SANDIA REPORT

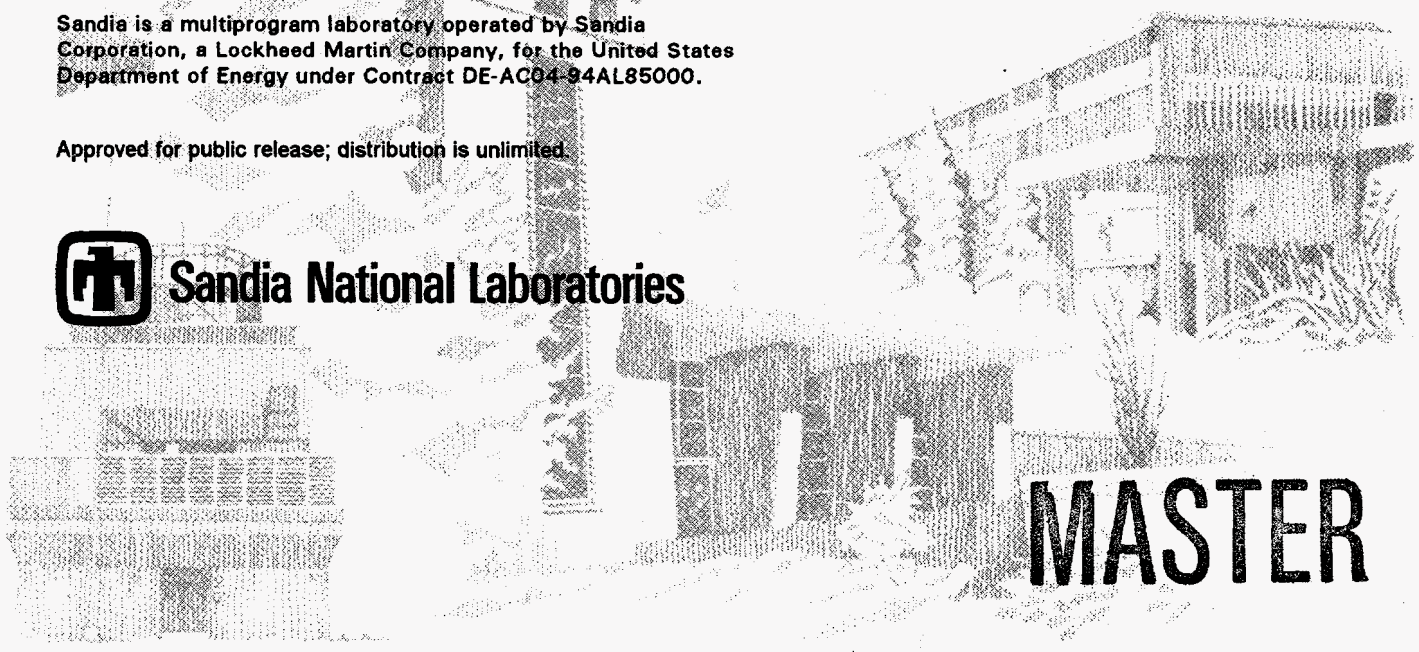
SAND97-0545 • UC-705
Unlimited Release
Printed March 1997

Parallel Paving: An Algorithm for Generating Distributed, Adaptive, All-quadrilateral Meshes on Parallel Computers

Randy R. Lober, Timothy J. Tautges, Courtenay T. Vaughan

Prepared by
Sandia National Laboratories
Albuquerque, New Mexico 87185 and Livermore, California 94550

Sandia is a multiprogram laboratory operated by Sandia Corporation, a Lockheed Martin Company, for the United States Department of Energy under Contract DE-AC04-94AL85000.



Sandia National Laboratories

Issued by Sandia National Laboratories, operated for the United States Department of Energy by Sandia Corporation.

Available to DOE and DOE contractors from
Office of Scientific and Technical Information
P.O. Box 62
Oak Ridge, TN 37831

Prices available from (615) 576-8401, FTS 626-8401

Available to the public from
National Technical Information Service
U.S. Department of Commerce
5285 Port Royal Rd
Springfield, VA 22161

NTIS price codes
Printed copy: A03
Microfiche copy: A01

SAND97-0545
Unlimited Release
Printed March 1997

Distribution
Category-705

Parallel Paving: An Algorithm for Generating Distributed, Adaptive, All-quadrilateral Meshes on Parallel Computers

Randy R. Lober
Thermal Sciences Department

Timothy J. Tautges
Parallel Computing Sciences Department

Courtenay T. Vaughan
Parallel Computing Sciences Department

Sandia National Laboratories
P.O. Box 5800
Albuquerque, New Mexico 87185-0835

Abstract

Paving [1] is an automated mesh generation algorithm which produces all-quadrilateral elements. It can additionally generate these elements in varying sizes such that the resulting mesh adapts to a function distribution, such as an error function. While powerful, conventional paving is a very serial algorithm in its operation. Parallel paving is the extension of serial paving into parallel environments to perform the same meshing functions as conventional paving only on distributed, discretized models. This extension allows large, adaptive, parallel finite element simulations to take advantage of paving's meshing capabilities for h-remap remeshing. A significantly modified version of the CUBIT [2] mesh generation code has been developed to host the parallel paving algorithm and demonstrate its capabilities on both two dimensional and three dimensional surface geometries and compare the resulting parallel produced meshes to conventionally paved meshes for mesh quality and algorithm performance. Sandia's "tiling" dynamic load balancing code [3], [4] has also been extended to work with the paving algorithm to retain parallel efficiency as subdomains undergo iterative mesh refinement.

Acknowledgments

We would like to acknowledge several individuals who have been instrumental in the development of parallel paving and the application of paving within adaptive methods at Sandia. Karen Devine, Bruce Hendrickson, and Steve Plimpton provided valuable consulting on the parallelization issues we faced, and Joe Jung continually exercised the adaptive paver and provided numerous examples which tested the limits of what the adaptive paver could manage.

In addition to Joe, significant assistance with the adaptivity aspects of the work was received from Roy Hogan and Bob Cochran, while consulting was provided by Dominique Pelletier and Steve Attaway. Greg Sjaardema and Malcom Panthaki provided invaluable assistance in the development of the adaptive remeshing code. Randy Schunk and Rich Cairncross also tested the adaptive paving code and provided welcome feedback.

Helpful assistance was also provided by James Peery with the message passing and parallel file reading and writing work. Finally, Rob Leland provided very helpful consultation with parallel efficiency issues and data interpretation.

This work was performed under Sandia's Laboratory Directed Research and Development program.

Contents

DISCLAIMER

Figures

1. Parallel Paving Project Background

1.1 Introduction

The parallel paving project was initiated to apply advanced meshing technology to the solution of parallel adaptive finite element simulations. The finite element method is an established and widely used modeling technique in the engineering analysis community. To use the finite element method, a discretized model of the physical geometry must first be constructed. As computing resources and their availability have increased, the average model complexity has also increased and has fueled the demand for more automation in the model construction area of mesh generation. In the specific case of the nuclear weapons laboratories, solid mechanics' non-linear analyses [5] [6] have historically driven the effort to obtain meshes with quadrilateral and hexahedral instead of triangular or tetrahedral element types [7].

To address these issues, Sandia's Engineering Sciences and Computational Sciences Centers have invested significant effort in automated meshing and finite element model generation software development. Numerous internally developed meshing codes have contributed to Sandia's knowledge base in meshing since the early 1970's (QMESH [8], AMEKS [9], PHARAOH [10]). One of the key meshing codes employed since the late 1980's is FASTQ [11], which is still in widespread use at Sandia. FASTQ produces two dimensional finite element models using several algorithms, including a two dimensional (planar) version of paving. More complex three dimensional models can then be constructed with a suite of complimentary codes which perform extrusions on the two dimensional models and node equivalencing between the resulting volumetric meshes (GEN3D [12], GJOIN [13], and GREPOS [14]). The ExodusII [15] data format is used to exchange finite element model data between these preprocessing codes and the analysis codes across the Engineering Sciences Center technology areas of solid mechanics [5] [6], thermal sciences [16], and fluid mechanics [17]. Almost all of the aggregate capability resident in these initial preprocessing codes has been replicated and significantly extended within CUBIT [2], a two and three dimensional, solid model based meshing toolkit currently under development at Sandia.

CUBIT can create its own internal solid model geometry or can import existing ACIS® [18] solid model files or FASTQ input decks. The ACIS® model format is a widely supported industry standard used internally by numerous commercial vendors, and accessible via translator from the majority of commercially available computer aided engineering software packages. Following geometry creation and decomposition, users can select from several two and three dimensional meshing algorithms to generate quadrilateral and hexahedral element meshes. Among these algorithms is the extension of the original planar paving algorithm from FASTQ; the current version of paving is still topologically two dimensional, but can now create all quadrilateral meshes on any three dimensional surface capable of being supported by CUBIT's internal ACIS® solid

modeler and bounded by at least one curve. These surface meshes can be used in place as shell elements, or they can be extruded through an attached volume to create hexahedral elements. This latest version of paving has also benefited from significant additional testing and performance tuning efforts by both Sandia and multiple private companies who have teamed with Sandia to develop advanced quadrilateral and hexahedral meshing algorithms. CUBIT is also the testbed for Sandia's automated all-hexahedral meshing research, which has resulted in the experimental whisker weaving [19] and plastering [20] [21] algorithms.

Our motivation to develop this newest meshing toolkit is the need for a dual purpose capability: a tool which can be used to efficiently and interactively construct quadrilateral and hexahedral finite element models for conventional analysis jobs (including batch executions and parameterized models), and additionally an environment for meshing and geometric algorithm research, particularly in the areas of general, free form hexahedral meshing, adaptivity (deformed geometry generation and adaptive remeshing [22]), extensions to the traditional meshing techniques of transfinite mapping [23] and extruding [12] (submapping [24] [25] and multi-sweeping [26]), mixed format geometric model representations, and parallel processing.

While the level and accessibility of automation in the finite element model generation task has increased with the new tools, this activity continues to remain a bottleneck in the design to analysis cycle. With the arrival of increasingly fine grained massively parallel computing platforms, more demand is being placed on preprocessing codes to deliver larger finite element models, on the order of 10^8 finite element nodes. In addition, adaptive methods are becoming more common and general. In brief, there is more demand than ever for very large finite element models that can support successive adaptive mesh refinements. The construction of models of this size with traditional preprocessing tools on workstation class machines is very difficult if not intractable in many cases.

While parallel paving is but one piece of a multi-component solution to the problem of the efficient construction of large finite element models, it can serve as a first exploratory effort into the practical extension of Sandia's general advancing front meshing technology to parallel batch operation. As the general free form hexahedral meshing algorithms increase in robustness and maturity, they should be extensible to parallel environments in a similar manner.

One possible future scenario is the combined operation of multiple surface and volumetric meshing operations in parallel to allow the construction of very large, volumetric finite element models. Since the motivation behind parallel paving is parallel adaptive analysis, this work has concentrated on general and extensible methods of performing parallel advancing front meshing.

Thus the parallel paving project has focussed specifically on the extension of paving technology to parallel platforms, and in general on the viability of automated parallel finite element model construction. This report details the conversion of paving into a meshing algorithm which

can be employed in parallel on distributed data. This report will also document the system which allows for the repeated construction of valid, similarly distributed finite element model data which supports repeated adaptive mesh refinement and data redistribution via dynamic load balancing.

1.2 Conventional Paving

Paving was originally developed as a planar, all-quadrilateral automated meshing algorithm for general geometries (FASTQ [11]). Following its initial use and success on widely varying geometries, a subsequent effort was made to extend paving to mesh topologically two dimensional, but non-planar three dimensional surface geometries [27]. This work was first delivered within the solid model based CUBIT [2] code.

Since these efforts, continual improvements and enhancements have been implemented in the current CUBIT version of the paver by both Sandia staff and software developers and mathematicians at private companies, working jointly with Sandia on meshing technology through cooperative research agreements. These enhancements have included significant element connectivity, element quality, performance, and robustness improvements. While fully detailed discussions of the basic paving technique are already available in the literature [1] [27], a short review will be presented here as background context for the parallel paving work.

The motivation for the paving algorithm came from the need for high quality, all quadrilateral finite element models within the solid mechanics community. Conventional means of constructing these meshes meant time consuming geometric decomposition of geometry into regions meshable by standard mapping techniques [8] [11]. Paving eliminated much of the need for decomposition of these two dimensional models through its ability to handle general regions. In addition, paving provided some nice characteristics not necessarily available with traditional methods: paved meshes were boundary sensitive and orientation insensitive.

Because paving progresses in an advancing front manner, it creates its initial and highest quality elements along the boundaries of a geometry, yielding a "boundary sensitive" mesh, a desirable attribute for many types of analyses. Its operation is completely local in scope and is independent of the orientation of the coordinate system it resides in. Thus it is "orientation insensitive", unlike quadtree and octree meshing methods.

