Parallel Electronic Prototyping of Physical Objects

Poting Wu

Elias N. Houstis
Purdue University, enh@cs.purdue.edu

Report Number:
93-026
PARALLEL ELECTRONIC PROTOTYPING
OF PHYSICAL OBJECTS

Poting Wu
E. N. Houstis

CSD-TR-93-026
April 1993
Parallel Electronic Prototyping of Physical Objects

Poting Wu and Elias N. Houstis

Department of Computer Sciences
Purdue University
West Lafayette, Indiana 47907, U.S.A.
e-mail: wu@cs.purdue.edu

Technical Report CSD-TR-93-026
CAPO Report CAPO-93-19
April, 1993

Abstract

The electronic prototyping of a physical object starts with the user completely specifying the problem on an assumed initial geometry, followed by the simulation of the physics and the satisfiability of some a priori defined design objectives. The process might be repeated several times until the optimal design is obtained. This paper addresses the various issues involved in the parallel implementation of the above design process. The methodology adopted is applied on the continuous and discrete geometric data associated with the physical object and the simulation of its physics respectively. In this paper we present the formulation of the parallel electronic prototyping process for some class of structural engineering problems and the parallel algorithms developed and implemented on the nCUBE II machine for the realization of adaptive mesh generation, mesh splitting and shape optimization together with their measured performance.

This work was supported by NSF grants 9123502-CDA and 9202536-CCR, AFOSR F49620-92-J-0069 and PRF grant 6902003.
# CONTENTS

## Chapter 1  Introduction

1.1 Geometry-based modeling subsystems  
1.2 Parallel mesh and mesh-splitting preprocessor  
1.3 Analysis and shape optimization  

## Chapter 2  Parallel Adaptive Mesh Generation and Decomposition  

2.1 Methodology of the parallel mesh generation and mesh splitting  
2.2 Formulation and implementation of initial refinable background grid  
2.2.1 Mesh generation scheme  
2.2.1.1 Decompose domain into quadtree data structure  
2.2.1.2 Element meshes generation  
2.2.1.3 Element meshes adjustment  
2.2.1.4 Adjacency lists  
2.2.2 Special techniques in mesh generation  
2.2.2.1 Neighbor finding  
2.2.2.2 Polygon locating recognition  
2.3 Initial domain splitting  
2.3.1 Group I: Neighborhood - Searching scheme  
2.3.1.1 Algorithm 1: Depth - First Search  
2.3.1.2 Algorithm 2: Breadth - First Search  
2.3.2 Group II: Domain - Axis scheme  
2.3.2.1 Algorithm 3: Cartesian Axis Splitting  
2.3.2.2 Algorithm 4: Polar Axis Splitting  
2.3.2.3 Algorithm 5: Inertia Axis Splitting  
2.3.3 Performance in domain decomposition algorithm  
2.4 Subdomain boundary linking  
2.5 Final mesh generating  
2.6 Final domain splitting  
2.7 Fully automatic mesh adaptation  
2.8 Performance in parallel mesh generation  
2.8.1 Performance 1 - Engine rod head  
2.8.2 Performance 2 - Torque arm
1 Introduction

This report presents the formulation and implementation of preprocessing and postprocessing geometry based tools for achieving the parallelization of the overall design and analysis of physical objects. These tools are part of a generic software we are building for the electronic prototyping of geometry based physical objects [Hous 92]. Figure 1.1 shows a conceptual view of the processes involved in the electronic prototyping system. The prototyping of a physical object starts with the user completely specifying the problem on an assumed initial geometry. Next a fully automatic parallel mesh and mesh-splitting preprocessor creates the parallel discrete data structures on each processor for the generation of the local discrete analysis and optimization equations. For the implementation of the shape optimization phase the mesh is fully flexible (parametrized) so that it can be adapted based on the processing results. The shape adaptation is implemented on the mesh-splitting already defined on the local (subdomain) and global (interface) data.

Figure 1.1: The analysis and design process of a physical object.
1.1 Geometry-based modeling subsystems

There are many types of geometry model representations: wireframe, surface, or solid modeling. In our system we have integrated two solid modeling systems: XXoX and PATRAN. Following we give a brief overview of the two geometry modeling systems.

XXoX: This is a solid modeling system which is based on the XoX geometry and graphics libraries [XoXE 92], [XoXR 92] with an X-window interactive user interface supporting CSG type primitive operations [Wu 93]. In the XXoX environment one can create 3-D primitives and 2-D outlines of cross-sections, manipulate the geometry by orienting, combining, cutting, and deforming the objects. XXoX provides extremely powerful multiple user-interfaces, and the UNDO/REDO functions by using the higher level programming interface of the Motif widget in one integrated program. For example, the description of an engine part in XXoX language is as follows:

\[
\begin{align*}
\text{cylinder1} &= \text{cylinder}(0,0,-0.52,1.56,1.04) \\
\text{cylinder2} &= \text{cylinder}(0,0,-0.52,1.04,1.04) \\
\text{cylinder3} &= \text{cylinder}(0,0,0.5,5.4) \\
\text{box1} &= \text{box}(-0.26,-0.52,1.04,0.52,1.04,6.6) \\
\text{rod} &= \text{rotate(cylinder1-cylinder2,0,0,0,1,0,-90)} \\
\text{rod} &= \text{rod | box1 | translate(scale(rod,1,0.5,0.5),0,0,8.16)} \\
\text{rod} &= \text{rod | translate(rotate(cylinder3,0,0,0,1,0,-90),2.6,0,8.16)}
\end{align*}
\]

Figure 1.2: The description of an engine part in XXoX language.

PATRAN: This is an open-ended, general purpose, 3-D MCAE (Mechanical Computer Aided Engineering) software package that uses interactive graphics to link engineering design, analysis and results evaluation functions. By its solid geometry editor, we can create virtually any geometry or modify existing geometry imported from other design system. This environment is described in [Patran 92].
1.2 Parallel mesh and mesh-splitting preprocessor

The mapping of computations to parallel machines can be realized at the various data structures associated with the computations. In the case of PDE (Partial Differential Equations) based applications we have selected to formulate this problem at the discrete data structures of the underlying computation [Chri 91]. In this paper we formulate and implement parallel mapping algorithms on distributed memory machines including the nCUBE II and Intel iPSC/860. The uniqueness of our parallel mapping scheme is the fact that it integrates the mesh splitting (decomposition) with the mesh generation. Thus the mesh generation and mesh-splitting preprocessors are integrated into one that runs in parallel on the targeted machine. Several algorithmic alternatives are investigated for implementing the various parts of this preprocessor including suitable algorithms for mesh generation [Lohn 92] and mesh decomposition [Chri 91], [Lori 88]. Local and global mesh refinements are also supported with mesh smoothing and side swapping. Optimal domain partitioning algorithms of the mesh data are considered [Sava 91]. Figure 1.3 shows the methodology on which the parallel mesh and mesh decomposition preprocessor is based, the example assumes a 2-D region and on a 4-processor machine.

1.3 Analysis and shape optimization

The analysis and shape optimization is implemented using the domain decomposition methodology which is based on the static parallel mesh and mesh-splitting described above. The domain decomposition solvers of the parallel ELLPACK system [Hous 92] are currently used for the analysis. For the shape optimization problem we are developing two phase semi-optimal algorithms based on the local and global mesh and decomposition data [Ding 86].
1. A refinable background grid algorithm is selected to form the initial grid.

