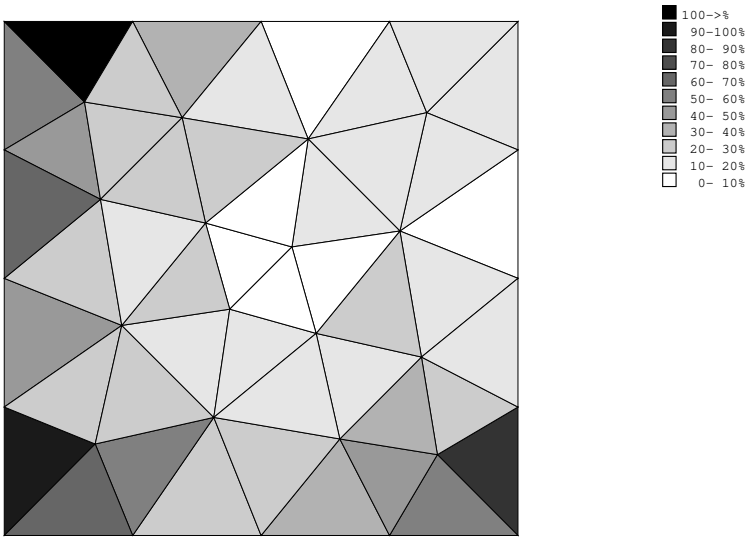

Figure 12: The first generated mesh and the error plot

Remeshing	ROOT		ROOT+3		ROOT+5		ROOT+7		ROOT+11	
	t_{pre}	t_{parr}	t_{pre}	t_{parr}	t_{pre}	t_{parr}	t_{pre}	t_{parr}	t_{pre}	t_{parr}
1st	0	6	0	3	0	3	0	3	0	3
2nd	1	60	1	15	1	12	1	11	1	9

Table 4: Run times for preprocessing t_{pre} and parallel mesh generation t_{parr} in seconds.

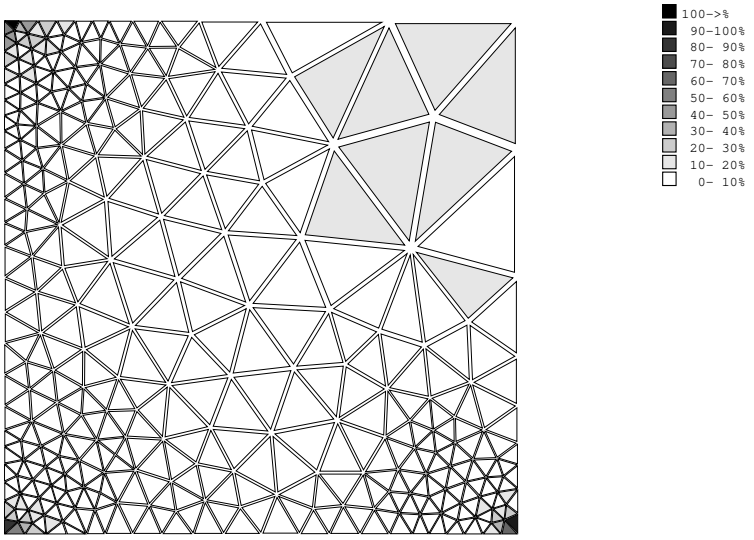

Figure 13: The second generated mesh and the error plot

Remeshing	ROOT		ROOT+3		ROOT+5		ROOT+7		ROOT+11	
	t_{pre}	t_{parr}	t_{pre}	t_{parr}	t_{pre}	t_{parr}	t_{pre}	t_{parr}	t_{pre}	t_{parr}
1st	0	4	0	1	0	1	0	1	0	1
2nd	0	24	0	7	0	6	0	5	0	5
3rd	2	12	2	3	2	3	2	3	2	3

Table 5: Run times for preprocessing, t_{pre}, and parallel mesh generation, t_{parr}, in seconds.

Example 1

The first remeshing was done with the background mesh comprising only 2 elements and 4 nodes as shown in Fig. 11. The mesh thus generated consisted of 32 nodes and 46 elements as shown in Fig. 12. From the adaptive analysis the overall percentage domain error, η, was 49.18%. The second remeshing with the mesh shown in Fig. 12 (of 33 packets) acting as the background mesh yielded 240 nodes and 412 elements as shown in Fig. 13. The overall percentage domain error η was reduced to 33.8% . The response curve for the second remeshing is given in Fig 14. The run times for the execution of the parallel code on 1, 4, 6 and 12 transputers for the first and second remeshings are recorded in Table 4

Example 2

The first remeshing was performed using the mesh shown in Fig. 15 (of 14 packets) as the background mesh. This resulted in a uniform mesh of 45 nodes and 56 elements as shown in Fig. 16. The adaptive analysis yielded overall percentage domain error $\eta = 57.63\%$. The mesh generator response curve is given in Fig. 17.