Paving is initiated with the construction of an element "front" about each boundary. If a geometry contains internal voids, each void will support its own independent front of elements in addition to the outermost front (see Figure 1-1). Elements within fronts are formed such that they extend perpendicularly away from the boundary they originate from in the direction of propagation. Special cases for elements which fall at locations of sharp change in direction of the bounding curves are managed with predefined connectivity solutions. Typical examples of these cases can be found at natural "corners", where two consecutive element edges meet at an angle within a

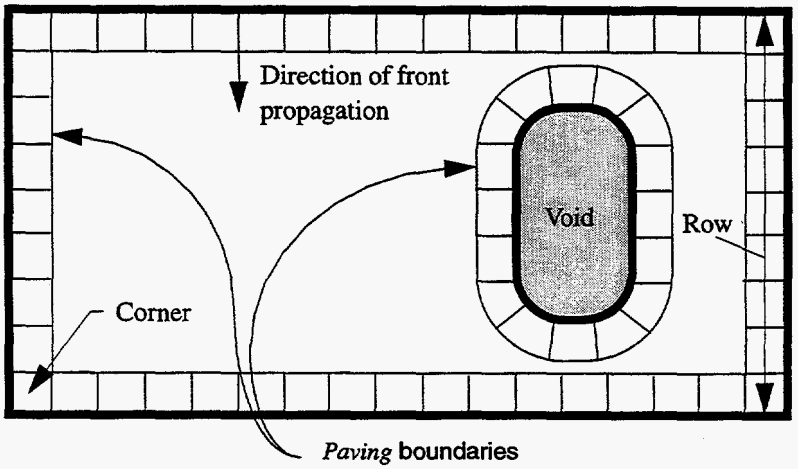


Figure 1-1 Initial paving process: internal void

tolerance value of 90° or 270°, or at a "reversal", where two consecutive element edges make a sharp angle within a tolerance value less than 360° (see Figure 1-2). The original implementations

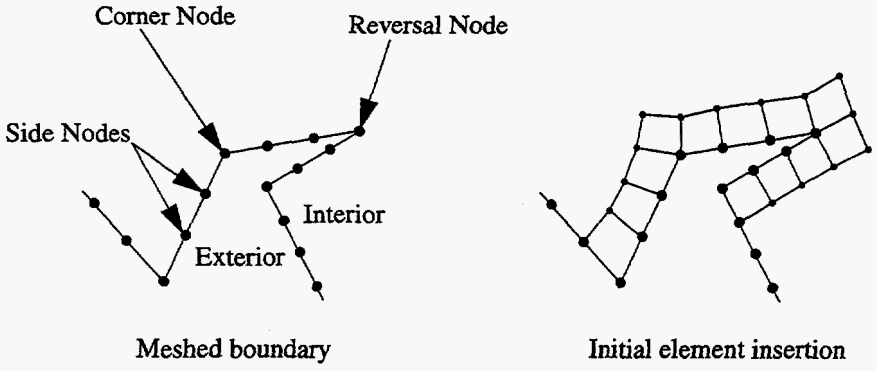


Figure 1-2 Paving row end cases: corner, reversal

of paving defined entire rows of elements to be created at once and the ends of these rows were defined by these special cases, thus their name, "row end cases". The more recent versions of paving in CUBIT [2] construct element faces one at a time, with significantly improved robustness for intersection detection. Paving fronts which are expanding in size (such as those propagating outward from an internal void) maintain consistent element size by inserting new elements referred to as "wedges" along the front (see Figure 1-3). Fronts which are contracting (propagating inward from an external boundary) also work to maintain consistent element size by eliminating elements using a "tuck" operation. As each void on the surface being paved is gradually reduced in size,

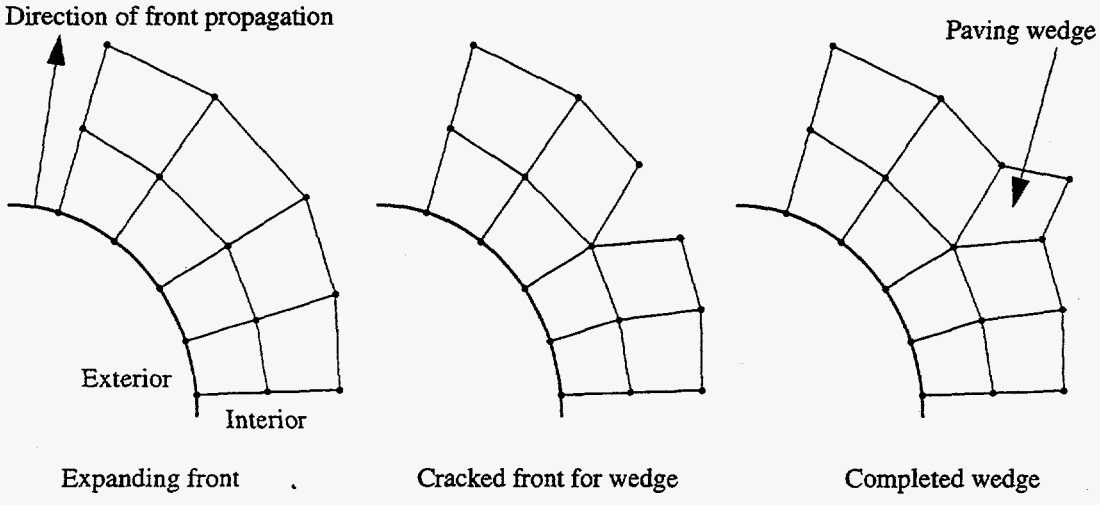


Figure 1-3 Paving front expansion: wedge element insertion

eventually the voids will be reduced to small regions which can be closed with a small group of elements in one of several predefined connectivity arrangements. These predefined element groups are termed closure cases: once the possibility of a closure is detected, the void in question is fully closed (filled with quadrilateral elements) and operation continues to another void until all voids on the surface have been resolved. When a void is bounded by more than one front (such as a geometry which contained an internal void in the original geometry), eventually the fronts will intersect, and one or more new voids will be formed from the intersection. These voids will also be reduced until closure. A typical paved mesh of a general planar geometry with internal voids is depicted in Figure 1-4.

In summary, the paving algorithm operates from the principle that a general region is fillable with quadrilateral polygons provided that the sum of all the polygon sides (or intervals) on the region boundaries is an even quantity. As paving progresses forward, it maintains this requirement of even intervals bounding the remaining void to be meshed (including the case when intersecting fronts create multiple new voids), ensuring that closure remains possible. As the progressing paving fronts collide, expand, and contract, the generated element quality is maintained all along through generous use of length-weighted Laplacian smoothing of element nodes behind the front and isoparametric smoothing of element nodes lying on the advancing front. In addition to the extensive management of the advancing fronts, once the region has been fully filled with quadrilaterals, the "clean up" phase of the algorithm takes control and studies the final filled mesh for possible connectivity and element quality improvements.

Therefore, although paving is generally an advancing front method, it maintains a strong control over the nature of the advancing fronts and how they are allowed to interact. This control

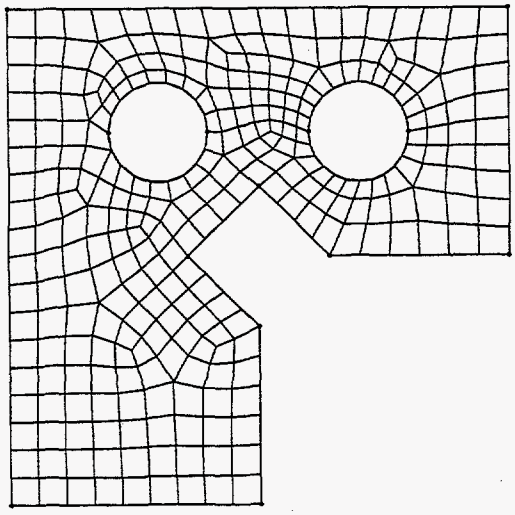
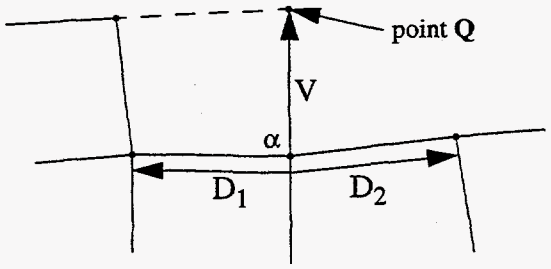


Figure 1-4 Typical final paved mesh: general geometry with internal voids

is part of the reason paved grids exhibit the characteristics they do, such as smooth transitions between varying element sizes and high overall element quality. This control also is why paving is very computationally expensive when compared to a traditional mapping technique for mesh generation.

1.3 Adaptive Meshing with Paving

The quality of elements produced by the paving algorithm is in large part controlled by the mechanism which dictates how new nodes are placed ahead of the current paving front when new elements are being formed. Figure 1-5 illustrates how a new node is placed during a normal paving front progression.



α = inner angle between D_1 and D_2

Figure 1-5 New node placement: normal front progression

$$|V| = \frac{0.5(|D_1| + |D_2|)}{\sin(\alpha/2)} \quad (1)$$

In Equation (1), the relationship between the new nodal location (point **Q**) and its local element neighbors is given where V is oriented such that it bisects the angle between D_1 and D_2. The projection distance outward from the paving front (the length of V) is determined by the surrounding element edges on the front and their included angle. By modifying this relationship to reflect a user defined function, the paving algorithm can produce meshes which capture gradients of the user function with varying mesh element size. To generate new nodal locations from a user defined function, paving samples the background function several times in the local neighborhood of a trial point **Q** initially placed with the normal method for calculating projection distance from Equation (1). Equation (2) indicates a modified relationship that is a function of s and t, where s indicates the default projection distance from normal paving and t represents the average preferred projection distance from the background function in the local neighborhood of point **Q**.

$$|V| = F(s, t) \quad (2)$$

Several examples of adaptive paving are shown in Figure 1-6, Figure 1-7, and Figure 1-8. Figure 1-6 and Figure 1-7 are typical examples of sizing functions that are available in CUBIT [2], while the mesh in Figure 1-8 was generated with input from externally generated data. A simple example of a user defined test function can be observed in Figure 1-6. Another example of sizing

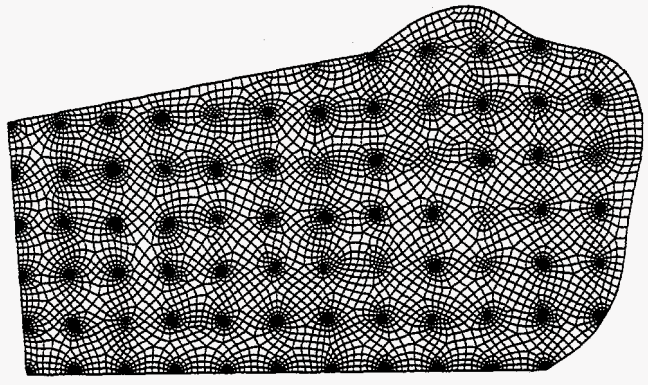


Figure 1-6 User defined test function for adaptive paving

based paving is to allow the background objective function to size the mesh from an observable geometric feature, such as the curvature of the surface. Thus where the surface is rapidly undergoing curvature change, dense mesh will be generated. Figure 1-7 displays a three dimensional surface mesh which was paved with this type of sizing function.

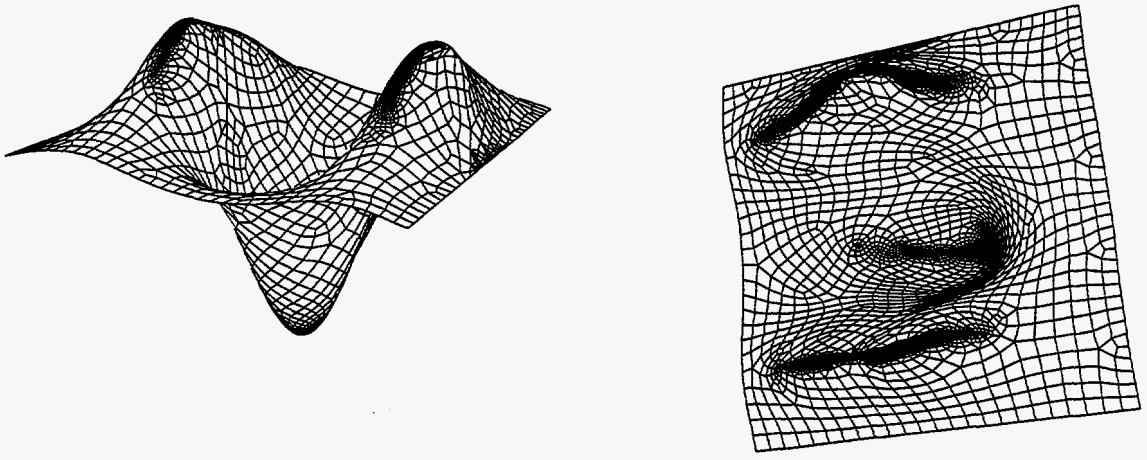


Figure 1-7 Surface curvature mesh sizing function

A practical application of adaptive paving is to base the objective function on an error measure such that elements are concentrated where necessary to minimize error in the finite element solution. When employed iteratively with the analysis and error measure codes, this technique can produce an optimally sized mesh with the minimal number of nodes for the given error tolerance [28]. This approach of completely remeshing the problem domain with a dissimilar mesh is referred to as the h-remap type of adaptive remeshing. A more common "classic h" approach would only subdivide the existing mesh in areas of high gradient in the objective function (or error measure in this case).

An example of using the Coyote [16] heat conduction code with a version of the Zienkiewicz-Zhu [29] projection error estimator which was modified to be used for heat transfer [30] to adaptively remesh a one meter square domain is displayed in Figure 1-8. The left figure shows constant contours of the desired element size as a function of grid location used to minimize the temperature gradient error. The temperature gradients were solved from a steady-state conduction problem with insulated vertical sides and internal void and constant, unequal temperatures at the top and bottom boundaries (copper material with 1000 Kelvins differential between top and bottom). Following nine remeshing iterations, the smallest element size of 0.0125 [m] was requested around the inner void by the error reduction code, increasing outward (in constant contour intervals of 0.015 [m]) to the maximum requested size of 0.0875 [m] along portions of the exterior boundary. After these iterations, the average error was reduced by 86.54% for this problem. Additional description of this technique and an example applying the Babuska-Rheinboldt error estimation method to a linearly elastic statics problem is detailed in Reference [31].