2. A scheme to split the initial grid into equal-sized subdomains is applied.

3. A linking routine to form the new subdomain boundaries is called.

4. The mesh algorithm in step 1 is applied to generate a finer mesh in parallel.

5. An optimal mesh splitting scheme to minimize the bisection width is applied.

Figure 1.3: The methodology of the parallel mesh and mesh-decomposition preprocessor.
2 Parallel Adaptive Mesh Generation and Decomposition

In general, the requirement to generate finite element meshes has been an obstacle of using the finite-element method. However, there are many methods available today to assist in the generation of finite-element meshes. This is not to say that the generation of the element meshes is no longer a major bottleneck, but the situation today is better than it was. The need to generate element meshes fast is common to a number of computational fields especially in adaptive finite element processor. Therefore, the mapping of element meshes generation to parallel machines becomes urgent. In this paper we formulate and implement parallel mapping algorithms on distributed memory machines including the nCUBE II and Intel iPSC/860. The uniqueness of our parallel mapping scheme is the fact that it integrates the mesh splitting (decomposition) with the mesh generation. Thus the mesh generation and mesh-splitting preprocessors are integrated into one that runs in parallel on the targeted machine. Several algorithmic alternatives are investigated for implementing the various parts of this preprocessor including suitable algorithms for mesh generation [Lohn 92] and mesh decomposition [Chri 91], [Lori 88]. Local and global mesh refinements are also supported with mesh smoothing and side swapping. Optimal domain partitioning algorithms of the mesh data are considered [Sava 91].

2.1 Methodology of the parallel mesh generation and mesh splitting

The parallel mapping scheme in this article contains five major steps:

1. Form an initial refinable background grid:
   Algorithm is selected to form the initial grid. Because a fairly fine initial background grid can be assumed which allows division of the background grid into subdomains of nearly equal size with maximum difference of one. In the further step, we can use the same algorithm and same code to generate mesh on subdomains in a parallel manner. These include the mesh generator, mesh refiner, mesh smoother, and mesh side swapper.

2. Split the initial grid into equal-sized subdomains:
   Several decomposition schemes are supported so that for different shapes of geometry objects we have the opportunity to test which algorithm is most optimal. In addition, there is a "local optimal" scheme has been developed. This scheme makes decision on local data to split domain into two subdomains with minimum inter-node communication during each step of the domain decomposition.

3. Link the subdomains to form new boundaries:
   Before parallel mesh generation, we need to form the new boundary of the subdomains which we got from the domain decomposition phase. Since multi-region and new holes may be created, extra effort will need in mesh generation and polygon locating recognition algorithm.

4. Generate finer element mesh in parallel:
   Since the introduction of the quadtree node distribution data structure, we can get the refined node distribution before generating mesh in parallel. Therefore, it will reduce the communication between the processor nodes to minimum. Furthermore, the generated mesh will contain more global smoothness than other approach.
5. Minimize the bisection width between each subdomain:
   In practical, different sized sets of mesh with large number of edges between these
   subdomains may happen even we generated perfectly initial domain partition. Since the
   goal of the optimal partition is NP-complete, in this step our algorithm will try to form an
   approximately optimal graph partition instead.

2.2 Formulation and implementation of initial refinable background grid

The methods for the element meshes generation of unstructured grids can be classified into two
families:
1. Advancing front algorithms.
2. Quadtree / Octree algorithms.

Several automatic mesh adaptation techniques on the first family of methods are found in
previous literatures. The scheme described in [Khan 91] can make the adaptation by predefined
the node distribution on boundary only. Adaptation in [Lo 91] introduced the boundary and
internal contours to decide the node distribution when generating element meshes. Article of
[Bykat 76] made adaptation possible by subdivision of a general polygon into convex subregions.
All methods based on the advancing front algorithms can make mesh adaptation only on specified
or computed boundary. And generate other internal element meshes by non-adaptive scheme or
interpolating computation.

The second family of methods are based on modifying an existing grid. The adaptation technique
in [Cheng 89] is defined on the user specified level assignment and vertex assignment.

For the automation and generality purpose, it seems appropriate to pursue the use of the quadtree/
octree algorithms for the following reasons:
1. Automation: Unlike the advancing front algorithms which users need to specified the node
distributing information on objects boundary, the quadtree can automatically divides the
domain into a tree structure that depends on the objects geometry. Its critical state is to
maintain all the subregions be simple - each contains only one polygon vertex or one
polygon segment.

2. Smoothness: Because the quadtree maintains the adjacency density to be 1/2 ratio -
difference of tree level between neighbors [Samet 82, 85, 89] is always no larger than one,
it manages the adaptive node distribution not only on the outer boundary of objects but
also the internal region of objects. Therefore, it provides a global smooth node distribution
when generating element meshes.

3. Adaptation: It is normal to refine whole domain globally or subregion locally. That is, it
supports a totally controlled tree structure that decides the node distribution. Therefore,
adaptive finite element processor is easy and user specified refine region is possible.

4. Parallelism: Because the refining property, it has the information of the global node
distribution before generating element meshes in parallel. This character can reduce the
communication between processor nodes to minimum even zero.
2.2.1 Mesh generation scheme

2.2.1.1 Decompose domain into quadtree data structure

The introduction of quadtree in [Samet 84] defines the node distribution on its related hierarchical data structure. The following steps will call when 1. Create the initial background quadtree, and 2. Maintain a local or global refinement.

Step1: Automatically divide the domain into a tree structure that depends on the objects geometry. Example in figure 2.1 shows that each subregion of quadtree should contain only one polygon vertex or one polygon segment after constructing the quadtree structure. To achieve this simple criterion, it needs two basic geometric techniques of the vertices finding and line segment locating.

![Figure 2.1: Construct quadtree structure on polygon object.](image1)

During constructing the quadtree structure, it also needs the tree level control to prevent the infinite refinement which occurred on few abnormal shape as shown in figure 2.2.

![Figure 2.2: Infinite refinement occurred in abnormal polygon.](image2)
Step 2: Maintain the adjacency density to be 1/2 ratio as shown in the first quadrant of figure 2.3. The neighbor finding techniques include face adjacency elements finding and corner adjacency elements finding which will be described in sec. 2.2.2.1.

![Figure 2.3: Maintain the adjacency density to be 1/2 ratio.](image-url)

Step 3: Merge the alias nodes which have the same position but belong to different nodes in the quadtree structure. It happens when vertex exactly locating on the quadtree division boundary like the case happens in figure 2.4. The reasons to merge them are to group the element meshes in next phase easily and let the mesh smoothing and side swapping work correctly.

![Figure 2.4: Merge the alias nodes.](image-url)
2.2.1.2 Element meshes generation

In this stage, triangular element meshes will be generated by connecting the precomputed nodes in previous phase. The generating steps are as follows:

Step 1: Neighbor node finding and connecting - It includes three types of neighbor connecting:

1. $N_1-N_2$ or $E_1-E_2$ connections: When the northern neighbor or the eastern neighbor exist and is not the leaf node of quadtree.

   ![Figure 2.5: Generate mesh by $N_1-N_2$ or $E_1-E_2$ connections.](image)

2. E-N connection: When northern neighbor of the eastern neighbor is equal to the northern neighbor, or eastern neighbor of the northern neighbor is equal to the eastern neighbor.