The second remeshing using the mesh shown in Fig. 16 (of 41 packets) as the background mesh resulted in 147 nodes and 244 elements as shown in Fig. 18. The adaptive analysis showed that the overall percentage domain error was 28.77%. The mesh generator response curve is given in Fig. 19.

The third remeshing using the mesh shown in Fig. 18 (of 148 packets) as the background mesh gave a mesh with 177 nodes and 298 elements. The overall percentage domain error , η, was further reduced to 25.45% as shown in Fig. 20. The mesh generator response curve is given in Fig. 21. The the run times for these remeshings are given in Table 5.

Example 3

Adaptive analysis were performed on the initial mesh comprising 72 nodes and 51 elements. This gave the overall percentage domain error $\eta = 31.46\%$. The first remeshing was performed using the mesh shown in Fig. 22 (of 53 packets) as the background mesh and it resulted in a mesh comprising 274 nodes and 476 elements. The overall percentage domain error , η, was reduced to 23.05%. The generated mesh is shown in Fig. 23. The response curve is given in Fig. 24. The run times are recorded in Table 6 for this example.

33 PACKETS; 366 ELEMENTS

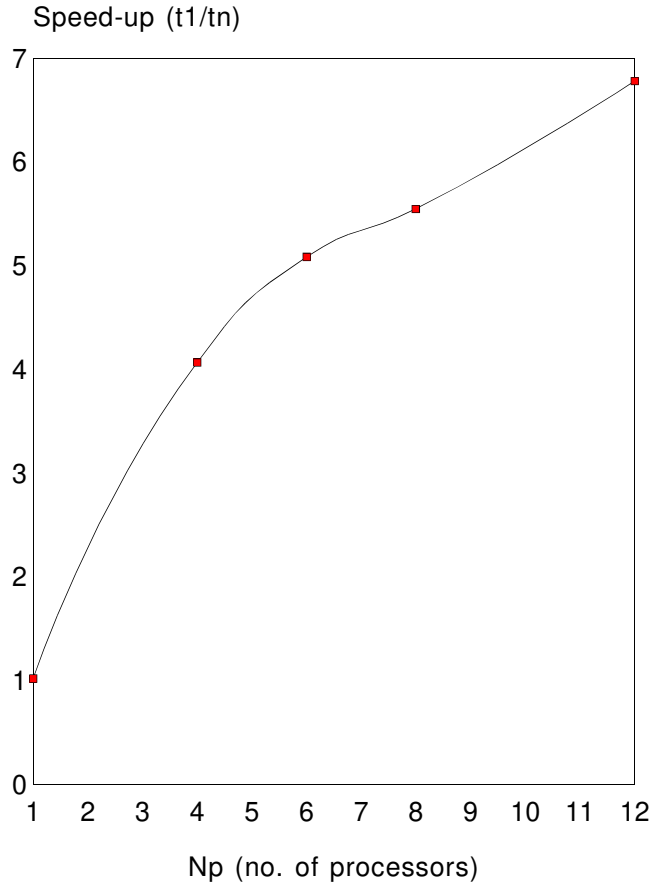

Figure 14: The mesh generator response curve for the second generated mesh of Example 1 with 33 packets and 366 generated elements as shown in Fig. 13

Remeshing	ROOT		ROOT+3		ROOT+5		ROOT+7		ROOT+11	
	t_{pre}	t_{parr}	t_{pre}	t_{parr}	t_{pre}	t_{parr}	t_{pre}	t_{parr}	t_{pre}	t_{parr}
1st	0	53	0	28	0	16	0	15	0	15

Table 6: Run times for preprocessing, t_{pre}, and parallel mesh generation, t_{parr}, in seconds.