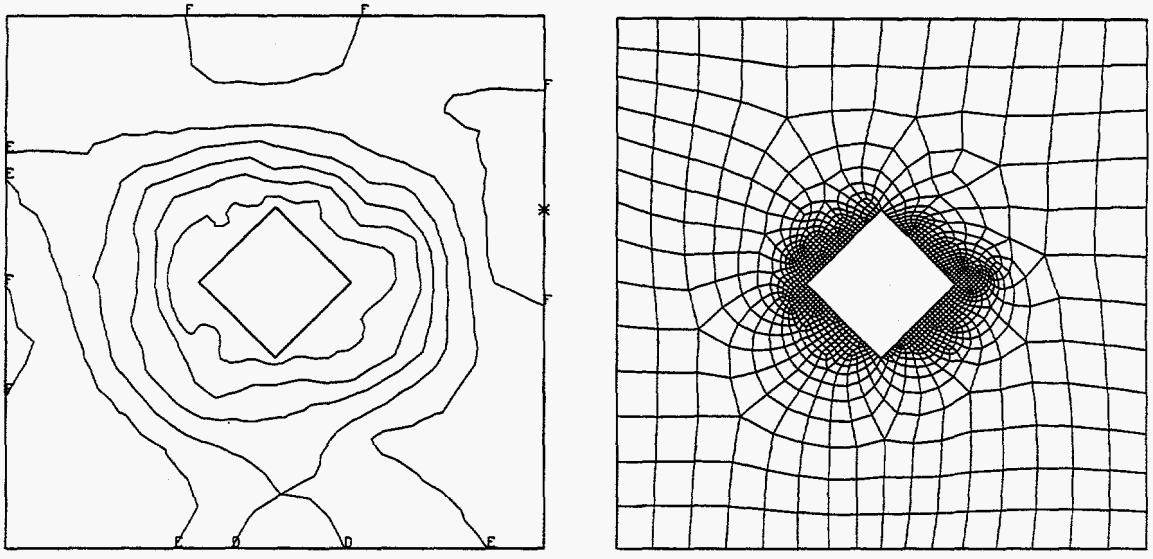


Figure 1-8 Adaptively generated mesh from temperature gradient error

1.4 Design Goals of Parallel Adaptive Paving

The primary motivation of the parallel paving effort is to develop meshing tools to support an adaptive, parallel finite element analysis capability using an "h remap" adaptive remeshing approach. This adaptive approach is the best fit for use with a paving type algorithm, because it operates by generating an entire new mesh to adjust mesh resolution, then remaps the data from the old to the new grid. Other approaches include "classic h" methods (mesh resolution changed by subdividing existing elements), "p" methods (element resolution changed by adjusting the element integration levels), and "r" methods (existing mesh topology unchanged but relocated via grid relaxation).

While mesh generation is but one component of several required to realize this capability, it is a pivotal one. Paving has demonstrated a strong capability in the area of adaptive mesh generation, and can be used as an enabling technology for two dimensional topologies. For the full three dimensional capability to be achieved, a three dimensional analog to paving must be acquired for h remap techniques to be used with hexahedral elements. Even so, a robust parallel paver will allow significant progress to be achieved even if initial adaptive three dimensional efforts are pursued with mixed element (hex-dominant, some tetrahedrons) models. Several options are available for parallel, adaptive volumetric meshing if this scenario is considered, and the development of parallel paving yields important insight into the parallel construction of finite element models in general.

Consequently, the primary goal in the successful rederivation of the paving algorithm is the retention of beneficial characteristics of conventional paving while minimizing the introduction of negative side effects. The earlier discussion on conventional paving provided sufficient detail on the major benefits of paving: geometric generality, boundary sensitivity, and orientation insensitivity. The preservation of these characteristics is central to the parallel paving development effort, and is reflected in the design decisions that were made.

A second major goal was to conceive a parallel meshing strategy that could be fully extended to other algorithms for two and three dimensions: the achievement of this is required to meet the overall goal of three dimensional adaptive parallel analysis. The intent of this work is to prototype potential meshing tool designs with parallel paving, then focus on a best suited prototype to implement for current and future use. If the prototype parallel meshing tool design is well conceived, then the eventual implementation of a volumetric meshing algorithm should be limited to the algorithm input and control development only, as the meshing tool infrastructure and mesh data management and communication will already be planned for.

An important goal for parallel paving is the construction of a meshing tool that behaves in a nearly identical manner as conventional paving. If the parallel algorithm were to become overly sensitive to control input, for example, then its strength of generality would deteriorate unacceptably. In addition to the algorithm's response from a user's point of view, the commands to control and initiate the parallel meshing process should be very similar if not identical to the conventional suite of commands for creating normal or adaptive paved meshes. The rational is simply to divorce the implementation from the usage of the algorithm: once users have developed their methods of controlling the paver in its conventional form, these methods ought to extend directly to the parallel version of the algorithm.

In addition, if core algorithm code compatibility can be achieved everywhere possible between the parallel version of paving and the convention one, the value of the parallel capability will be tremendously increased. This core algorithm compatibility is important for the future reintegration of a parallel meshing capability back into the conventional, production mesh generation tool, CUBIT [2]. Because the parallel paving development will still rely on a capable serial paving implementation, this integration is very important to allow parallel paving to benefit from the continual improvements which are made to the conventional paving code located in production CUBIT.

A final important goal for the parallel paving development project is the strict adherence to established standards for code data and method structures and finite element data exchange formats [15] wherever possible. Parallel paving was integrated into the CUBIT meshing toolkit to first allow for better integration back into a production meshing tool, and secondly to accelerate the development of parallel paving by reusing needed data constructs and capabilities already existing

within CUBIT. Both conventional CUBIT and parallel paving CUBIT (CUBITP) are constructed entirely in the C++ language which allows for straightforward encapsulation of capability. By using this environment, parallel paving can immediately take advantage of a number of standards including ExodusII finite element data exchange, ACIS® and FASTQ geometric input data, and various graphical capabilities for both debugging and performance testing (graphics disabled). Following this idea, the parallel paving effort has embedded its communication methods within a generic communication tool to allow additional communication standards to be added in the future with no impact to the parallel meshing and finite element model construction code. Throughout this report, descriptions of "classes" will be included, which will refer to the C++ classes used in the design of the parallel meshing tool, CUBITP.

The remainder of this report is structured as follows: Chapter 2 will describe the design decisions and implementation of the parallel paving algorithm, Chapter 3 will address the role which parallel paving and load balancing play in the adaptive process, Chapter 4 will provide numerical results for parallel performance and mesh quality, and conclusions are made in Chapter 5. Relevant details of example problems and selected code listings are included in the Appendices.

2. Design and Implementation of a Parallel Paving System

In this chapter we will discuss the process which led to the final design implemented for parallel paving into the CUBITP code. This process includes the introduction of a new geometric representation system as well as the steps which must be followed to use parallel paving in the construction of a distributed and complete finite element model. Since our scope is limited to issues surrounding the operation of paving in parallel, we will not detail the extent of capabilities required for full volumetric finite element model generation, but we will describe how these extensions have been accounted for in our design decisions and in the implementation.

2.1 Design Alternatives

At the beginning of this investigation, multiple approaches to the problem of redesigning the paver while producing equivalent output were considered. The paver is inherently serial and non-deterministic in that the construction, placement, and size of each subsequent element depends on the previous element. Thus to operate in a manner which is no longer subject to these constraints is to solve the automated quadrilateral meshing problem anew. Our intent has not been to resolve the paving problem, but to take as much capability as possible and recast it such that it operates efficiently and within a subdomain using mixed geometric representation. The following design ideas are described to document this intent to reuse existing technology and how well each fits into the final goal of parallel adaptive meshing.

Two approaches have been considered to redesign the paver for parallel operation. One approach employs a moving or dynamic boundary definition for a paving meshing process to operate in, and the other approach uses a static or constant area subdomain for the local paving operation.

2.1.1 Moving Boundary Front Approach

The moving boundary, or parallel advancing front approach was conceived to retain more of the serial paving algorithm's desirable characteristics, the attributes of boundary sensitivity and orientation insensitivity (discussed in Section 1.2 on page 3) by avoiding the placement of a fixed decomposition boundary on the problem geometry. The moving boundary approach operates by placing paving "seed" initiation points in the center of each subdomain, and paving outward from those seeds until a neighbor subdomain front is reached. Front resolutions are then accomplished between neighboring processors.

Figure 2-1 displays a sample geometry of the initial domain for a moving boundary front paving run with a decomposition from the background mesh (background mesh not shown) overlaid. This approach assumes the existence of decomposed background mesh (the construction

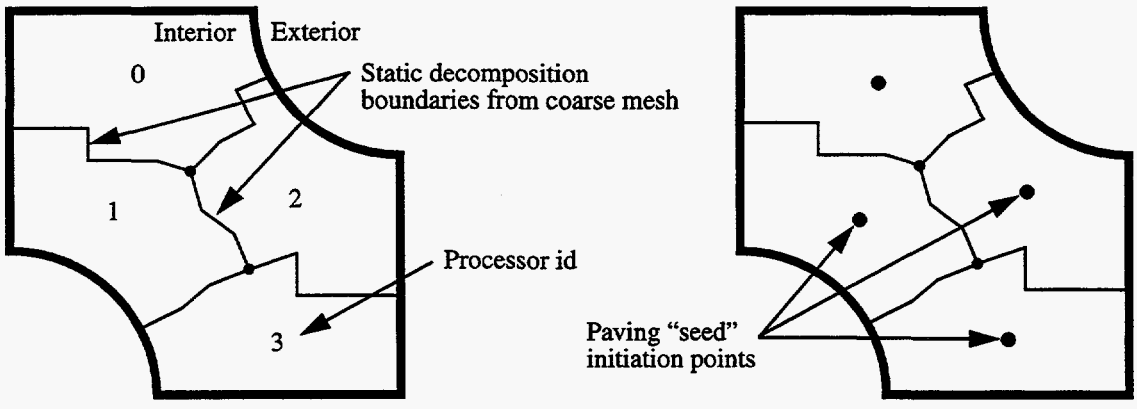


Figure 2-1 Initial coarse decomposition and seed points for moving boundary approach

of this is discussed in Section 2.3.1 on page 25) that has been distributed onto a parallel machine. The background mesh can be from either a previous iteration or an initial coarse mesh. In this example, each subdomain is owned by a single processor, although this is not a requirement because a processor can manage multiple subdomain regions if the decomposition doesn't exactly match the number of available processors. Also indicated here are the paving "seed" initiation locations, located at the centroids of the subdomains. A region slightly extended beyond the coarse mesh decomposition for the subdomain is considered the "region of influence" of the processor for that subdomain, and is purposely not fixed directly to the decomposition boundary.

To create a mesh with this technique, paving begins expanding a front outward from the seed location until it intersects a processor boundary at which time it freezes the intersecting front and signals the neighboring processor at the intersected boundary. If a front intersects a normal geometric boundary, it attaches the front to that boundary at an element size appropriate for the requested size of the local neighborhood of the boundary. If an interior void is completely included within a subdomain, a front is defined about that void and is handled as if the void were another seed point. Figure 2-2 indicates the fronts forming at the seed initiation points. Once a front has intersected a neighbor processor boundary, it retains the intersecting mesh topology, but waits (freezes) until the neighbor processor front intersects the same boundary, at which time the fronts are joined via conventional paving methods (see Section 1.2 on page 3).

The motivation behind this method of paving is to minimize the introduction of artificial geometric artifacts from the coarse mesh decomposition. By not forcing the decomposition boundaries to become part of the fine mesh *a priori*, the resulting fine mesh will more naturally represent a paved surface as the advancing front collisions at the decomposition boundaries are sewn together with conventional paving methods. To achieve this mesh characteristic, however, a

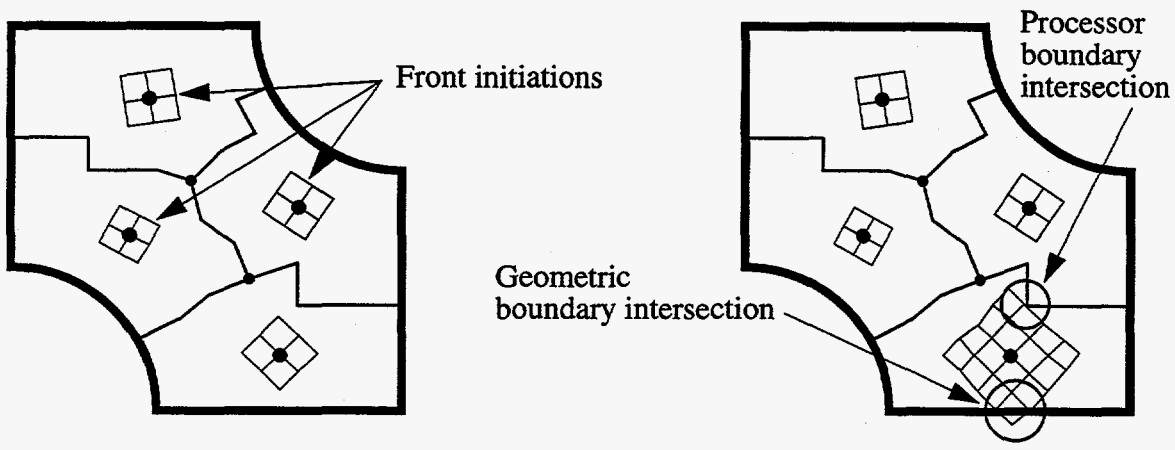


Figure 2-2 Front formations at seed points for moving boundary approach

significantly more complex communication and mesh front management scheme must be supported, which suffers from timing sensitivity. If a processor has a background sizing function significantly coarser than a neighbor has, it will reach the boundary of its subdomain, or region of influence significantly faster than its neighbor, and will bottleneck the process because it has to wait for its neighbor to resolve the boundary intersection. An obvious solution is to manage the boundary regions first to avoid the timing difficulties, which is precisely the major difference between the moving boundary approach and the constant area technique, which will be described in the next section. An additional issue with the moving boundary method is the problem of orienting the initial seed mesh at the centroids of the subdomains, which if not managed correctly, can induce an unintended orientation to the final mesh. Also some boundary sensitivity at the geometric boundaries can be lost with this approach since no control is exercised over the manner in which the fronts impact the geometric boundaries.