   ![Figure 2.6: Generate mesh by E-N connection.](image)

3. E-NE-N connection: Happened when both cases in 2 are failed.

   ![Figure 2.7: Generate mesh by E-NE-N connection.](image)
Step 2: Divide quadrangle into two triangles - In E-NE-N connection, it selects the diagonal by following three criteria:

1. Common edge: Choose the diagonal which does not share a common edge to prevent an invalid zero area triangle created.

![Figure 2.8: Choose the diagonal not share a common edge.](image)

2. Overlapped triangular elements: Avoid to select the diagonal which will cause two created triangular elements be overlapped. This can be done by checking the crossing point of these two diagonal.

![Figure 2.9: Avoid to create overlapped triangular elements.](image)

3. Shorter diagonal and Nearer area: Select the shorter diagonal which creates two triangular elements with the nearer area. (apply 2 factors on these two criteria. [Sadek 80])

\[
Factor = C_d \cdot \frac{AB}{CD} + C_a \cdot \frac{(\Delta ACD)/(\Delta BDC)}{(\Delta CBA)/(\Delta DAB)} = C_d \cdot \frac{AB}{CD} + C_a \cdot \frac{AO/BO}{CO/DO}
\]

![Figure 2.10: Select shorter diagonal with nearer area.](image)
Step3: Triangle dividing - It needs to divide a triangular element if:

1. Vertex inside the triangular area: It needs to divide the original triangular element into three new triangular elements as following:

![Figure 2.11: Vertex inside the triangular area.](image)

2. Vertex on the triangular edge: It needs to divide the original triangular element into two new triangular elements recursively as following:

![Figure 2.12: Vertex on the triangular edge.](image)

Step4: Triangle validating - A valid triangular element should be:

1. All three vertices are not outside polygon.
2. Triangular area is larger than 0.
3. Centre of mass is inside polygon.

Step5: Triangle adjusting - We can apply on-line mesh smoothing and side swapping in this step locally. And after all meshes generated, apply them off-line globally again. These two techniques will discuss in next section.

In step 3 and 4, the polygon locating recognition technique is needed. It recognizes a vertex where it located. That is, a vertex is inside, outside, on the edge, or on the vertices of the object polygon. It also needs to identify holes in the polygon. The detail of this technique will be described in sec. 2.2.2.2.
2.2.1.3 Element meshes adjustment

Practical implementations of element meshes generation indicate that in certain region of the mesh abrupt in element size of shape may be present. The usual way to solve this problem is to adjust the element and node distribution. This adjustment includes mesh smoothing and side swapping.

1. Mesh smoothing - Here we introduce two adjust schemes to improve the uniformity of the meshes. Since we have to know the adjacency nodes of current node to perform the smoothing, maintain a \textit{Node-node adjacency list} is necessary when generating mesh:

   a. Centre of mass - In each node smoothing, the standard Laplacian smoother is employed. Each edge of triangular element is assumed to represent a spring. Therefore, we have the active force on the current node by:

   \[ F = K \cdot \sum_{i=1}^{n} (X_i - X) \]

   where \( K \) denotes the spring factor. At the smoothing adjustment, we set \( F = 0 \) to get the centre of mass as follows:

   \[ N = \frac{\sum_{i=1}^{n} Ni}{n} \]

   Figure 2.13: Mesh smoothing by the centre of mass.

   b. Equilateral triangle - Instead of using adjacency nodes, we compute the equilateral triangles from these nodes. And apply the same Laplacian smoothing scheme on the current node by using the computed triangle nodes.

   \[ N = \frac{\sum_{i=1}^{n} Ei}{n} \]

   Figure 2.14: Mesh smoothing by the equilateral triangle.
2. **Side swapping** - In this adjustment, each quadrilateral area which contains two triangular elements has been checked. We divide the current quadrilateral area into two new triangular elements if they do not disobey the following two restrictions we discussed in step 2 of sec. 2.2.1.2:

a. Diagonal does not share a common edge.
b. Diagonal does not create two overlapped triangular elements.

and these two new triangular elements will form a better diagonal factor for shorter diagonal and nearer area we discussed:

\[
Factor = C_d \frac{N_{i+2}N_i}{N_{i+1}N} + C_a \cdot \frac{N_iO_i/N_{i+2}O_i}{N_{i+1}O_i/NO_i}
\]

![Figure 2.15: Diagonal factor for side swapping.](image)

- 13 -
2.2.1.4 Adjacency lists

In this implementation, four types of adjacency list are introduced: [Delj 90]

1. **Node-node adjacency**: It is a list of all the nodes adjacent to current node. We need it when mesh smoothing.

2. **Node-element adjacency**: It is constructed when node-node adjacency list is created. We use it when applied the side swapping.

   In practical, we can combine these two list in one data structure as follows:

   ![Figure 2.16: Structure for node-node & node-element adjacency list.](image)

3. **Element-node adjacency**: It contains three pointers to the three nodes for each triangular element. It is the basic information for a mesh element.

4. **Element-element adjacency**: It is constructed when node-element and element-node adjacency list are created. It is useful when calling neighborhood searching in decomposition process.

   We can also connect these two adjacency list in one data structure as follows:

   ![Figure 2.17: Structure for element-node & element-element adjacency list.](image)

Since all adjacency lists are constructed in the process of mesh generation, no expensive searching process is needed. We use the information of these lists during mesh splitting and linking processes.
2.2.2 Special techniques in mesh generation

2.2.2.1 Neighbor finding

Neighbor finding [Samet 82, 85, 89] includes two types of schemes in 2-D domain: one is the Face adjacency neighbor which sharing an edge with the original node, another one is the Corner adjacency neighbor which sharing a vertex with the original node. Since the well defined quadtree data structure in our implementation, we can easily find these two types of neighbors by the following algorithms:

1. **Face adjacency neighbor**: A quadtree node has face neighbors in four possible directions. They are W, S, E, N neighbors along a common edge. The algorithm to search face neighbor of equal or greater size in the horizontal or vertical direction is:

   **Step1**: Start at an original node corresponding to a specific leaf in the quadtree structure. The searching direction is $d$.

   **Step2**: Ascends the quadtree until locating the first common ancestor of the original node and its neighbor. That is the first ascending process which is not reached from a child on the node's $d$ side.

   **Step3**: Traverse downward the quadtree to find the desired neighbor by referring in a mirror image of the path from the original node to the ancestor, reflected about the common boundary.

2. **Corner adjacency neighbor**: A quadtree node also has corner neighbors in four possible directions. They are SW, SE, NE, NW neighbors along a common vertex. The algorithm to search corner neighbor of equal or greater size in the diagonal direction is:

   **Step1**: Start at an original node corresponding to a specific leaf in the quadtree structure. The searching direction is $d$.

   **Step2**: Locate the original node's nearest ancestor who is also adjacency (horizontally or vertically) to an ancestor of the sought neighbor. That is the first ancestor which is not reached from a child on the node’s $d$ corner.

   **Step3**: Make use of "face adjacency neighbor" scheme to access the ancestor of the sought neighbor on the side which direction $d$ and the child’s position shared.