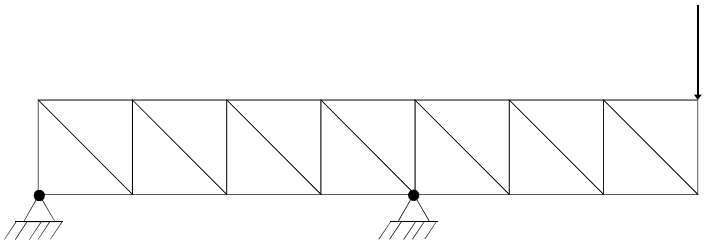

Figure 15: An overhanging beam with a vertical concentrated load at the end of the cantilever

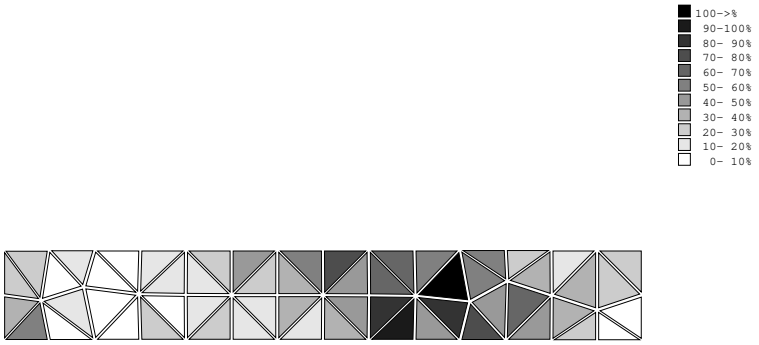

Figure 16: The first generated mesh with 45 nodes, 56 elements and $\eta = 57.63\%$

14 PACKETS; 56 ELEMENTS

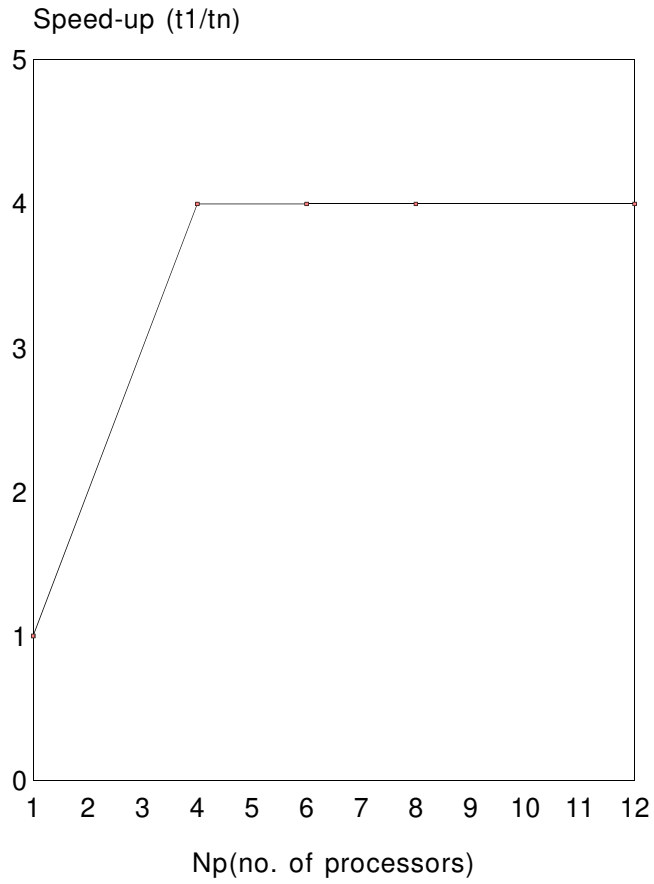

Figure 17: The mesh generator response curve for the first remeshing with 14 packets and 56 generated elements as shown in Fig. 16

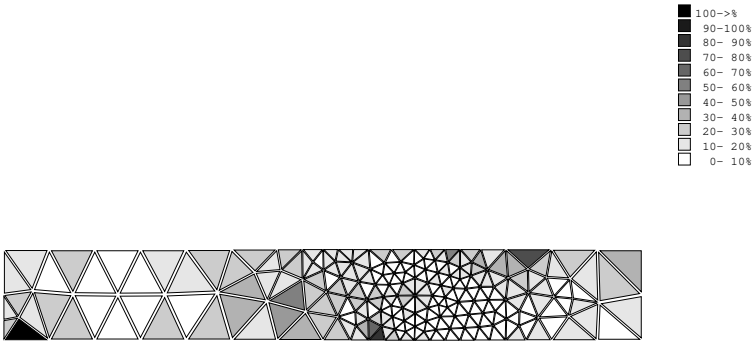

Figure 18: The second generated mesh with 147 nodes, 244 elements and $\eta = 28.77\%$