2.1.2 Constant Area Subdomain Approach

The constant area subdomain method of parallel paving is designed to be more like a conventional paving operation, and as such, operates with significantly less complexity than the moving boundary method. The constant area approach operates by first meshing the boundaries of each subdomain, then advancing inward in the same manner as conventional paving uses to close a void region. Following closure of the subdomain, the initial inter-processor boundaries of the subdomain can be smoothed between processors as needed to mitigate any intruding geometric artifacts left from those artificial boundaries.

This method also assumes the existence of a decomposed background mesh (see Section 2.3.1 on page 25) that has been distributed across the parallel processors. The background mesh can be from either a previous iteration or an initial coarse mesh. As mentioned, the constant area

paving approach employs paving in essentially the same form in which it operates serially, but contained within a subdomain. The boundaries of the subdomain (the coarse mesh decomposition boundaries) are used as geometric boundaries to be discretized in the same way as CUBIT meshes the bounding curves of a surface prior to constructing a surface mesh. The definition and construction of these interior boundaries (represented by previous mesh data, not solid model entities) will be discussed in Section 2.2.3 on page 23 and Section 2.3.2 on page 26. The primary task then is to negotiate the mesh on subdomain boundaries which are shared by two processors.

Each subdomain (Figure 2-1 and Figure 2-2 contain four subdomains) ensures that its bounding curves are meshed prior to paving initiation. If a bounding curve is shared by two processors, the processor with the lower processor id is deemed to be the owner, and as such is responsible for meshing the curve and communicating the resulting mesh to the adjacent processor which relies on the curve. If a processor does not own a curve that it needs for the bounding mesh data, it waits until it receives this data from its partner, the owner of the curve. Once all of a surface's boundaries have been meshed (including any internal voids), paving initiates within the surface as if it were a conventional geometric surface (see Section 1.2 on page 3). Figure 2-3 uses the same sample geometry to show a typical initiation of the constant area method of parallel paving as processors 0 and 1 are forming the first necklace of mesh rows away from the bounding curves and into the void areas to be meshed.

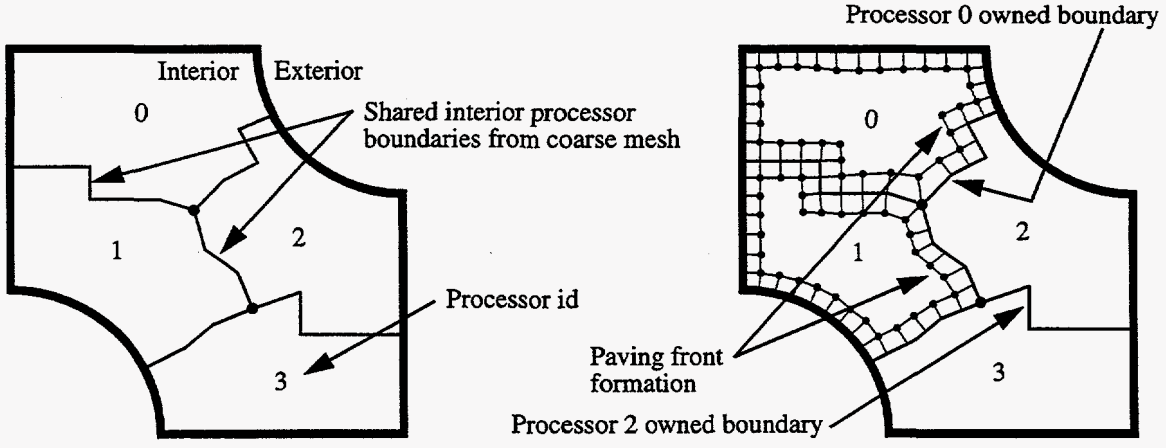


Figure 2-3 Initial coarse decomposition and front formation for constant area approach

When the bounding curves of a surface are meshed, the number of element edges created on a given curve is constrained to be even. The closure requirement for an all-quadrilateral mesh is that the sum of all the edges that the void to be meshed "sees" is an even quantity. Our implementation ensures that this requirement will be satisfied because the sum of all the element edges bounding the void will be even. The mesh being created in Figure 2-3 exhibits a uniform

element size, but when a background sizing function is introduced (see Section 1.3 on page 6), gradients in the sizing function will introduce mesh size transformations in the generated mesh, and for regions with coarsening transitions along internal shared subdomain boundaries, the mesh edges along these boundaries will seem to deviate from the actual curve definition at the internal boundary. This is natural and expected, and is discussed in further detail at the end of Section 2.3.2 on page 26.

Communication requirements of the constant area approach are reasonable, and primarily involve the resolution of the interior curves which lie along shared processor boundaries (see Figure 2-3) as each processor discretizes its owned curves with locally available size data and sends the resulting mesh to its partner processor for shared curves. This sizing data at the curves is guaranteed to be consistent between processors by design, although some additional communication can be required for certain types of sizing data (see Section 3.4 on page 38). This consistency is possible since the sizing data will only arise from the background mesh (the mesh from the previous iteration), and the values at the nodes and elements at the subdomain boundaries of the new decomposition will be the same since they both originated from the same background mesh.

Given the advantages of low communication needs and significantly less complex design, the constant area method of parallel paving was selected for full implementation as the most viable approach. The issues of boundary sensitivity and orientation insensitivity are managed well by the constant area approach, benefiting from the natural strength of the conventional paver which is implemented unchanged into this method. While generality is maintained with the constant area method, artificial geometric constraints are created by statically defining the paving region boundaries between processors from the parallel decomposition. This issue of introducing geometric artifacts into the final mesh from the previous decomposition and methods to mitigate this are discussed at the end of Section 2.3.2 on page 26, after the description of the virtual geometry model.

2.2 Virtual Geometry Model

One of the challenges of employing the constant area parallel paver technique is the minimization of unwanted geometric artifacts due to the decomposition of the computational domain. Typical parallel computations construct a discretization using conventional serial tools, then partition the domain into the desired number of subdomains using one of several standard partitioning techniques [36]. These partitions do not adversely affect the calculations because they are transparent to the physics being simulated. For our purposes, however, the task discretizing the domain is a parallel process, and it can be significantly impacted by the previous discretization and decomposition being used.

An interesting approach of using a coarse background mesh for an initial static decomposition, then continuing to refine the mesh within the original decomposition and dynamically adjusting the decomposition as needed was suggested in Reference [32]. This basic technique is employed in our implementation, and because the paving algorithm is employed for the background coarse mesh, the geometric artifacts are as boundary sensitive as possible for the level of refinement present in the initial coarse model.

The refinement of the initial coarse mesh into a denser mesh, and the subsequent changes to the decomposition from an integrated load balancer require a flexible and dynamic geometric representation for the subdomains. Fundamental differences between the geometric representations enjoyed by conventional paving and the constant area parallel paver will be presented in the following sections on the ACIS® solid modeler, the reference entity data structures in CUBIT [2], and the virtual geometry paradigm replacement for those data structures in CUBITP.

The ACIS® section is intended to provide sufficient insight into the structure of how ACIS® defines solid model topology, since conventional CUBIT has historically used ACIS® for topology traversal, and has its reference entity structures modeled after the ACIS® geometric entities. Following this discussion, the section on reference entity data structures in CUBIT explains how conventional CUBIT employs reference entities to support non-manifold models using ACIS® as the underlying geometric representation. Finally, the section on the virtual geometry paradigm details how the interface to the geometric topology and geometric interrogations is now divorced from the underlying geometric representation, and how non-manifold data can be represented as before whether ACIS® or any other format is being used for geometric representation.

2.2.1 ACIS® Manifold Solid Modeler

The ACIS® [18] modeler is a boundary representation solid modeler, or a modeler that maintains a complete topology and geometric curve and surface definition of geometry for an unambiguous model definition. Figure 2-4 displays the hierarchy of ACIS® solid model entities. In

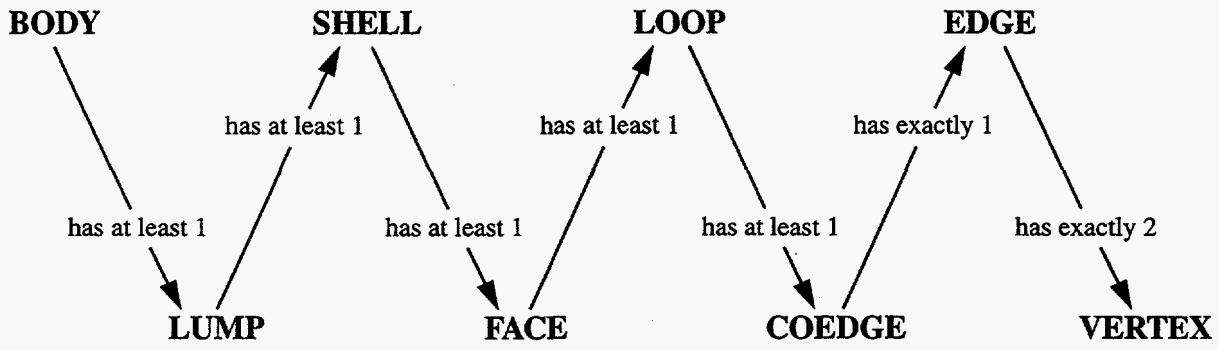


Figure 2-4 ACIS® entity hierarchy for manifold non-wire geometry

this hierarchy, which pertains to normal, manifold [33] (non wire) solid models, a **BODY** is a collection of one or more **LUMP**s, or valid closed volumes. These **LUMP**s need not be continuous, they can be disjoint and yet part of the same **BODY**, thus in this sense, a **BODY** is somewhat like an assembly. **LUMP**s contain one or more **SHELL**s, which are containers for a series of one or more continuously connected **FACE**s, or surfaces. As an example, a volume with a completely enclosed interior void would be represented as a **LUMP** with two **SHELL**s, one defining the exterior of the volume, and the second comprising the definition of the interior void.

FACEs in turn contain one or more **LOOP**s which each contain at least one **COEDGE**. **LOOP**s are ordered lists of contiguous, chained **COEDGE**s, and have an orientation with respect to the owning **FACE**. The orientation indicates on which side of the **LOOP** the material of the **FACE** is, because the underlying surface definition will be an infinite geometrical surface definition. If only a single **COEDGE** were present in a **LOOP**, it would have to be self-connected (its start point being spatially equivalent to its own end point) as on a disc, such as the end cap of a cylinder. Thus **COEDGE**s have direction (or a "sense"), and all chained **COEDGE**s are oriented in the same direction such that the **LOOP** which they form has a consistent orientation. **COEDGE**s in turn have exactly one **EDGE**, although an **EDGE** will have *exactly two* **COEDGE**s referring to it. Finally, an **EDGE** has exactly two **VERTEX** objects, for its start and end point entities. If an **EDGE** is self-connected, then its two **VERTEX** objects can refer to the same spatial point (consider the disc example mentioned above), yet there will still be the two separate objects in the topology definition.

Not all of these entity types represent physical geometry, some are grouping mechanisms and some are signing mechanisms. The following entities have physical counterparts and topological functions: **LUMP** (represents volumes and terminates or bounds a half-space), **FACE** (represents surfaces and terminates or bounds a **LUMP**), **EDGE** (represents curves and terminates or bounds a **FACE**), and **VERTEX** (represents points and terminates or bounds an **EDGE**). The **BODY**, **SHELL**, and **LOOP** entities are grouping mechanisms, and **COEDGE**s are signing mechanisms.