   **Step4**: Retrace downward the quadtree in a mirror image of the path from the original node to the ancestor, reflected by opposite direction.
Example:

![Quadtree structure in example of neighbor finding.](image)

- West(L5): L5 -> N1 <= L6
- South(L5): L5 -> N1 <= L4
- East(L5): L5 -> N1 -> N2 -> N7 <= N4 <= N3 <= L14
- North(L5): L5 -> N1 -> N2 -> N7 <= N6 <= L20
- West(L11): L11 -> N3 -> N4 -> N7 <= N2 <= N1 <= L4
- South(L11): L11 -> N3 -> N4 <= L8
- West(L22): L22 -> N6 -> N7 <= nil

SW(L5): L5 -> N1 <= L3
SE(L5): L5 -> N1 == N3 <= L11
NE(L5): L5 -> N1 -> N2 -> N7 <= N5 <= L15
NW(L5): L5 -> N1 == L20 <= nil
SW(L11): L11 -> N3 -> N4 == N2 <= L2
SE(L11): L11 -> N3 == L8 <= nil
SW(L22): L22 -> N6 == nil

where ->: upward, <=: downward, ==: face_adjacency.
2.2.2.2 Polygon locating recognition

To determinate a vertex \( Q \) is either inside, outside, on the edge, or on the vertices of a polygon \( P = \{P_1, P_2, \ldots, P_n\} \) can be done by the following algorithm:

\[
\text{Locate\_Poly}(Q, P):
\]
\[
\Theta_{\text{total}} = 0;
\]
\[
\text{For each vertex } P_i \text{ of the polygon Do } \{
\]
\[
V_i = P_i \cdot Q;
\]
\[
V_{i+1} = P_{i+1} \cdot Q;
\]
\[
sinv = (V_i \times V_{i+1}) \cdot N;
\]
\[
cosv = V_i \cdot V_{i+1};
\]
\[
\text{If (sinv == 0) } \{
\]
\[
\text{If (cosv == 0) Return "On the vertices";}
\]
\[
\text{Elseif (cosv < 0) Return "On the edge";}
\]
\[
\}
\]
\[
\Theta_{\text{total}} += \text{atan2(sinv, cosv)};
\]
\[
\text{If (|\Theta_{\text{total}}| == 2\pi)} \}
\]
\[
\text{Return "Inside";}
\]
\[
\text{Else}
\]
\[
\text{Return "Outside";}
\]

Figure 2.19: Algorithm to locate vertex of a simple polygon.

For those polygons with holes \( H = \{H_1, H_2, \ldots, H_n\} \) in their region, we need to make more checking on the determination. Following algorithm will implement this work:

\[
\text{Locate\_Poly\_with\_Holes}(Q, P, H):
\]
\[
\text{If (Locate\_Poly}(Q, P) \neq \text{ "Inside")}
\]
\[
\text{Return Locate\_Poly}(Q, P);
\]
\[
\text{Else } \{
\]
\[
\text{For each hole } H_i \text{ in the polygon Do } \{
\]
\[
\text{If (Locate\_Poly}(Q, H_i) \neq \text{ "Outside") } \{
\]
\[
\text{If (Locate\_Poly}(Q, H_i) \text{ == "Inside")}
\]
\[
\text{Return "Outside";}
\]
\[
\text{Else}
\]
\[
\text{Return Locate\_Poly}(Q, H_i);
\]
\[
\}
\]
\[
\}
\]
\[
\text{Return "Inside";}
\]

Figure 2.20: Algorithm to locate vertex of a polygon with holes.
2.3 Initial domain splitting

In this phase, we choose several algorithms that split the initial grid into equal-sized subdomains with maximum size difference of one. Five algorithms of two scheme groups are discussed in this step:

2.3.1 Group I: Neighborhood - Searching scheme

Two algorithms to split the initial grid are based on the neighborhood traversal scheme. The starting mesh may be determined by the problem or may be chosen arbitrarily. The decomposed sets of the subdomains are grouped on the basis of searching order.

2.3.1.1 Algorithm 1: Depth - First Search

Depth - first search, which can be simply described by a recursive algorithm, is a generalization of preorder traversal of trees. When a mesh is first visited and becomes part of the depth - first tree, it recursively search its children if exist. Then the traversal scheme backs up to it and branch out in a different direction several more times.

```
Depth_First_Search(E):

Visit and mark E with partition number;
While there is an unmarked element A adjacent to E Do {
    Depth_First_Search(A);
}
```

Figure 2.21: Domain splitting by depth-first search.
2.3.1.2 Algorithm 2: Breadth - First Search

In a Breadth - first search, meshes are visited in the order of increasing distance from the starting point, where distance is simply the number of adjacency edges in a shortest path.

```
Breadth_First_Search(E):
Initialize queue Q to be empty;
Visit and mark E with partition number;
Insert E into Q;
While Q is non-empty Do {
    A = Remove_From_Queue(Q);
    For each unmarked element B adjacent to A Do {
        Visit and mark B with partition number;
        Insert B into Q;
    }
}
```

Figure 2.22: Domain splitting by breadth-first search.
2.3.2 Group II: Domain -Axis scheme

Three algorithms to decompose the initial grid are based on the domain splitting along the different defined axis. That is, define the domain axis by the basis geometry *Cartesian Axis* or *Polar Axis*, or pre-compute the *Main Symmetry Axis* according to the mesh elements in the domain region. Then split the mesh elements along the axis into subdomain.

2.3.2.1 Algorithm 3: Cartesian Axis Splitting

Splitting the domain along the cartesian axis by sorting the X, Y, Z coordinates of centre of mass of mesh elements. Several minor schemes are presented for selecting the suitable subdomains on different problems. They are:

a. **Row· Column Cartesian Axis Splitting**: Suppose we need to split the domain into nr rows by nc columns subdomains, where nr * nc = np no. of processors. This scheme first splits the domain into nr subdomains by sorting their Y coordinates. Then for each subdomain, the scheme splits it into nc subdomains again by sorting their X coordinates.

![Diagram of domain splitting by row-column cartesian axis splitting](image)

**Row·Column·Cartesian()**: 

Sort their Y coordinates;
Split the domain into nr subdomains along the Y axis;
For each splitted subdomain Do {
    Sort their X coordinates;
    Split the domain into nc subdomains along the X axis;
}

Figure 2.23: Domain splitting by row-column cartesian axis splitting.
b. **Column - Row Cartesian Axis Splitting:** The scheme is similar to a but it splits the domain into nc subdomains by sorting their X coordinates first. Then for each subdomain, the scheme splits it into nr subdomains again by sorting their Y coordinates.

![Diagram of domain splitting by column-row cartesian axis splitting]

**Column_Row_Cartesian()**:

Sort their X coordinates;
Split the domain into nc subdomains along the X axis;
For each splitted subdomain Do {
    Sort their Y coordinates;
    Split the domain into nr subdomains along the Y axis;
}

Figure 2.24: Domain splitting by column-row cartesian axis splitting.
c. **RCRC Cartesian Axis Splitting**: This scheme is similar to a but it splits each domain into 2 subdomains each time. That is, it splits the domain into 2 subdomains by sorting their Y coordinates. Then for each subdomain, the scheme splits it into 2 subdomains again by sorting their X coordinates. Repeat these 2 steps until number of subdomains is reached.