41 PACKETS; 188 ELEMENTS

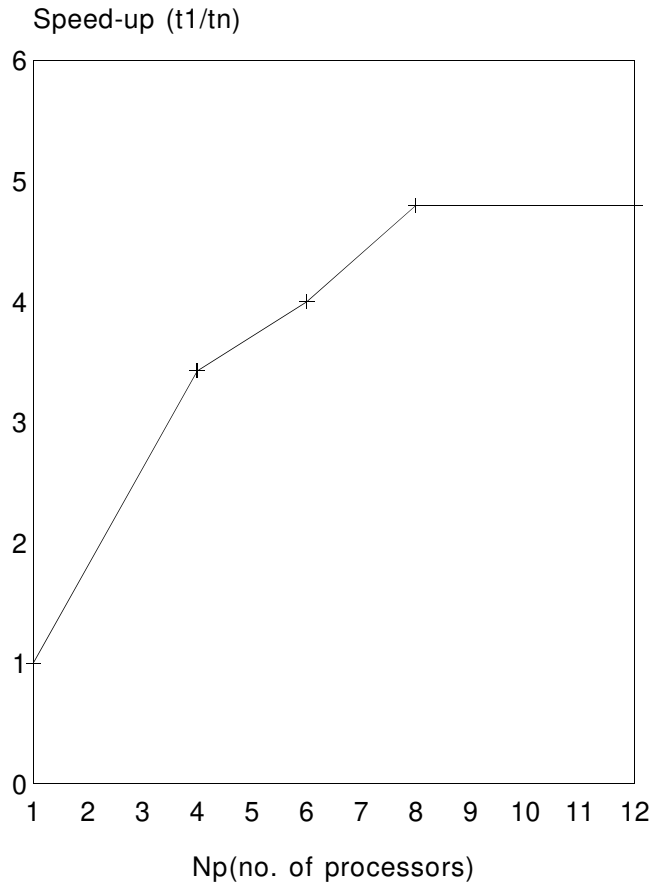

Figure 19: The mesh generator response curve for the second remeshing with 41 packets and 188 generated elements as shown in Fig. 18

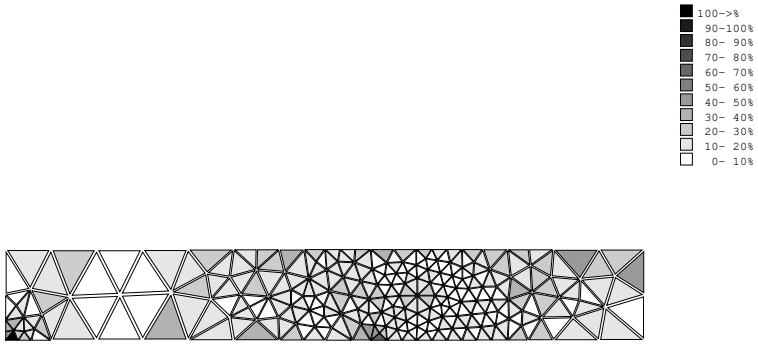

Figure 20: The third generated mesh with 177 nodes, 298 elements and $\eta = 25.45\%$