Figure 2-5 shows how these entities are used to define the topology of a simple model, and clearly shows the role of the **LOOP** and **COEDGE** mechanisms. By the consistent usage of these topology entities, ACIS® is able to model quite complex geometries very concisely. Details on the mathematical definitions of manifold cellular topology geometry can be found in [34], but a succinct and practical definition is as follows: a "manifold" **EDGE** can bound or be lengthwise attached to exactly two **FACE**s or surfaces simultaneously. For example, if an interior vertical web surface were added to Figure 2-5 such that it attached to the top, bottom, and side surfaces, the **EDGE** (and hence the model) would no longer be manifold, because an **EDGE** would exist at each location where the web surface met the bounding surfaces, and that **EDGE** would be attached to

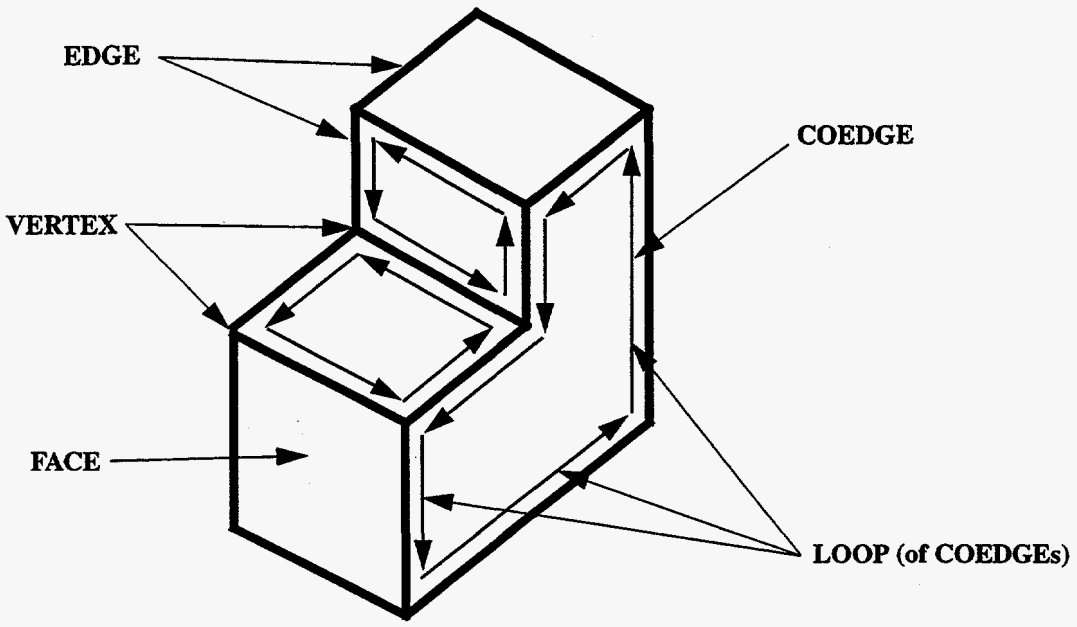


Figure 2-5 Simple geometric model defined by ACIS® topological entities

three **FACE**s at once (when a new **FACE** attaches to an existing **FACE**, it will split or break the original **FACE** at that location).

CUBIT (and conventional paving) uses the ACIS® kernel as a geometric modeler, yet since finite element models are typically non-manifold, it employs a series of data structures to manage the non-manifold relationships of the finite element data. These data structures and how the non-manifold relationships are established will be discussed in the following section.

2.2.2 Non-manifold Reference Entity Overlay

CUBIT is designed to facilitate the construction of finite element models (non-manifold) from manifold computer aided design (CAD) data. To achieve this, it has to maintain a level of indirection in its data structures for finite element model data. This level is referred to as the reference entity structure in the CUBIT documentation (see [2] and [35]). This structure provides containers to store finite element model data, such as mesh data (nodes, elements, element connectivities, boundary conditions, meshing algorithm schemes, adaptive sizing data) and ensures that the non-manifold requirements are supported.

Finite element models are almost always non-manifold because when two elements share an edge, they define a non-manifold body by the definitions provided in Section 2.2.1 on page 18. Furthermore, when an all-hexahedral or all-quadrilateral element model is desired or when there is simply more than one material contained in a model, some decomposition is often preferable if not required. If a boolean operation is made to perform the decomposition (cutting an existing solid in

the model into one or more regions), then the result in CUBIT is more manifold solid model entities. By default, the reference entity overlay structure in CUBIT remains manifold as well until the user specifies how the non-manifold relationship is to be constructed. Figure 2-6 shows an example of how the reference entity overlay works on a manifold model before and after it is cut, or decomposed via a boolean operation. The graphic on the right side of Figure 2-6 has been cut in

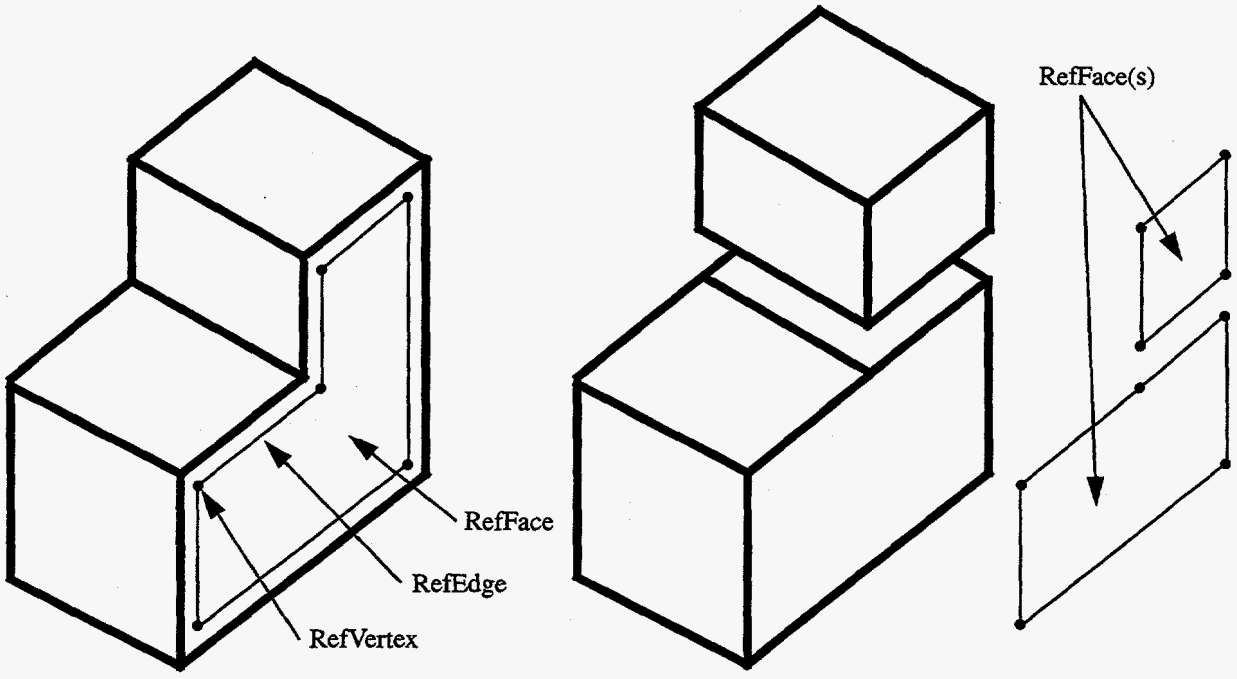


Figure 2-6 Manifold reference entity overlay with original and cut solid model

the plane of the lower step surface, but it has not been translated. The exploded view with the cube translating upward is for clarity. The reference entity structure in the rightmost surface (**FACE**) is displayed while the remaining reference entities are omitted for clarity. This default manifold representation of the reference entity overlay will always exhibit a one-to-one mapping between the physical ACIS® entities and the CUBIT reference entities (**LUMP** -> RefVolume, **FACE** -> RefFace, **EDGE** -> RefEdge, and **VERTEX** -> RefVertex).

To change this one-to-one relationship and invoke a non-manifold reference entity overlay to allow a continuous mesh, a user executes a command which directs CUBIT to combine all the redundant reference entity data structures, or structures that represent spatially equivalent ACIS® entities. This is the *merge* command in CUBIT [2] (see Appendix A.1 on page 55). The merge operation searches the geometry for any spatially equivalent geometric entities (spatially matching surfaces, curves, and vertices) and "merges" their respective reference entity counterparts when a match is verified. Users can also selectively exclude entities from this search and combine

operation to allow models to support discontinuous mesh when desired. Options on the merge command include the ability to specifically merge selected pairs of surfaces and curves. Figure 2-7 shows the same model after a merge operation, with the now combined reference entity overlay structure modified for non-manifold data and continuous mesh. Once mesh data is created and

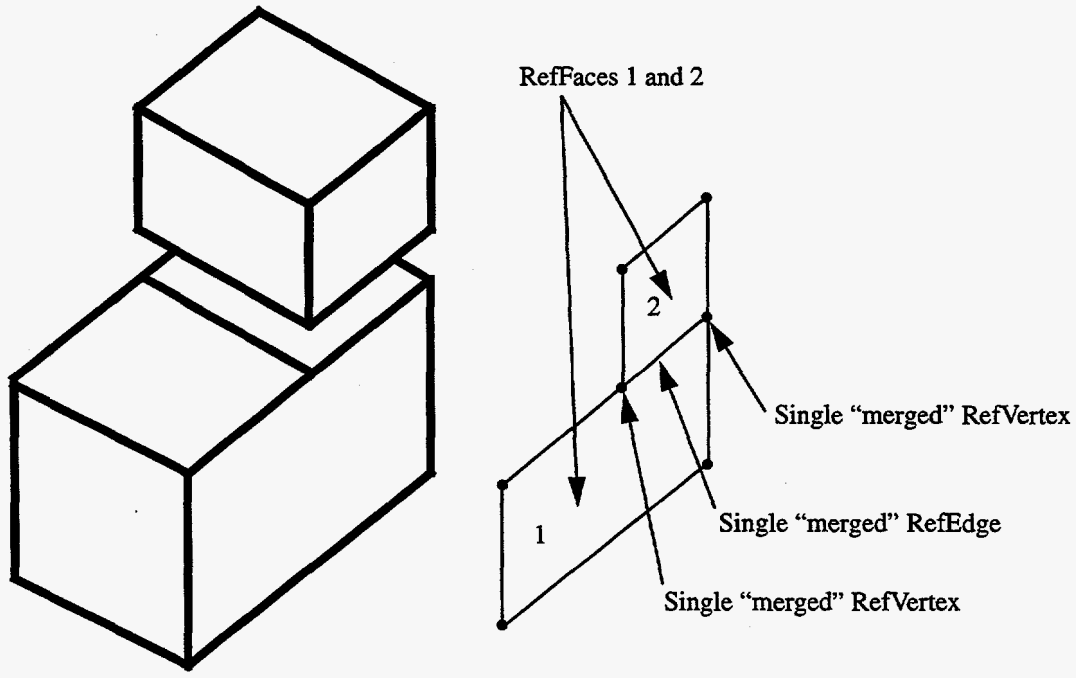


Figure 2-7 Non-manifold reference entity overlay following merge operation

stored on a reference entity, it is employed by all future uses of that entity. For example, if the upper RefFace indicated in Figure 2-7 was meshed, the mesh created for the merged RefEdge which bounds both the upper and lower RefFace would be used for both surface meshes. This shared RefEdge is an intentional artifact of the cutting operation on this solid, which now allows two separate materials or meshing algorithms to be employed on the resulting decomposed entities, where before only a single material or meshing algorithm could have been used. Once a reference entity is meshed, it uses that mesh data for all subsequent mesh requests unless the mesh data is specifically deleted. In a similar manner, the RefFace at the division between the upper cube and the lower parallelepiped would be merged to ensure a continuous mesh between the two volumes.

This technique is repeated throughout a model to generate a continuous finite element mesh model. For all but simplistic geometries, the interaction of all the possible merged entities can quickly become complex. To assist in managing the complexity, CUBIT employs a volume coloring paradigm to visually aid the manifold status of the geometry (see Appendix A.1 on page 55).

2.2.3 Virtual Geometry Representation

The conventional paving algorithm in CUBIT operates solely on geometries which are represented within CUBIT's reference entity data structures described in the previous section. In addition, these reference entities will only represent geometry which can be supported by geometric entities defined within ACIS®, as described in Section 2.2.1 on page 18. While the ACIS® modeler is very capable, its traditional curve data types are either linear, elliptical, spline based, or the result of the intersection of two surfaces. While almost all CAD models can be constructed of these data types, for parallel meshing the requirements are somewhat unique. The input data to parallel paving for some of the model entities (interior curve boundaries shared between adjacent processors - see Section 2.2 on page 17 and Section 2.3.2 on page 26) will be faceted while for other entities it will be conventional and analytic (normal geometric boundary curves). This mixed (faceted and analytical) collection of geometric data presented a challenge to represent in a manner that appears consistent to the meshing algorithm.

One of the initial prototype implementations of the constant area parallel paver relied strictly on the generation of spline curves to represent the interior shared processor boundaries. When the subdomains were defined by the partitioning code, CUBITP used curve fitting calls within ACIS® to generate the spline curves, and subsequently created surfaces from connected loops of these curves. The advantage of this idea is the complete reuse of existing ACIS® and CUBIT data structures and relationships, which allowed the development to concentrate on the parallel issues at hand. The key drawbacks, however, were the computational expense of fitting the spline curves and the inflexibility of the curves once they were constructed. The splines could be modified, yet the cost for the benefit of performing this modification was not very attractive. Even if stretched very tightly, a spline representation of faceted data is still a weak approximation when compared to a linearly segmented representation. Since CUBIT requires that mesh nodes generated on a curve must in fact lie on that curve, problems could arise since the underlying mesh data might contain sharp corners or even kinking. Accurate representation of this data with a spline curve could be difficult if not impossible. As a result of these factors, this approach was terminated in favor of other alternatives.

A second obvious solution would be to simply represent each facet with a distinct linear curve complete with RefEdge and underlying ACIS® entity. While this is a possibility and could be implemented very quickly within CUBITP, it has several significant problems. The most fundamental drawback is the inability to coarsen a grid if requested by a background sizing function since the data structure within CUBIT and CUBITP cannot support mesh entities which span multiple reference entities. In other words, a single RefEdge cannot support an element edge which has either of its end nodes beyond the end of the RefEdge itself. A second issue would be the proliferation of reference entity objects to the resolution of the coarse background grid,

resulting in an unacceptable overhead demand on the available memory space. For these reasons, this approach was never pursued to the implementation stage.