![Diagram of RCRC Cartesian Axis Splitting]

```
RCRC_Cartesian()

While nr or nc is larger than 1 Do {
    If (nr > 1) {
        For each splitted subdomains Do {
            Sort their Y coordinates;
            Split the domain into 2 subdomains along the Y axis;
        }
        nr /= 2;
    }
    If (nc > 1) {
        For each splitted subdomain Do {
            Sort their X coordinates;
            Split the domain into 2 subdomains along the X axis;
        }
        nc /= 2;
    }
}
```

Figure 2.25: Domain splitting by RCRC cartesian axis splitting.
d. **CRCR Cartesian Axis Splitting**: This scheme is similar to c but it first splits domain into 2 subdomains by sorting their X coordinates. Then for each subdomain, the scheme splits it into 2 subdomains again by sorting their Y coordinates. Repeat these 2 steps until number of subdomains is reached.

\[
\begin{array}{c}
\text{CRCR Cartesian():} \\
\text{While } nr \text{ or } nc \text{ is larger than } 1 \text{ Do } \\
\quad \text{If } (nc > 1) \{ \\
\quad \quad \text{For each splitted subdomain Do } \\
\quad \quad \quad \text{Sort their X coordinates;} \\
\quad \quad \quad \text{Split the domain into 2 subdomains along the X axis;} \\
\quad \quad \} \\
\quad \quad nc /= 2; \\
\quad \} \\
\quad \text{If } (nr > 1) \{ \\
\quad \quad \text{For each splitted subdomains Do } \\
\quad \quad \quad \text{Sort their Y coordinates;} \\
\quad \quad \quad \text{Split the domain into 2 subdomains along the Y axis;} \\
\quad \quad \} \\
\quad \quad nr /= 2; \\
\} \\
\]

Figure 2.26: Domain splitting by CRCR cartesian axis splitting.
e. **Optimal Cartesian Axis Splitting**: This scheme is similar to c and d. Each time it splits the domain into 2 subdomains by selecting either the X axis or the Y axis which causes less communication between these new generated subdomains. Repeat this step until number of subdomains is reached.

![Diagram of domain splitting by optimal cartesian axis splitting]

**Optimal Cartesian()**: 

While np larger than 1 Do {
   For each splitted subdomain Do {
      Sort their X coordinates;
      Split the domain into 2 subdomains along the X axis;
      Compute the communication bisection width BW_x;
      Sort their Y coordinates;
      Split the domain into 2 subdomains along the Y axis;
      Compute the communication bisection width BW_y;
      Select the one which has smaller bisection width;
   }
   np /= 2;
}

Figure 2.27: Domain splitting by optimal cartesian axis splitting.
### 2.3.2.2 Algorithm 4: Polar Axis Splitting

Similar to the cartesian axis splitting, it splits the domain along the polar axis by sorting the R, Θ, Z coordinates of center of mass of mesh elements. In addition to the minor adjustment in cartesian axis splitting, the various definitions of the original point are possible. Its calling process is as following:

Define the original point as either:
1. Centre of Inertia of meshes.
2. Centre of Mass of meshes.
3. User specified.

Map coordinates from Cartesian to Polar:

\[(X, Y, Z) \rightarrow (R, \Theta, Z)\]

Call relative Cartesian Axis Splitting routine:

\[X \rightarrow R, \ Y \rightarrow \Theta, \ Z \rightarrow Z\]

Map coordinates from Polar to Cartesian:

\[(R, \Theta, Z) \rightarrow (X, Y, Z)\]

Figure 2.28: Strategy for polar axis splitting.

**a. Row - Column Polar Axis Splitting:** This scheme first splits the domain into nr subdomains by sorting their Θ coordinates. Then for each subdomain, the scheme splits it into nc subdomains again by sorting their R coordinates.

Figure 2.29: Domain splitting by row-column polar axis splitting.
d. **CRCR Polar Axis Splitting:** This scheme is similar to c but it first splits domain into 2 subdomains by sorting their R coordinates. Then for each subdomain, the scheme splits it into 2 subdomains again by sorting their Θ coordinates. Repeat these 2 steps until number of subdomains is reached.

![CRCR Polar Axis Splitting](image)

Figure 2.32: Domain splitting by CRCR polar axis splitting.

c. **Optimal Polar Axis Splitting:** This scheme is similar to c and d. Each time it splits the domain into 2 subdomains by selecting either the R axis or the Θ axis which causes less communication between these new generated subdomains. Repeat this step until number of subdomains is reached.

![Optimal Polar Axis Splitting](image)

Figure 2.33: Domain splitting by optimal polar axis splitting.
b. **Column - Row Polar Axis Splitting:** The scheme is similar to a but it splits the domain into \( nc \) subdomains by sorting their \( R \) coordinates first. Then for each subdomain, the scheme splits it into \( nr \) subdomains again by sorting their \( \Theta \) coordinates.

![Diagram of Column - Row Polar Axis Splitting](image)

Figure 2.30: Domain splitting by column-row polar axis splitting.

c. **RCRC Polar Axis Splitting:** This scheme is similar to a but it splits each domain into 2 subdomains each time. That is, it splits the domain into 2 subdomains by sorting their \( \Theta \) coordinates. Then for each subdomain, the scheme splits it into 2 subdomains again by sorting their \( R \) coordinates. Repeat these 2 steps until number of subdomains is reached.

![Diagram of RCRC Polar Axis Splitting](image)

Figure 2.31: Domain splitting by RCRC polar axis splitting.
2.3.2.3 Algorithm 5: Inertia Axis Splitting

In this scheme, it first pre-computes the main symmetry axis according to the centre of mass of mesh elements. Then splits domain into several subdomains along the axis. Repeat this step until the number of subdomains is reached.

![Inertia Axis Splitting Diagram]

\[
\text{Inertia\_Axis}(J:\ # \text{: split level number})
\]

While np is larger than 1 Do {
    For each splitted subdomain Do {
        Compute the main symmetry axis \( \alpha \);
        Split the domain into ns subdomains along axis \( \alpha \);
    }
    np /= ns;
}

Figure 2.34: Domain splitting by inertia axis splitting.

\textbf{Computation of the main symmetry axis:}

Let \( A \) be the \((2 \times N)\) matrix of the mesh coordinates which belong to the current domain with the original point as either:

1. Centre of Inertia of current meshes.
2. Centre of Mass of current meshes.
3. User specified.

The main symmetry axis is given by the Eigen vector corresponding to the largest Eigen value of \( A^T A \).
2.3.3 Performance in domain decomposition algorithm

In the following table, we list the Maximum communication bisection bandwidth between processors. Where total means the maximum total edges one specified processor node joining with others, and 1-by-1 means the maximum edges between any two processor nodes. The first three examples are based on the engine rod head with different mesh density and processor number, while the fourth one is based on the torque arm.