148 PACKETS; 54 ELEMENTS

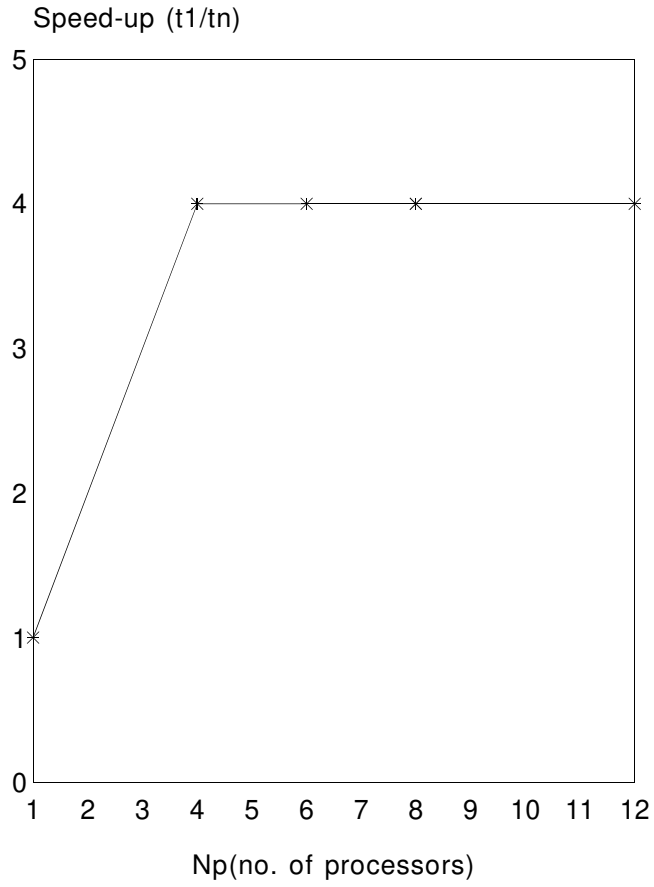

Figure 21: The mesh generator response curve for the third remeshing 148 packets and 54 generated elements as shown in Fig. 20

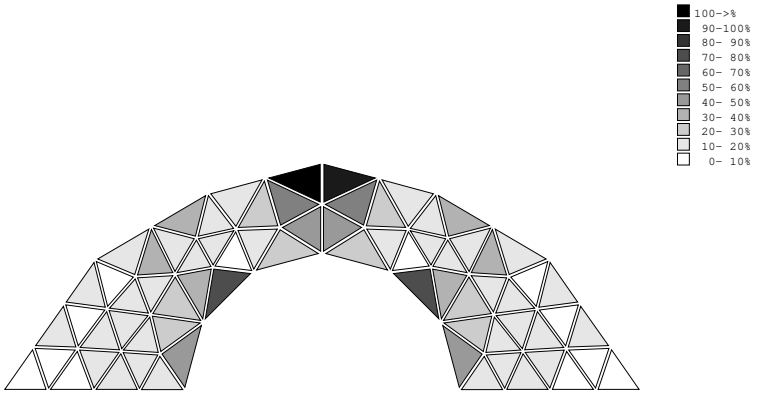

Figure 22: An arch carrying a vertical concentrated load at the crown with 72 nodes, 51 elements and $\eta = 31.46\%$

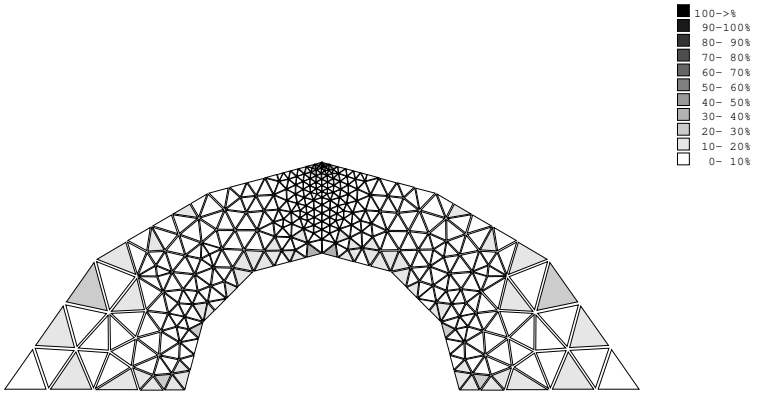

Figure 23: The first generated mesh with 274 nodes, 476 elements and $\eta = 23.05\%$