The solution selected to address the problem of representing the faceted and conventional geometric data simultaneously was to devise alternative curve and surface representations complete with geometric evaluators for each data type. A virtual geometry interface was defined to abstract the geometric queries out from the underlying implementation. While the existing RefEntity structure was left mostly intact, the previous access directly to the ACIS® modeler was removed and replaced with calls into the virtual geometry interface.

Three types of representations have been conceived: conventional ACIS® representation, mesh topology representation (facetted mesh data for geometry), and composite representation (a mixture of the first two within a single geometric entity). The first two representations have been implemented for surfaces, curves, and vertices. Standard geometric queries supported include "move to" (calculate a vector which would relax the input point onto the geometric entity) and "normal at" (calculate the normal from the geometric entity from the input point on the entity). For mesh topology surfaces, the "normal at" queries are passed through to the underlying RefFace since it exists for our geometries, if it was not present, the query would be resolved from the element facets.

For mesh topology curves, parameterized models are maintained of consecutive linear segments and standard parameter queries (parameter at point, length to parameter, contains) are supported in the MeshTopology classes (see Appendix B on page 61). The conventional ACIS® curves have these queries supplied already. The topological relationships which must be created for MeshTopology representations are derived from the background mesh data connectivity for mesh topology curves and surfaces. Since the background mesh already has a valid topology, a new topology can be constructed using the element connectivities from this previous mesh. The construction of the virtual geometry representations will be discussed in Section 2.3.2 on page 26.

With the virtual geometry paradigm, the one-to-one mapping between the analytical ACIS® geometry entities and the reference entities (see Section 2.2.2 on page 20) is relaxed since mesh data can also represent geometry. The advantages of this system are a consistent geometry interface for the meshing algorithms, regardless of geometry representation, a valuable characteristic. In addition, geometry definitions are flexible in that if the underlying topology of a mesh topology curve needs to be changed rapidly, a simple modification of the underlying element edge list completes the change. This will be needed as dynamic load balancing (see Section 3.2 on page 34) adjusts the subdomain boundaries for efficiency. Any sharp corners are resolved automatically by the linear curve segments, no curve fitting required. The challenge of this method is the need to develop robust, efficient, and unambiguous geometric evaluators for the alternative data types,

primarily mesh topology data in this effort. In addition, a valid topological tree must be supported for the mesh generation code to be able to traverse the geometric model.

These challenges have been met within the scope of the parallel paving effort (support for mesh topology vertices, curves and surfaces), yet they will still exist in a somewhat more complex form for volumetric data (mesh topology volumes). The potential benefits of this method are quite significant, so much so that the production version of CUBIT has an effort underway to replace its conventional reference entity scheme with the virtual geometry paradigm.

2.3 Parallel Mesh Generation Process

The following several sections outline from a user's point of view the process followed for parallel paving, with additional detail provided for topics that are specific to parallel meshing and code capabilities that are essential to the parallel operation. Briefly, this will cover the tasks of initial coarse model construction, decomposition, and subdomain initialization. Included will be more detail on the virtual geometry model representation, meshing scheme and size control, the propagation of element block and boundary condition information, non-manifold model issues, and exporting the distributed data. The overall process will be described here, but for specific command syntax and examples, see the discussion and journal files for generating paved meshes in parallel in Appendix A on page 55.

2.3.1 Initial Model Construction and Decomposition

For the initial finite element model, any conventional geometry construction method can be used. This can include creating geometry within CUBITP or CUBIT with solid modeler construction techniques, or it can be accomplished by importing a FASTQ [11] input deck, which includes geometric construction information, boundary conditions, and meshing control data. The mesh resolution of this model can be coarse or fine, but for this investigation we always constructed the initial model to be coarse.

These conventional modeling and meshing techniques are covered in detail in Reference [2], and further discussion of conventional meshing methods will not be addressed here. Because the focus of this effort is the generation of paved meshes, paving the initial course grid will yield the most desirable final refined mesh from the parallel meshing process. While this step is not required, it will have significant impact on the final grid. By using paving as the coarse background generation method, the issues of geometric artifacts intruding on the final mesh will be mitigated to a significant extent.

Once all required model construction has been completed, all needed boundary conditions (nodesets and sidesets, see [2], [15]) have been applied and element block information constructed, the user *must* generate the topology records for storage into the ExodusII [15] file format. The need for and the mechanism of storing these records is addressed in Section 3.3 on page 36.

The *nodeset associativity* command (see Appendix A.1 on page 55) in CUBITP or CUBIT will generate the necessary topology data for export to the ExodusII file format. Following this step, once the model is completed with all needed finite element model data and boundary conditions, the model can be exported to an external file (for sample journal files of coarse model construction, see Appendix A.2 on page 57 and Appendix A.4 on page 58).

The initial decomposition of the coarse mesh file for this investigation was performed with the standard partitioning tool within the Engineering Sciences center at Sandia, EDDT (Exodus Domain Decomposition Tool). EDDT is a wrapper around Chaco [36], a popular domain partitioning code. The inertial partitioning method was used for the domains discussed in this report. EDDT applies Chaco's partitioning methods to finite element models stored in the ExodusII file format, and creates the distributed files in the ExodusII file format. An analogous code to EDDT is EDCT (Exodus Domain Concatenation Tool) which reconstructs a single model for postprocessing from distributed files. Other domain decomposition tools could have been used, yet EDDT was the most convenient for our environment due to its ExodusII [15] compatibility.

Following the successful decomposition and distribution of the coarse model and the preparation of a journal file for batch parallel meshing, the model is prepared for the meshing task.

2.3.2 Virtual Geometry Model Construction

Within CUBITP, the subdomain representation is constructed with two separate steps. First, a geometric model is instantiated in the conventional way and represented by the conventional reference entity objects in CUBITP. These objects will be accessing the conventional solid model entities but through the virtual geometry interface contained in CUBITP. At this point, the decomposition output by EDDT is used to construct subdomains when the user imports the EDDT files.

A faceted geometry representation is then constructed automatically for those boundaries which are defined only by mesh data, in this case the interior shared processor boundaries. These boundaries are defined within the suite of mesh subdomain files (produced by EDDT) by node set and side set definitions which indicate adjacent processors in the decomposed model. Faceted geometry representations are also constructed for curves which are split between two or more processors. In this case, if a single conventional RefEdge spans through two or more subdomains from the EDDT decomposition, each span will be represented by a separate MeshTopology curve.

Consider the example geometry shown in Figure 2-8. After construction of the coarse mesh, the model is partitioned into four subdomains and stored in separate files. CUBITP is then initiated on each of the four active processors. The original geometry is loaded first to create the background solid (See the left side of Figure 2-9. Vertex, Curve, and Surface reference entities are denoted as all physical geometry entities are given reference entity counterparts.). Then the distributed

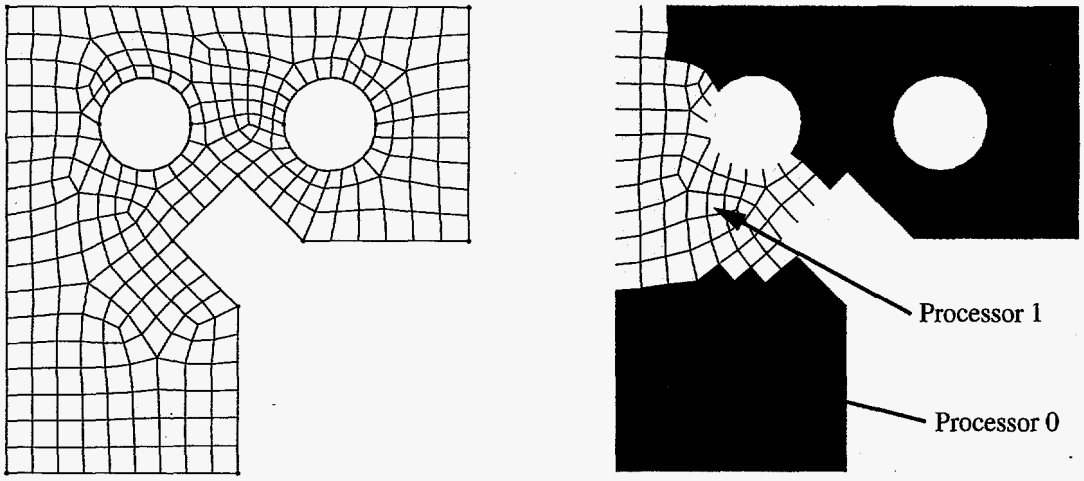


Figure 2-8 Sample coarse mesh and 4 processor decomposition

subdomain files are imported into CUBITP and the mesh-based geometry is constructed (See the right side of Figure 2-9. Vertex, Curve, and Surface reference entities are again denoted but this time are assigned to virtual geometry entities specific to processor 0.).

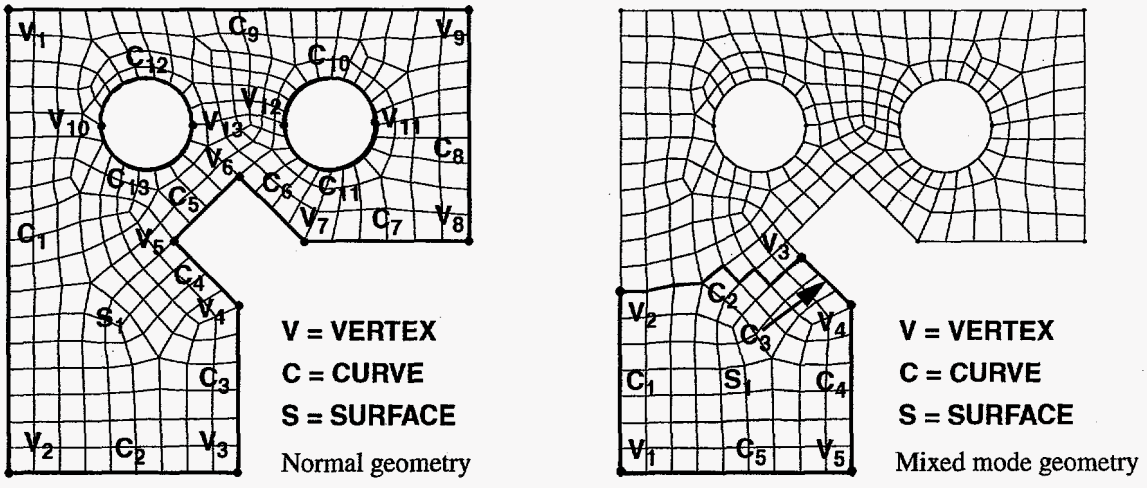


Figure 2-9 Standard reference entities and virtual geometry for processor 0

For virtual geometry, reference vertices are created at points if and only if they meet one of the following conditions: 1) the point is already a vertex in the physical geometry, 2) the point is being used by *more* than two subdomains, or 3) the point is being used by two subdomains and is on a physical geometric boundary. Reference curves are defined between two consecutive reference vertices, and can be shared by no more than two processors. While reference vertices must remain fixed during a given remeshing iteration, reference curves for interior facetted

boundaries can float, or undergo smoothing if necessary. Because these boundaries are defined by mesh entities, and if a smoothing operation can improve the quality of the mesh, then they need not be fixed as their location is arbitrary as long as they are contained within the surface. This relationship is much like a magnet which will remain on a magnetic board, but is free shift around anywhere on the board.

Consider curve 2 (indicated by C_2 on the right portion of Figure 2-9) which divides processors 0 and 1. This curve contains several sharp corners and could, if significant mesh coarsening via a background sizing function occurred, produce mesh edges which seem to shift off of the underlying MeshTopology curve definition (imagine extending just three linear edge segments through curve 2, locating the end points of each segment at one third and two thirds of the arc length of curve 2). This is to be expected, and is exactly the same as what occurs when a typical analytical curve is discretized, yet because facetted data by definition will not be C^1 continuous [34], the difference is simply more pronounced. Succinctly, nodes always lie on the curve while edges may not.

The command used to accomplish this virtual geometry construction is simply the importing of the decomposed subdomain mesh files. In CUBITP, use the *import subdomains* command (see Appendix A.1 on page 55).

2.3.3 Finite Element Model Generation

When the external suite of subdomains is imported, they are matched against the underlying reference entity structure from the initial solid model. The initial solid model has curve and surface topological entities, and the final number of virtual geometric entities created is a function of the intersection of these original reference entities and the newly imported subdomain boundary definitions. Because subdomains are constructed strictly from topology considerations (equalize work load or degrees of freedom while minimizing communication requirements, or region perimeter length), they can and typically do span across element blocks within the model.

While this is not a problem geometrically, CUBITP can only generate a single type of element on a given geometric entity, such that if a subdomain crosses an original reference entity boundary, CUBITP respects that boundary by simply creating one more local subdomain (reference surface for the parallel paving domain) for the local processor. Thus a given processor will have multiple local surfaces to pave after the model is fully loaded into the virtual geometry paradigm.

Once the full set of geometric entities has been constructed by CUBITP, they are available for meshing control specification. The specification of these controls is identical to the specification methods outlined in Reference [2], and consists of assigning a meshing scheme to each surface to be meshed and an indication of typical mesh size. In addition, the existence of a background sizing

function (see Section 1.3 on page 6) from the imported subdomain mesh should be specified at this time.