<table>
<thead>
<tr>
<th>Equal-sized with maximum difference of one</th>
<th>Max. communication bandwidth (total / 1-by-1)</th>
<th>remark</th>
</tr>
</thead>
<tbody>
<tr>
<td>Neighborhood - Search</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Depth - First Search</td>
<td>26 / 13</td>
<td>117m, 4p</td>
</tr>
<tr>
<td>Breadth - First Search</td>
<td>21 / 13</td>
<td>2191m, 4p</td>
</tr>
<tr>
<td>Domain - Axis Split</td>
<td></td>
<td>2191m, 16p</td>
</tr>
<tr>
<td>Cartesian</td>
<td></td>
<td>3734m, 16p</td>
</tr>
<tr>
<td>Depth - First Search</td>
<td>176 / 111</td>
<td></td>
</tr>
<tr>
<td>Breadth - First Search</td>
<td>121 / 75</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Column - Row</td>
<td>9 / 5</td>
<td>63 / 26</td>
</tr>
<tr>
<td>Column - Row</td>
<td>8 / 4</td>
<td>73 / 31</td>
</tr>
<tr>
<td>R - C - R - C</td>
<td>9 / 5</td>
<td>63 / 27</td>
</tr>
<tr>
<td>C - R - C - R</td>
<td>8 / 4</td>
<td>87 / 29</td>
</tr>
<tr>
<td>Optimal</td>
<td>8 / 4</td>
<td>Loriot 1988</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Polar</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Depth - First Search</td>
<td>18 / 10</td>
<td>68 / 23</td>
</tr>
<tr>
<td>Breadth - First Search</td>
<td>16 / 10</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Cartesian</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Depth - First Search</td>
<td>64 / 36</td>
<td></td>
</tr>
<tr>
<td>Breadth - First Search</td>
<td>65 / 43</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Column - Row</td>
<td>16 / 10</td>
<td>57 / 27</td>
</tr>
<tr>
<td>Column - Row</td>
<td>18 / 10</td>
<td>152 / 74</td>
</tr>
<tr>
<td>R - C - R - C</td>
<td>18 / 10</td>
<td></td>
</tr>
<tr>
<td>R - C - R - C</td>
<td>16 / 10</td>
<td>81 / 38</td>
</tr>
<tr>
<td>C - R - C - R</td>
<td>8 / 5</td>
<td>Loriot 1988</td>
</tr>
<tr>
<td>Optimal</td>
<td>8 / 5</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Inertia Axis Split</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Depth - First Search</td>
<td>20 / 12</td>
<td>161 / 96</td>
</tr>
<tr>
<td>Breadth - First Search</td>
<td>99 / 56</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Column - Row</td>
<td>79 / 34</td>
<td></td>
</tr>
<tr>
<td>Column - Row</td>
<td>35 / 21</td>
<td></td>
</tr>
<tr>
<td>C - R - C - R</td>
<td>35 / 18</td>
<td></td>
</tr>
<tr>
<td>Optimal</td>
<td>64 / 21</td>
<td></td>
</tr>
</tbody>
</table>

Table 1: Performance of domain decomposition algorithm
2.4 Subdomain boundary linking

After mesh decomposition, we need a linking routine to connect the new boundary for each subdomain before progressing mesh generation in parallel. Since the domain splitting will create more than one subregion for one processor in some problems, the mesh generation scheme should be capable of handling such case. New holes may be created also. Therefore, the linking routine needs to separate the outer boundary polygon and the hole polygon. And identify a hole polygon belongs to which outer boundary polygon. The linking algorithm is as follows:

Step1: Find the polygon list including outer boundary polygon and hole polygon:

While there is an unmarked mesh S which is on the boundary Do {
    Mark S and insert it into boundary list L;
    E₁ = S;
    Do {
        Search an unmarked mesh E₂ which is on the boundary and adjacent to E₁;
        Mark E₂ and insert it into boundary list L;
        E₁ = E₂;
    } While (E₁ ¹ S);
}

It needs the element-element adjacency list to search the boundary meshes, and node-element adjacency list to locate the next boundary mesh.

Step2: Separate the outer boundary polygon and the hole polygon by the polygon locating recognition algorithm we described in sec. 2.3.2.

For each polygon P₁ Do {
    For each polygon P₂ except P₁ Do {
        If (Locate_Poly(P₁, P₂) = "Inside") {
            Link P₁ to the hole list of P₂;
            Break;
        }
    }
}

2.5 Final mesh generating

Reviewed those previous literatures about the parallel mesh generation: In [Lohn 92], it either generates mesh of each subdomain separately in parallel, then generates mesh of the inter-subdomain region sequentially. Or generates mesh of the inter-subdomain region sequentially, then generates mesh of each subdomain separately in parallel. In both cases, they need to communicate between processor nodes for generating mesh in the inter-subdomain region, or generating them sequentially. Furthermore, when generating mesh in each subdomain, it does not have a global smooth node distribution.

In our approach, since we introduce the quadtree data structure to supervise the node distribution, it is easy and efficient to refine it globally before generating mesh in parallel. Therefore, during the progress of parallel mesh generation, it does not need the communication between processor nodes. And because of the global smoothness of the node distribution, it will generate better mesh elements.

In addition, we can use the same algorithm and same code in sequential mesh generator to generate the unstructured mesh on each subdomain in parallel. And the same code for parallel local mesh smoothing and side swapping, sequential global mesh smoothing and side swapping. These will reduce the development effort and cost.

![Diagram of final mesh generating in parallel]

Figure 2.35: Strategy of final mesh generating in parallel.
2.6 Final domain splitting

Practical implementations of parallel mesh generators indicate that different sized sets of mesh with large number of edges joining the sets between subdomains may be present. These variations appear even when trying to generate perfectly initial domain splitting. At this phase, two ways to partition mesh elements are possible [Krish 84]: schemes we normally used as described in sec. 2.3, called constructive algorithm that construct a partition from a set of finite element meshes; and schemes that improve upon an existing partition, called refinement algorithm which we will discuss in this section.

Since the goal of the optimal graph partition is NP-complete, in this step the domain decomposition schemes are needed to approximate the minimum bisection width [Sava 91]. Two widely used strategies for this problem are the Kernighan-Lin (KL) algorithm [Krish 84], [Kern 70] and the Simulated Annealing (SA) algorithm [John 89], [Kirk 83].

Kernighan_Lin():

\[
P = \text{Initial partition}; \]
\[
Q = \text{Best\_Partition\_in\_KL}(P); \]
\[
\text{While Bisection\_Width}(Q) \text{ is smaller than Bisection\_Width}(P) \text{ Do } \{ \]
\[
P = Q; \]
\[
Q = \text{Best\_Partition\_in\_KL}(P); \]
\[
\}

The above scheme makes small local improvement only by downhill moves until no such alternation yields a better solution to reach a local optimal partition. In order to avoid a poor locally optimal partition, the simulated annealing algorithm occasionally allows the uphill moves to randomize this procedure. Therefore, it could prevent the refinement stuck in a globally poor mesh partition.
2.7 Fully automatic mesh adaptation

After generating an initial mesh and performing a finite element analysis, our automatic finite element modeling program is able to measure the local error, determine the areas of the mesh where the solution is not sufficiently accurate, and refine mesh in the specified area. That is, it can automatically perform as many iterations of analysis, error estimation, and mesh improvement as required to reach the desired degree of accuracy.

Since only displacements are guaranteed to be continuous in displacement based finite element analysis, and stresses are not assured to be continuous. The stresses on those nodes sharing by different elements will not match. Therefore, the feature of the posteriori error estimators we are currently used in P/FEA is based on the difference between the averaged and the unaveraged element stresses. [Zienk 87]

\[ E_\sigma = \sigma' - \sigma \]

where \( \sigma \) is the unaveraged element stresses which are always directly computed by the Gauss points and extrapolated to the node points to obtain the node stresses. And \( \sigma' \) is the averaged element stresses which are obtained to produce a continuous stress field. In general, the error estimator \( E_\sigma \) will be nonzero unless the finite element analysis result is exact. Based on the energy norm to compute the element stress error for the \( i \)th element as,

\[ \| E_\sigma \|^2 = \int_{\Omega_i} \{ E_\sigma \}^T [K]^{-1} \{ E_\sigma \} \, d\Omega_i \]

where \([K]\) is the material stiffness matrix. And the refinement strategy is dependent on the specified accuracy requirement of a certain minimum percentage error in the energy norm.