53 PACKETS; 404 ELEMENTS

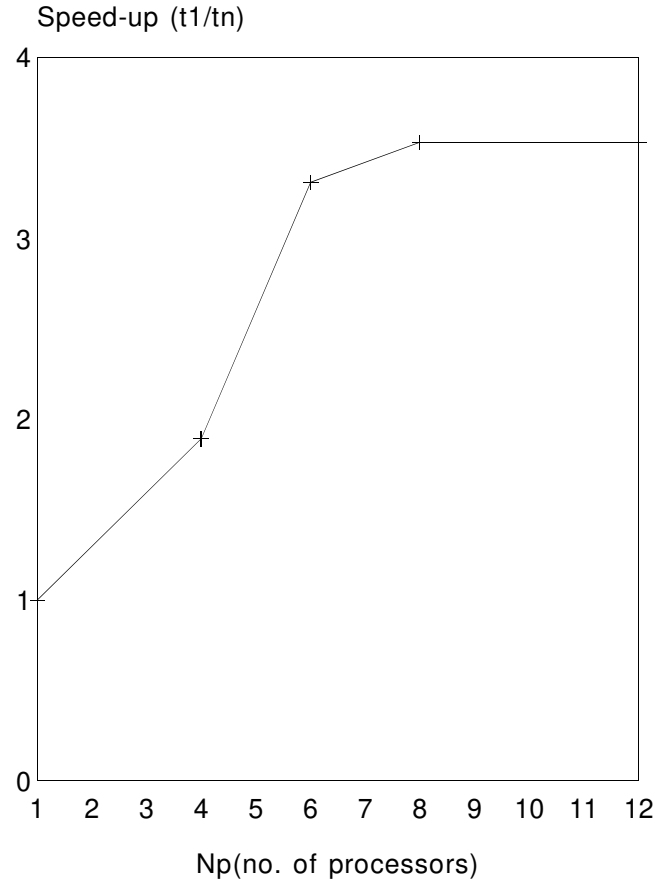

Figure 24: The mesh generator response curve for the second remeshing with 53 packets and 404 generated elements as shown in Fig. 23

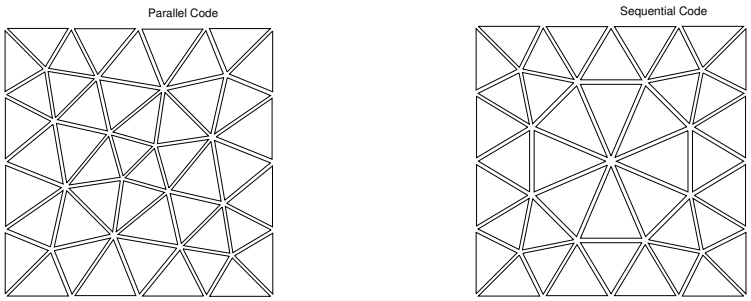

Figure 25: Mesh produced by the parallel code comprising 32 nodes, 46 elements and mesh produced by the sequential code comprising 29 nodes and 40 elements

Example 4

For comparison between the parallel and the sequential code the square domain of example 1 was selected and $\delta_n = 25.0$ was fixed for both the background mesh elements, with the over-all dimensions of the domain being 100 by 100 units. The run times for the sequential code, parallel code running on a single transputer and parallel code running on 4 transputers, were noted as 9, 6 and 3 seconds respectively. Although the background mesh comprised only 2 elements, even then the sequential code based upon the advancing front algorithm [2], was 33.33% slower than the parallel code running on a single transputer. The run time comparisons are represented graphically in Fig. 26

Example 5

The domain of example 2 was assigned a background mesh of 14 elements, as shown in Fig. 15, for parallel generation. For the sequential generation the number of the background mesh elements was reduced to only two. The length of the domain was 350 units and the height 50 units. The value of δ_i was fixed as 12.5 in both parallel and sequential cases. The run times for the sequential code executing on the root transputer, parallel code executing on root transputer and parallel code executing on 4 transputers were 277, 48 and 16 seconds respectively. The run time comparisons are represented graphically in Fig. 28

16 Remarks and Conclusions

1. From the response curves of the mesh generator, it is apparent that the performance of the parallel mesh generator is dependent on the number of packets and the computational load carried within each packet. It is also noted from the response curves that for the given range of the generated elements (i.e. under 500 elements), maximum utilization of the transputer resources occurred when the number of transputers was within 4 to 6. The steepest slopes of the response curves were located within these regions.
2. The parallel code (based upon mesh generation within the specified sub-domains as compared to the mesh generation within the whole domain as in the original advancing front algorithm) carries computational advantage over the original algorithm, even when executed in a *de-facto* sequential environment: i.e. when run on a single transputer. The computational advantage results from two aspects in this *de-facto* sequential mode.
   (a) The need for the search over the background mesh to find the local δ_n values is eliminated as the δ_n values for a triangular sub-domain are already known.
   (b) The number of segments of the advancing front for a sub-domain will always be less than the number of segments if the whole domain was considered. The segments of the fronts are scanned each time a node is considered for the formation of the triangle apex. Thus, in the parallel algorithm the number of the segments to be searched for the generation of a triangular element will always be less than the number of the segments to be searched if the whole domain was used to form the advancing front.

   It is therefore anticipated that the parallel algorithm when adopted even in a sequential mode will still hold computational advantage over the conventional advancing front algorithm.
3. As it is not possible to generate elements with sizes larger than the sub-domain, the solution mesh cannot have elements fewer in number than the starting mesh. This can only become computationally uneconomical if a relatively fine mesh was selected as the initial mesh. However, in practice this will seldom be the case. The whole concept of adaptive remeshing derives strength from the fact that from a relatively coarser mesh a fine efficient mesh is obtained with the overall domain error η reduced and evenly distributed.