Following the specification of this information, parallel meshing is initiated in CUBITP with the *mesh subdomains* command (see Appendix A.1 on page 55). This command invokes the parallel mesh tool, which ensures that subdomains can in fact close with all-quadrilateral meshes by guaranteeing even intervals about the subdomains. In addition to the closure requirements, the parallel mesh tool resolves the mesh ownership and generation responsibilities at the subdomain boundaries and ensures that a consistent ordering is generated because the communication requirements need the boundary nodes ordered in the same manner for each subdomain sharing the boundary. For the ParallelMeshTool class header, see Appendix B.4 on page 66.

The balance of the attributes required for a complete finite element model include the specification of element and material type and boundary conditions. The ExodusMesh class in CUBITP imports the previous subdomain mesh and propagates the element block and boundary condition (including communication nodesets and sidesets) settings to the new local subdomain reference entities automatically, so no new specifications need be made in the batch parallel session if the model was specified completely during the coarse model construction. In addition, ExodusMesh propagates the correct non-manifold model structure to the automatically created local subdomains so no merge operations are necessary in parallel. Finally, ExodusMesh manages the construction of the background sizing functions for adaptive paving, although this step has only been demonstrated in the serial operation of CUBITP.

2.3.4 Model Export

To complete the construction of a finite element model with parallel paving, the model must be exported back into the suite of decomposed processor files with a refined mesh but identical element block, boundary condition, and communication nodeset and sideset settings. The GenesisTool class in CUBITP manages these issues and ensures that the refined node and element number maps are consistent such that analysis codes and EDCT can properly reconstitute the parallel model.

To perform this concatenation, nodal ids (stored in the node number map) must be globally unique, and element ids must be globally unique and in the same order of element block ids globally. Essentially this requires all the elements in block 1 to be globally unique and less than the lowest id of the first element of block 2, etc. GenesisTool ensures these requirements are met and uses the ParallelMeshTool class to calculate the global offsets in each case.

The user command to perform these operations and write out the suite of refined mesh data files is the *export subdomains* command in CUBITP (see Appendix A.1 on page 55).

2.4 Communication Requirements

One of the attractive features of the constant area parallel paving method is the limited amount of communication needed to exercise this technique. The communication that is required will be discussed in this section.

There are only three significant communication requirements of the constant area paver, and a fourth that could potentially require communication but is as of yet uninvestigated. The three main communications are: 1) boundary ownership negotiation and exchange of subdomain boundary curve mesh data, 2) exchange of boundary curve mesh sizing data, and 3) global numbering calculations for model exporting. The fourth potential communication requirement is the inter-processor smoothing of the interior processor boundaries. To perform this task, both processors have to know something about their neighbors' mesh topology, even if it is relegated to the simple communication of the boundary adjacent nodes and elements. This is still unknown at this point, but in the event this smoothing capability is developed, some communication requirements will exist for this task.

Currently, the boundary ownership negotiation is quite simple in that the processor with the lowest id is defined to be the owner of a shared boundary. An advantage of this elementary test is that it is deterministic at each processor without any communication since each processor knows the id of each of its neighbors. When a more appropriate test is devised, this current method can be easily modified. Thus the first communication area really involves packing and sending the mesh data (for owner processors), and receiving and unpacking the mesh data (for non-owners). The nodes that the boundary owner creates are packed (coordinate triplets only) into an array which is sent to the non-owner partner for the same boundary, where they are unpacked and instantiated once again. The connecting element edges are simply inferred from node order. The exchange of sizing data between at inter-processor boundaries is required to ensure that sufficient data resides with each processor to employ the sizing function along its boundaries (see Section 3.4 on page 38). The global numbering calculations are just the global calculation of offset id values for each processor for both nodes and elements, with a little more work performed for the elements because they must be globally unique and bounded within a range for a given element block.

These communication tasks are managed by the CommunicationTool, a class modeled after the singleton design pattern (see Appendix C on page 69). The singleton method ensures that the CommunicationTool is accessible at all times and from any location within CUBITP, but that there is always one and only one instantiated object of the tool per process. This guarantees that there is only one path to communicate with the set of active processors (in the MPI standard, this would be the MPI_COMM_WORLD group of processors).

Additionally, the class is abstracted such that one of several variants of a communication standard can be selected at run time, and only the selected version will be instantiated. The current

supported communication types are MPI [38] (Message Passing Interface), and no or null communication. This class design could easily be extended to include PVM (Parallel Virtual Machine) or other standards as needed, an important consideration for future extensibility. The implementation of these concrete instantiations was significantly aided by the nodes class from the ALEGRA [37] project.

3. Parallel Adaptive Meshing

The primary motivation behind extending the paving algorithm into parallel environments is to take advantage of its adaptive meshing capability for parallel analyses. To this end, this chapter addresses the overall adaptive analysis loop design which contains the parallel paving code, dynamic load balancer for parallel efficiency retention, an appropriate parallel analysis code, and an error estimation code for adaptive function generation. These components must be used in concert to support an environment capable of performing parallel adaptive analysis. In addition, the chapter will examine the dynamic load balancer to be used with parallel paving, and the mechanism needed to preserve topological data between loop components and hence remeshing iterations.

3.1 Adaptive Loop Overview

An adaptive loop for parallel analysis will contain multiple components, and the relationships between these components will affect their efficiency and viability. The significant steps of this loop are as follows:

Initialization portion:

1. Generate initial mesh and construct complete finite element model with geometric topology data
2. Decompose initial model into <n> files and load parallel machine
3. Initiate parallel mesher and construct necessary geometry for mesh refinement
4. Import background sizing function for adaptive mesher

Repeated portion:

5. Perform subdomain boundary negotiation, discretization, and data exchange
6. Perform subdomain meshing (adaptive if sizing function is available)
7. Re-map variables from previous mesh to current
8. Dynamically load balance current mesh for efficiency
9. Perform finite element analysis
10. Perform convergence test (met accuracy criteria? if yes, exit, else, continue)
11. Generate new background sizing function over problem domain (error estimator)
12. Go to step 5

This loop represents a parallel, adaptive analysis which contains an adaptive meshing component. Some assumptions can be made to simplify the loop, such as only considering remesh/restart (time independent) types of problems for now. This assumption would eliminate step 7, a difficult issue which is not yet handled in general. Another assumption may be to attempt to link together some combination of the parallel mesher, dynamic load balancer, and the analysis code. Were this possible and practical, a diskless data exchange mechanism could be devised to avoid writing to and reading from the disk each time a component must pass its data to the next task. However, if memory space is tight, the components can communicate their contributions to the next

task by writing to a common file format, such as ExodusII [15], although the disk writing and reading process will eventually prove to be a bottleneck. Other assumptions might include additional load rebalancings to gain even more efficiency on the balancing process itself. While this loop was not fully exercised due to resource limitations, it is an anticipated eventual use of the parallel paving capability in the near future.

3.2 Dynamic Load Balancing

Because adapting grids will redistribute degrees of freedom (mesh entities), the previous decomposition will normally become insufficient with time. Rebalancing the domains dynamically is key to achieving a successful adapted solution.

3.2.1 Overview of the "tiling" Load Balancer

The dynamic load balancer does a global load balance by means of a series of steps of local load balancing [3]. This algorithm was developed for adaptive order (p-) refinement finite element methods where the number of finite elements remains constant and the work per element varies according to the amount of refinement, while we are applying it here to an adaptive scheme where the number of elements changes. The original application would do one step of the dynamic load balancing after a given number of steps of the algorithm because the refinement could also change over time. With this application, because we do a complete calculation with the same grid to calculate the error estimation, and then do a large refinement of the size of the elements, we take several steps of the load balancer on the new grid until we converge to a balance. Because only local communication is involved for each step, the steps are very fast. Therefore, achieving a reasonable load balance is much faster with this method than for a global load balancer.

At each step of the local load balancer, there are four operations. The first is to determine the work loads on the processors. We have defined the work load as the number of elements on a processor. The definition of work load for the original application of the algorithm was the time that it took a processor to process its local data in a single time step. Because all of our elements have the same order, and we change the number of them, we use the number of elements for the work load.

The second operation is to determine the processor work requests. Each processor compares its work load to that of the other processors that it borders. If any of these processors have a work load greater than it does, then it sends a request for work to the processor which it borders which has the greatest amount of work.

The third operation is for those processors which received requests for work to determine which elements to send to the processors which have requested work. The elements are picked so that those elements which are neighbors of those assigned to the processor requesting work are

more likely to be sent. This helps the set of elements that a processor has to remain more geographically compact.

The fourth operation is to transfer the elements and to notify the other processors involved. If a given element is transferred from one processor to another, then all of the other processors which have elements which are connected to that element have to be notified where the element is going to be.

We start our adaptive loop with a grid which has previously been load balanced with a serial load balancer (see Section 2.3.1 on page 25). If the part of the grid which a processor has is a reasonably compact, connected set of elements, then it enters the load balance phase with a set of elements which are connected in the new grid. With these initial conditions, this load balance scheme tends to produce a connected, somewhat compact group of elements on the processor at the end of the load balancing operation. We tested this algorithm with starting meshes in which the elements assigned to a processor where essentially random, and it managed to produce partitioned meshes where most of the processors had one connected group of elements with few undesired features. Figure 3-1 shows a similar process in which the initial mesh contained contiguous groups

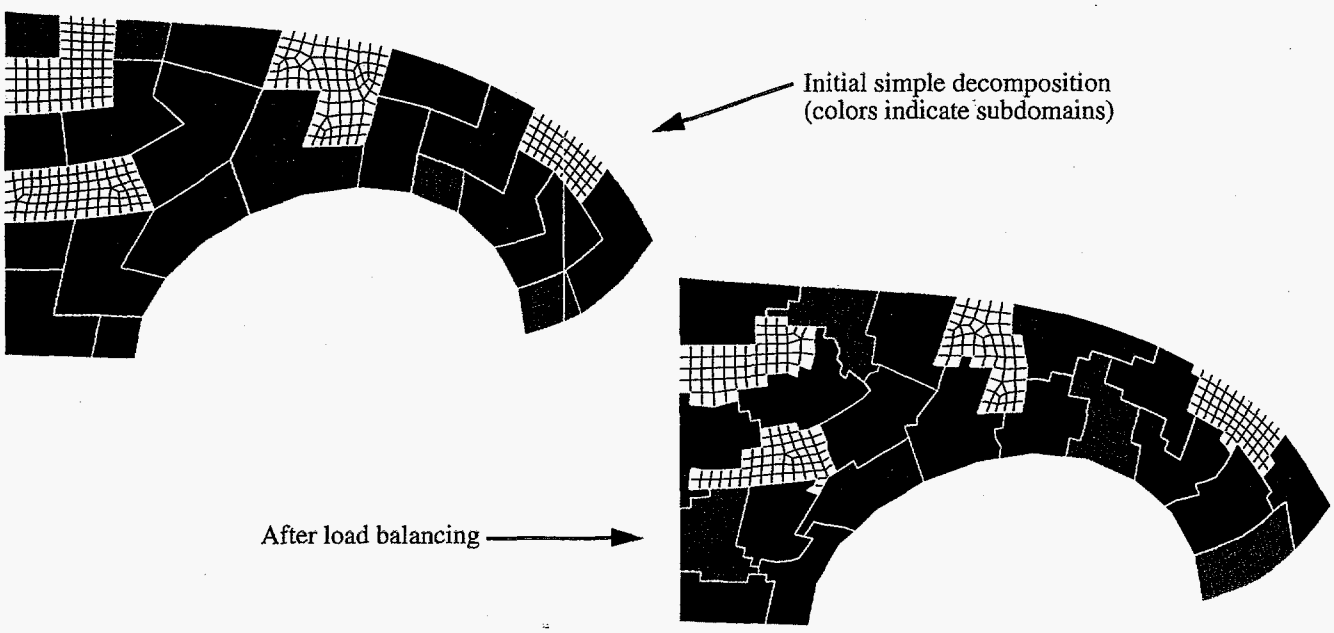


Figure 3-1 Load balancer test for randomly grouped element sets as input

of elements, not optimized but randomly arranged.

To use this with ExodusII data, we wrote a converter which converted from this data format to and from ExodusII. This translator read in a separate ExodusII file for each processor similar to

those that EDDT would produce. The data structures in the algorithm have a pointer (for each element) to storage for the information associated with that element. We put the information for each element along with the element's nodal connectivity information and the information for each of the element's nodes in that storage area. Therefore, to convert from this format to ExodusII, we reconstruct the element block information and the nodal arrays from this element associated information. We also reconstruct the node sets and side sets used for interprocessor communication from this data. The process for converting from ExodusII to this format is similar.

3.3 Geometric Topology Preservation

A modern mesh generator requires certain topological information to unambiguously represent geometric models. While this data is created automatically when geometry is created within the mesher, if existing mesh models are imported into the mesher from abbreviated model definitions (ExodusII [15]), the topology information which was available on the original geometric model is no longer present.

To resolve this issue, a method of grouping mesh entities (nodes) into sets which belong to geometric entities has been designed. The groupings are defined when the model is first constructed. Subsequent importing of this mesh model complete with the geometric groupings allows the mesher to reconstruct a valid solid model (deformed if necessary [22]), and to recreate the non-manifold reference entity structure (discussed in Section 2.2.2 on page 20).