Figure 2.36 shows an example of the fully automatic mesh adaptation in one adaptive step by using P/FEA. In our approach, mesh refinement is done by reduction of the mesh size (h-refinement) which is normal to most engineering problems.
15 refine points in one adaptive step

79 nodes, 117 elements

136 nodes, 222 elements

Figure 2.36: Fully automatic mesh adaptation in one adaptive step by using P/FEA.
2.8 Performance in parallel mesh generation

For the two examples, engine rod head and torque arm, we have discussed the performance evaluation including Speedup and Utilization which shows the three states - busy, overhead, and idle - as a function of time for each processor. We categorize each processor as idle if it has suspended execution awaiting a message that has not yet arrived or it has ceased execution at the end of the run, overhead if it is executing the communication stuff in program, and busy if it is executing the portion of program other than the communication stuff. [Geist 92] [Geist 90] [Heath 93]

2.8.1 Performance 1 - Engine rod head

![Utilization Count](image)

**Utilization Count**

states of idle, overhead, and busy as function of time.

![Utilization Summary](image)

**Utilization Summary**

overall cumulative percentage of time in idle, overhead, and busy states.

Figure 2.37: Performance of parallel mesh generation - Engine rod head.
2.8.2 Performance 2 - Torque arm

Utilization Count
states of idle, overhead, and busy as function of time.

Utilization Summary
overall cumulative percentage of time in idle, overhead, and busy states.

Figure 2.38: Performance of parallel mesh generation - Torque arm.
3 Parallel Shape Optimization

Shape Optimization of a large complex system with a great deal of variables and constraints is usually time consuming task. It might be more efficient to divide the system into several smaller subsystems. In general, an optimization problem involving many variables and constraints cannot be decomposed into independent subproblems which can be independently optimized. Nevertheless, the methods in this article do permit the decomposition of shape optimization into subproblem which solved independently in a parallel manner yields the whole system optimum. The analysis and shape optimization is implemented using the domain decomposition methodology which is based on the static parallel mesh and mesh-splitting described in Parallel Adaptive Mesh Generation. The domain decomposition solvers of the parallel ELLPACK system [Hous 92] are currently used for the analysis. For the shape optimization problem we are developing two level semi-optimal algorithms based on the local and global mesh and decomposition data [Ding 86].

3.1 Formulation of parallel shape optimization process

The two level semi-optimal algorithm is a hierarchical strategy in which there are two levels of optimization schemes. The lower-level which solves the optimization subproblem on the local mesh of each subsystem independently in a parallel manner. And the higher-level, controls the global mesh, coordinates the action of the lower-level units so that the optimum of the original problem is obtained.

Consider the general optimization problem of choosing the variables \( \{X\} \) such that

\[
Z = F(\{X\}) \Rightarrow \min.
\]

\[
\{h(\{X\})\} = \{0\}
\]

\[
\{g(\{X\})\} \leq \{0\}
\]

\[
\{X^L\} \leq \{X\} \leq \{X^U\}
\]

where \( F(\{X\}) \) is the objective function. \( \{h(\{X\})\} \) and \( \{g(\{X\})\} \) are set of equality and inequality constraints. And \( \{X^L\} \) and \( \{X^U\} \) are the lower and upper bound vector of \( \{X\} \).

Decomposition of the optimization problem is carried out by first converting the problem into a two level form with separate and distinct tasks assigned to each level. That is, we split apart of the variables and constraints for each subdomain which do not interact with others in other subdomain to form the lower-level units. Then choose the dependent variables, called coordinating variables, to the higher-level unit which correspond to an overall system optimum. In general, the lower-level and the higher-level problems are solved iteratively.

There are two different ways to convert a given problem into two-level schemes which are the model coordination method and the goal coordination method [Kirsch 75].
3.2 Model coordination method

We partition the vector \( \{X\} \) into two subvectors, \( \{S\} \) and \( \{T\} \)

\[
\{X\}^T = (\{S\}^T, \{T\}^T)
\]

in which \( \{S\} \) is called the subvector of **coordinating variables** between the subdomains and \( \{T\} \) is the **subdomain variables** which we decompose it into subdomains as follows:

\[
\{T_1\} \\
\vdots \\
\{T\} = \{T_i\} \\
\vdots \\
\{T_n\}
\]

where \( \{T_i\} \) represents the subdomain variables associated with the \( i \)-th subdomain and \( n \) is the number of subdomains. Decomposition is effected so that the objective function and the equality and inequality constraints can be rewritten in the following form:

\[
Z = F(\{X\}) = \sum_{i=1}^{n} F_i(\{S\}, \{T_i\})
\]

\[
\{h_1(\{S\}, \{T_1\})\} \\
\vdots \\
\{h(\{S\}, \{T_i\})\} \\
\vdots \\
\{h_n(\{S\}, \{T_n\})\}
\]

\[
\{g_1(\{S\}, \{T_1\})\} \\
\vdots \\
\{g(\{S\}, \{T_i\})\} \\
\vdots \\
\{g_n(\{S\}, \{T_n\})\}
\]

That is, the coordinating variables \( \{S\} \) may appear in all expressions, while the subdomain variables \( \{T_i\} \) appear only in the term \( F_i \) of the objective function and the equality sets \( \{h_i\} = \{0\} \) and the inequality set \( \{g_i\} \leq \{0\} \). Based on this, the original problem can be restated as:

\[
Z = \sum_{i=1}^{n} F_i(\{S\}, \{T_i\}) \Rightarrow \text{min}
\]

\[
\{h_i(\{S\}, \{T_i\})\} = \{0\} \quad i = 1, \ldots, n
\]

\[
\{g_i(\{S\}, \{T_i\})\} \leq \{0\} \quad i = 1, \ldots, n
\]

\[
\{S^L\} \leq \{S\} \leq \{S^U\}
\]

\[
\{T_i^L\} \leq \{T_i\} \leq \{T_i^U\} \quad i = 1, \ldots, n
\]

again, \( \{S^L\}, \{S^U\}, \{T_i^L\}, \) and \( \{T_i^U\} \) are the lower and upper bound vectors for the decomposed subproblems.
The two level problem can be solved iteratively as following steps:
1. Choose an initial value for the coordinating variables \( \{S^0\} \).
2. For a given \( \{S^0\} \) solve the \( n \) independent lower-level subproblems in a parallel manner.
3. Modify the value of \( \{S^0\} \) so that higher-level is optimum.
4. Repeat steps (2) and (3) until the global optimum is achieved.