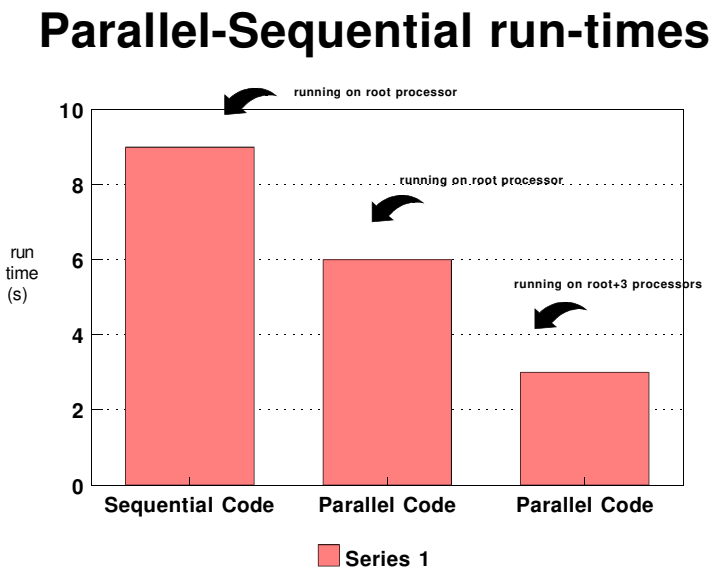

Figure 26: Run time comparison for Example 4

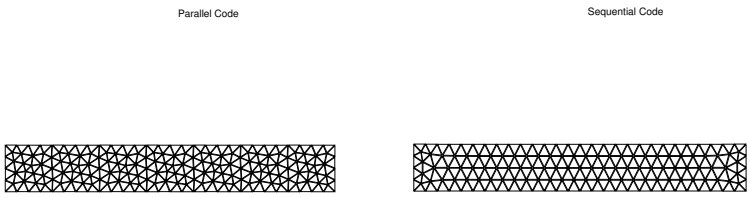

Figure 27: Mesh produced by the parallel code comprising 194 nodes, 322 elements and mesh produced by the sequential code comprising 149 nodes and 232 elements

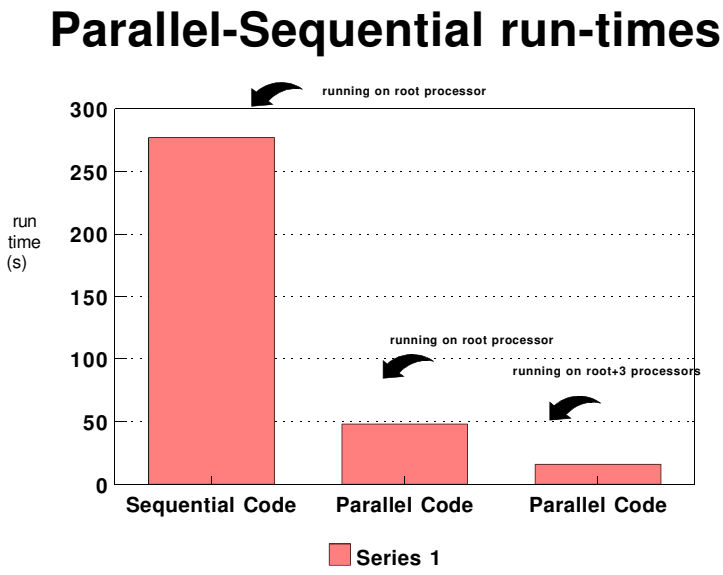

Figure 28:

4. The independent generation of elements within each sub-domain without requiring exchange of information from other sub-domains facilitated the adoption of a flood-fill environment for the parallel code. This gives portability to the parallel code and the once compiled source code file may be run on any number of transputers without recompilation.
5. As the parallel mesh generator has to traverse smaller remeshing regions as compared with the sequential mesh generator, there is relatively more squeezing of elements at the boundary fronts in the parallel algorithm. This phenomenon is illustrated in examples 4 and 5, where the number of elements generated by the parallel code is greater than the sequential code. This however does not imply that meshes obtained by one method are superior to the other if the consistency of the meshes is the criterion. All that may be concluded in this regard is that the parallel and the sequential algorithms give respectively upper-bound and lower-bound solutions, to the problem of accommodating elements of a given size/sizes within a specified remeshing region.
6. It is anticipated that the parallel mesh generators such as the one described above, coupled with parallel finite element and adaptivity modules would bring super computing performances in finite element methods to the every day desk top user. It is also believed that present day finite element procedures which are computationally time intensive and require the attention of an expert user, will find an even greater use with the degree of automation and speed which may be achieved with the full implementation of parallel adaptive tools with the finite element method.

Acknowledgements

The authors wish to thank Dr. J. Peraire for providing copies of his latest publications and reports on adaptive mesh generation.

The work is part of a larger project concerned with the simulation of the effects of gas explosions in buildings. This project is currently being funded by Building Research Establishment Research Contract F3/2/430. The authors are especially grateful to Dr. Brian Ellis who initiated this research contract, for his support and encouragement.

The authors acknowledge the assistance of 3L Ltd in providing a prototype function `net_broadcast` and advice on the operation of their Parallel C compiler.

References

[1] O. C. Zienkiewicz, J. Z. Zhu, *"A simple error estimator and adaptive procedure for practical engineering analysis"*, Int J for Nunmerical Methods in Engineering, vol. 24, 337-357, 1987.

[2] J. Peraire, M. Vahdati, K. Morgan, O. C. Zeinkiewicz, *"Adaptive remeshing for compressible flow computations"*, Journal of Computational Physics, vol. 72, 449-466, 1987.

[3] W. Atamaz-Sibia, E. Hinton, *"Adaptive mesh refinement with the Morley plate element"*, NUMETA 90, Proceedings of the Third International Conference on Numerical Methods in Engineering and Applications, University College of Swansea,

Wales, 7-11 January, 1990, Edited by G. Pande and J. Middleton, vol. 2 pp. 1044-1055, Elsevier Applied Science, London, 1990.

[4] J. Peraire, K. Morgan, J. Peiro, *"Unstructured mesh methods for CFD"*, Department of Aeronautics, Imperial College, London, IC Aero Report 90-04, June, 1990.

[5] J. Bonet, J. Peraire, *"An alternating digital tree (ADT) algorithm for 3D geometric searching and intersection problem"*, Int. J for Numerical Methods in Engineering, vol. 31, 1-17, 1991.

[6] J. S. R. Alves Filho, D. R. J. Owen, *"Using transputers in Finite Element Calculations : a first approach"*, SERC Loan Report No: TR1/034, Science and Engineering Research Council, 1989.

[7] J. Favenesi, A. Daniel, J. Tomnbello and J. Watson, *"Distributed finite element analysis using a transputer network",* Computing Systems in Engineering, vol. 1, Nos 2-4, 171-182, 1990.

[8] E. Moncrieff, B. H. V. Topping , *"The optimization of stressed membrane surface structures using parallel computational techniques",* Engineering Optimization, to appear May 1991.

[9] S.H. Lo, *"A new mesh generation scheme for arbitrary planar domains",* Int. J. for Numererical Methods in Engineering, vol.21, 1403-1426, 1985.

[10] H. Jin and N. E. Wiberg, *"Two-dimensional mesh generation, adaptive remeshing and refinement",* Int. J. for Nunerical Methods in Engineering, vol. 29, 1501-1526, 1990.

[11] TMB08, *Installation and User Manual*, Transtech Devices Ltd, Wye Industrial Estate, London Road, High Wycombe, Buckinghamshire.

[12] TMB12, *Installation and User Manual*, Transtech Devices Ltd, Wye Industrial Estate, London Road, High Wycombe, Buckinghamshire.

[13] INMOS Limited, *"Transputer Reference Manual"*, Prentice-Hall, Hertfordshire, UK, 1988.

[14] K.C. Bowler, R.D. Kenway, G.S. Pawley, D. Roweth, *An Introduction to OCCAM 2 Programming",* Chartwell-Bratt Ltd, Studentlitteratur, Lund, 1991.

[15] *"Tbug User Guide",* 3L Ltd, Livingston, Scotland, 1989.

[16] *"Parallel C user Guide (compiler version 2.1)", Reference Manual,* 3L Ltd, Livingston, Scotland, 1989.

[17] A. S. Tanenbaum, *"Operating Systems: Design and Implementation",* Prentice-Hall, Englewood Cliffs, NJ, USA, 1987.