The formulation of these two levels is as follows:

**Lower-level problem:**
For a given fixed value of coordinating variables \( \{S^0\} \), the problem in this level can be decomposed into \( n \) independent subproblems. Each of them stated as: find \( \{T_i\} \) such that,

\[
Z_i = F_i(\{S^0\}, \{T_i\}) \Rightarrow \text{min}
\]

\[
\{h_i(\{S^0\}, \{T_i\})\} = \{0\}
\]

\[
\{g_i(\{S^0\}, \{T_i\})\} \leq \{0\}
\]

\[
\{T_i^L\} \leq \{T_i\} \leq \{T_i^U\}
\]

**Higher-level problem:**
The task in this level is to find a \( \{S^0\} \) such that,

\[
Z = \sum_{i=1}^{n} F_i(\{S^0\}, \{T_i\}) \Rightarrow \text{min}
\]

\[
\{S^L\} \leq \{S^0\} \leq \{S^U\}
\]

while all \( \{T_i\} \) are fixed.

Figure 3.1: Model coordination method for parallel shape optimization.
3.3 Goal coordination method

In this scheme, the overall system is decoupled. That is, all links between its subsystems are disconnected. And the variables \{S\} called the *interconnection variables* are now permitted to differ on either side of each subsystem interface. Assign \{S_i\} as the vector of the above variables associated with the ith subsystem, then the vector of variables in the ith subsystem becomes

\[(X_i)^T = (\{S_i\}^T, \{T_i\}^T)\]

That is, while \{T_i\} and \{T_{i+1}\} represent different variables, \{S_i\} and \{S_{i+1}\} may represent the same variables with different values. \{S_i\} and \{T_i\} are chosen so that the variables \{X_i\} appear only in the term \(F_i\) of the objective function and the equality sets \(\{h_i\} = \{0\}\) and the inequality set \(\{g_i\} \leq \{0\}\).

\[
Z = F(\{X\}) = \sum_{i=1}^{n} F_i(\{X_i\})
\]

\[
\{h_i(\{S_i\}, \{T_i\})\} = \{0\} \quad i = 1, \ldots, n
\]

\[
\{g_i(\{S_i\}, \{T_i\})\} \leq \{0\} \quad i = 1, \ldots, n
\]

\[
(S^L) \leq \{S_i\} \leq (S^U) \quad i = 1, \ldots, n
\]

\[
(T_i^L) \leq \{T_i\} \leq (T_i^U) \quad i = 1, \ldots, n
\]

Therefore, the optimum of the overall system is achieved when the *interaction-balance conditions* to be satisfied:

\[
\{S_i, i+1\} = \{S_{i+1}, i\} \quad i = 1, \ldots, n - 1
\]

where \(\{S_i\}^T = (\{S_{i,i-1}\}^T, \{S_{i,i+1}\}^T)\)

\[\text{Figure 3.2: The interaction-balance conditions for the overall system.}\]

To obtain the optimum of the overall system, we need to define a new vector of coordinating variables,

\[
\{\lambda\}^T = (\{\lambda_1, 2\}^T, \{\lambda_{2, 3}\}^T, \ldots, \{\lambda_{n-1, n}\}^T)
\]

And introduce an extra term of penalty function, vanishing at the optimum when the interaction-balance condition is satisfied, in the new objective function which is defined by,

\[
Z = F(\{X\}, \{\lambda\}) = \sum_{i=1}^{n} F_i(\{X_i\}) + \sum_{i=1}^{n-1} \{\lambda_{i,i+1}\}^T (\{S_{i,i+1}\} - \{S_{i+1,i}\})
\]
Expanding the term of penalty function into the following form:

\[
\sum_{i=1}^{n-1} \{\lambda_{i,i+1}\}^T \{S_{i,i+1} - S_{i+1,i}\} = \sum_{i=1}^{n} \lambda_i^T S_i
\]

where \(\lambda_i^T = (\lambda_{i-1,i}^T, \lambda_{i,i+1}^T)^T\)

And the objective function of the dual optimization problem becomes,

\[
Z = F(\{X\}, \{\lambda\}) = \sum_{i=1}^{n} (F_i(\{X_i\}) + \lambda_i^T S_i)
\]

In general, for a given initial value of \(\{\lambda^0\}\), it can be shown that,

\[
(Z(\lambda) = F(\{X\}, \{\lambda\})) \geq \left(D(\lambda^0) = \sum_{i=1}^{n} (F_i(\{X_i\}) + \lambda_i^0)^T S_i\right)
\]

That is, \(D(\lambda^0)\) is a lower bound of the objective function. If \(F(\{X\}, \{\lambda\})\) has a saddle point, we will obtain

\[
\max D(\lambda) = \min F(\{X\}, \{\lambda\})
\]

Therefore, the two level problem can be solved iteratively as following steps:
1. Choose an initial value for the coordinating variables \(\{\lambda^0\}\).
2. For a given \(\{\lambda^0\}\) solve the n independent lower-level subproblems in a parallel manner.
3. Modify the value of \(\{\lambda^0\}\) in the higher level so that the objective function increase.
4. Repeat steps (2) and (3) until the maximum objective function is achieved.

The formulation of these two levels is as follows:

**Lower-level problem:**
For a given fixed value of coordinating variables \(\{\lambda^0\}\), the problem in this level can be decomposed into n independent subproblems. Each of them stated as:

find \(\{X_i\}^T = (\{S_i\}^T, \{T_i\}^T)\) such that,

\[
Z_i = F_i((X_i)) + \lambda_i^0 S_i \Rightarrow \min.
\]

\[h_i((X_i)) = \{0\}\]
\[g_i((X_i)) \leq \{0\}\]
\n\(X_i^L \leq X_i \leq X_i^U\)
Higher-level problem:
The task in this level is to find a $\{\lambda^0\}$ so that the interaction-balance condition is satisfied. That is, to make $\{S_{i,i+1}\}$ and $\{S_{i+1,i}\}$ be equal. This is achieved by,

$$Z = F(\{X\}, \{\lambda\}) = \sum_{i=1}^{n} (F_i(\{X_i\}) + \{\lambda_i\}^T\{S_i\}) \Rightarrow \max$$

while all $\{T_i\}$ are fixed.

![Goal coordination method for parallel shape optimization.](image)

Figure 3.3: Goal coordination method for parallel shape optimization.
3.4 Examples of parallel shape optimization

Figure 3.4 and 3.5 show two examples of the parallel shape optimization with the concept of the model coordination method and the goal coordination method.

3.4.1 Example 1 - Parallel shape optimization with load case #1

Figure 3.4: parallel shape optimization with load case #1.
3.4.2 Example 2 - Parallel shape optimization with load case #2

Shape Optimization

\[ Z = \text{Area} \Rightarrow \text{min.} \]

\[ \sigma \leq 18000 \]

\[ -\sigma \leq 22000 \]

\[ 0.5 \leq S_i \leq 1.0 \quad i = 1, \ldots, 3 \]

\[ 0.1 \leq T_{ij} \leq 1.0 \quad i = 1, 3 \]

\[ j = 1, 2 \]

\[ 0.5 \leq T_{ij} \leq 1.0 \quad i = 2, 4 \]

\[ j = 1, 2 \]

\[ 0.5 \leq T_{ij} \leq 1.0 \quad i = 1, 3 \]

Figure 3.5: parallel shape optimization with load case #2.

156 nodes, 264 elements

156 nodes, 244 elements

Stress: <-17643.3, 13895.8>

Stress: <-21490.4, 17731.9>
4 References


[Wu 93] Poting Wu and E. N. Houstis, “XXoX: An interactive X-window based user interface for the XoX solid modeling library”, CAPO Technical Report, Purdue University, Department of Computer Sciences, TR-93-08, (January 1993) 1-47.