Nodes which belong to a given geometric entity are assigned a nodeset with a particular id and label for use when the ExodusII data file is read back into the meshing code. A node contained within a RefVertex entity is placed in a nodeset with an id of 50,000 + the id number of the RefVertex. Nodes which are belong to a RefEdge (including its RefVertex entities) are assigned to a nodeset with an id of 40,000 + the id number of the RefEdge. Similarly, nodes which are owned by a RefFace and its lower order entities are put into a nodeset with id 30,000 + the id number of the RefFace. Once each nodeset is created, the data type of the reference entity and its numerical id are put into a string data variable and stored as a property table data entry for that nodeset.

While far from ideal or efficient, this method allows the meshing code to recreate non-manifold finite element models from an ExodusII database, an impossible task before this method was employed. In addition, extensions have been made to the EDDT decomposition code and the PRONTO 3D [5], Coyote [16], and GOMA [17] analysis codes to pass through the needed property table data where the topological data is stored.

As an example of this topology record, consider the model in Figure 3-2.This example problem models the melting front of a multi-component material in three phases (solid, liquid, and mushy zone). Using this method of tracking the phase fronts, the geometry can be deformed during the analysis, and can be imported back into CUBIT and have new, deformed geometry created

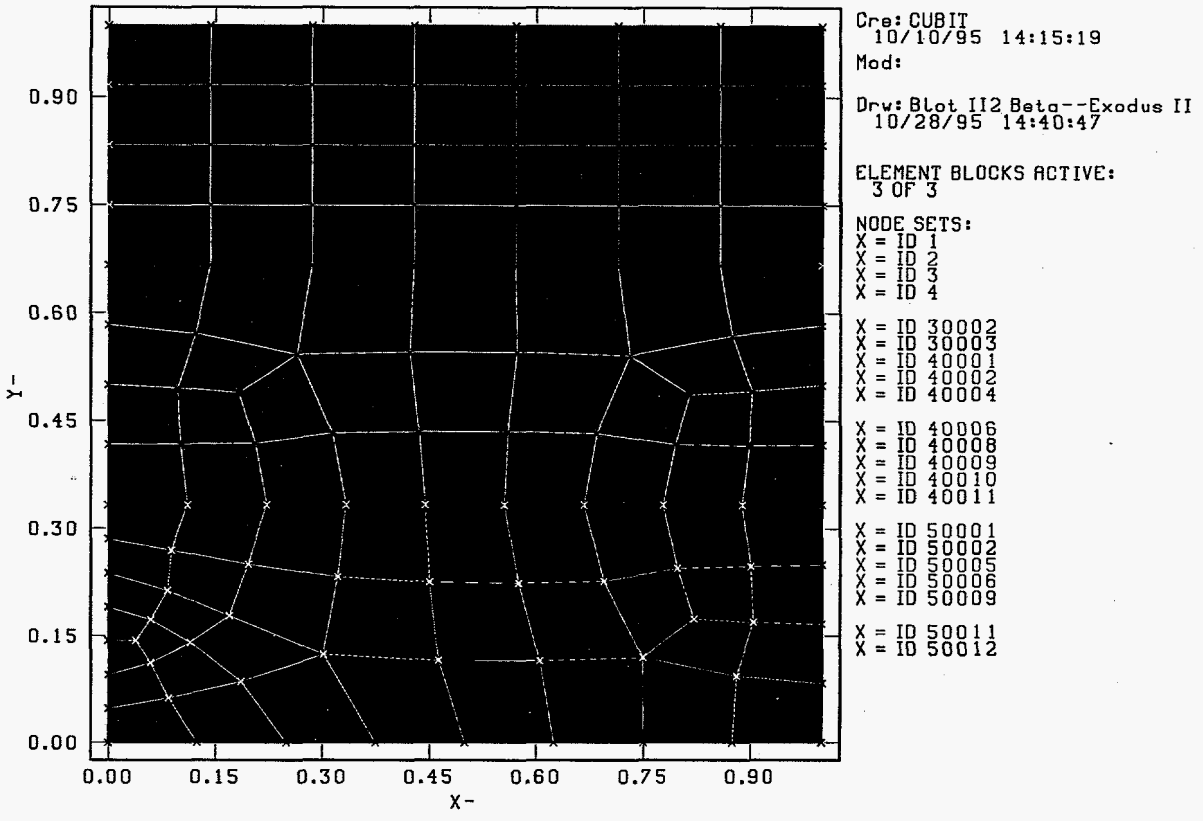


Figure 3-2 Initial geometry for 3 phase material melting example - GOMA

using the previously existing non-manifold reference entity structure for topology. This means of topology preservation is required for any task in which previous mesh data is to be imported back into the mesh generator and matched with some type of geometric model, whether the model is being created at the time from deformed mesh data, or if it is simply the original, undeformed solid that existed within the mesher when the mesh was originally generated. Figure 3-3 displays the subsequent deformed original model and the remeshed version with new, deformed geometry.

This method of topology preservation has some limitations. The reading and writing of large numbers of string data types is, or seems to be very inefficient in ExodusII. One model attempted recently contained approximately 4000 curves, not insignificant but certainly not intractable. The time required to *generate* the nodesets required to store the models topology approached 1 hour on a workstation class machine. Reading the decomposed mesh regions back into CUBITP took just as long, making the problem quite difficult to fine tune or even complete. While the issue of file formats is not in the scope of the parallel paving project, it is painfully clear that significant improvements must be made to begin to seriously attempt to do any kind of production adaptive remeshing and analysis.

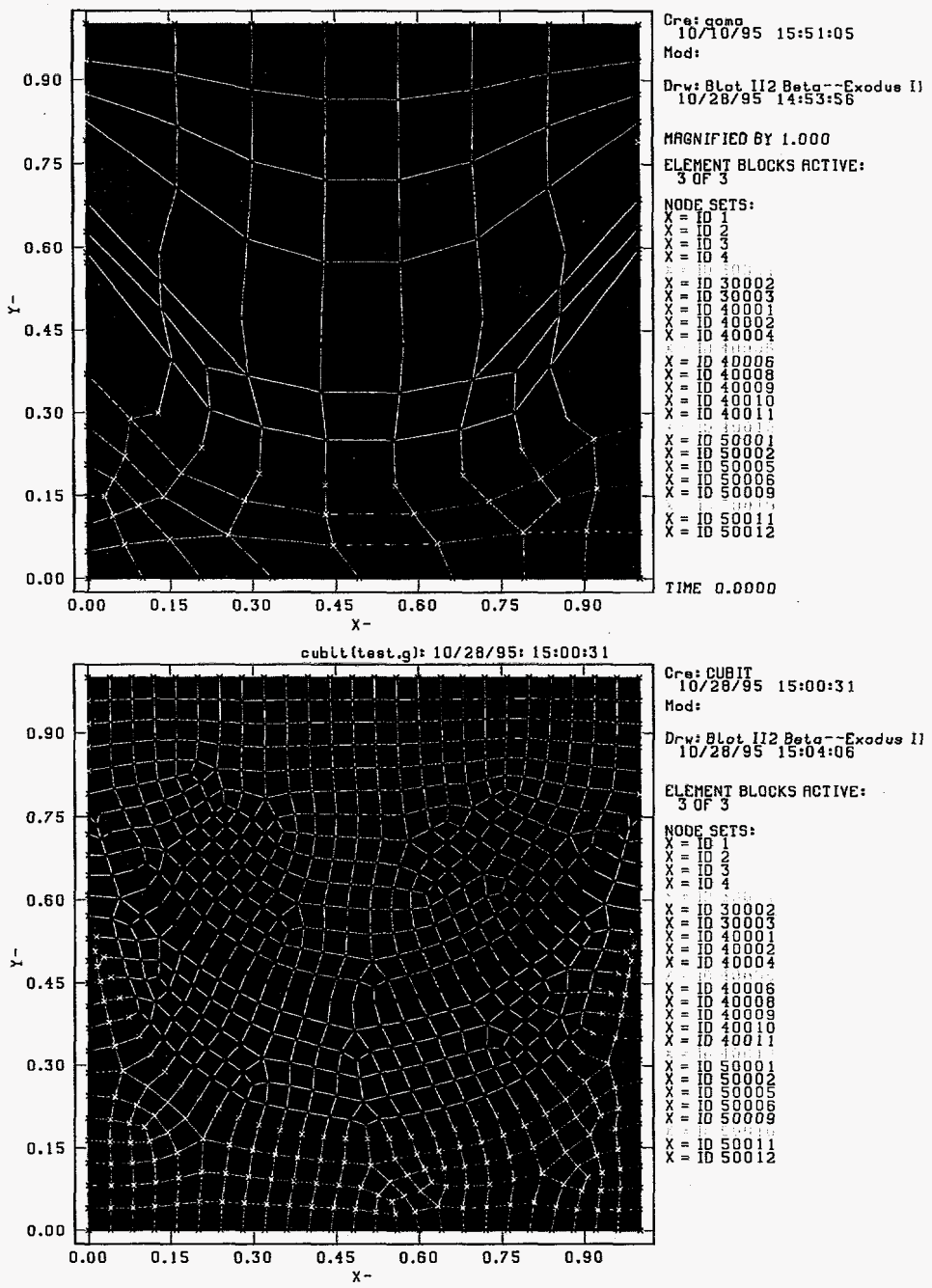


Figure 3-3 Deformed initial mesh (top) and new remeshed region (below) - GOMA

3.4 Parallel Use of Adaptive Functions

Aside from the need to store and reuse topology data, additional issues remain for the use of background sizing functions for meshing with parallel paving. Two types of analysis output

variables can be used to provide sizing information: nodal and element based. Nodal variables (such as temperature or displacement) contain discreet values at each node point of the previous mesh, and new point evaluations along a boundary between two subdomains can be determined explicitly with only the two neighboring subdomains. This characteristic of nodal variables is desirable, because the communication needs of these evaluations for a parallel computation are limited to the two neighboring domains. Element based variables must be extended to the node locations via a standard Laplacian interpolation. The value of the function at a node, n, is simply the summation of all the contributions from its surrounding elements divided by the number of surrounding elements. Due to this interpolation, this type of variable requires more information in the neighborhood of a vertex (to produce an equivalent field of nodal function values - see Figure 3-4) at which more than two subdomains meet. In other words, this is a vertex within the interior

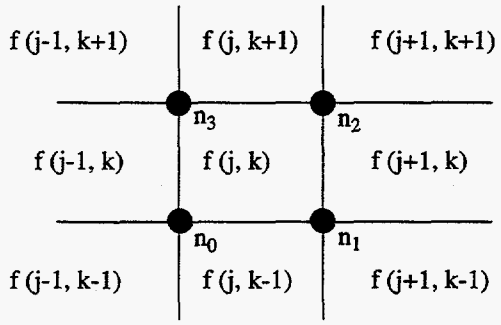


Figure 3-4 Interpolating element function values to nodal locations

of a decomposed model, not touching any physical geometric boundaries. Because the function value at the vertex requires contributions from each of the neighboring elements, each adjacent subdomain must communicate its corner element's contribution at this location. It is then important to minimize the number of subdomains which meet at a vertex for analyses which use element based variable functions. Once the nodal function values are available, function size requests are computed using a bilinear interpolation technique from the nodal values residing on the previous mesh nodal locations. This interpolation works well [28] for the resolution range of the paving algorithm, even though the gradient of the interpolated function changes discontinuously at the boundaries of each element of the previous mesh. Other higher order interpolations could be pursued (such as bicubic interpolation) which would alleviate the discontinuity, but at a higher computational cost.

4. Numerical Experiments

This chapter provides the numerical result data and discusses the different environments used for testing the constant area parallel paver. Two geometries are presented as examples for performance and mesh quality assessments regarding parallel paving.

4.1 TCP/IP Network Parallel Implementation

The initial parallel version of the CUBITP code was built and tested on a network of workstations sharing the same operating system version, but with varied CPU capabilities, ranging from Sun sparc10 models to 125 and 150 MHz sparc20 platforms. The workstation environment was preferred for the main construction phase of the project due to the superior debugging tools and the readily available graphical display capability, a critical need for a meshing algorithm development effort.

The *mpich* [38] message passing facility was used for the Sun implementation, and several of the mpi access scripts were modified to facilitate a productive debugging environment. A limited graphics capability was enabled for parallel meshing (Sun implementation only) allowing up to 8 processes to maintain separate graphics windows on the same X window display.

For code development, the advantages of debugging tools and graphical displays were more important than the substandard performance incurred during parallel processing over TCP/IP networked nodes. These CPUs were not dedicated either, so multiple processes could be competing for the limited number of compute cycles available in this environment. These shortcomings were overcome once the code was ported to the IBM SP2 parallel machine (8 processor architecture).

4.2 IBM SP2 Parallel Implementation

The final version of the parallel paving code was ported to AIX on an 8-node IBM SP2 machine. The required solid modeler ACIS libraries were built along with the meshing code. No graphics are supported on this performance designed version of the code. A high speed communications switch was employed on this machine, and the resident parallel job manager (POE [39] - parallel operating environment) assigned dedicated nodes for parallel runs in this environment.

Performance significantly improved in this environment, while debugging became more challenging. The parallel debugger **pdbx** worked adequately, but the graphical debugger **xpdbx** refused to load our entire code before crashing. This seemed peculiar, because it performed so well with the example code provided during the introduction to parallel programming course presented by IBM representatives.

The parallel job manager, POE, was well organized though and helped significantly to reduce the time needed to ensure the right switches were set for maximum performance gains during parallel operation. Several options were available such as the ability to run the CPU nodes in a dedicated or shared mode and to use the high speed communication switch or to employ TCP/IP packets for communicating.

4.3 Performance Assessments

Performance data is presented for two geometries under two parallel testing environments: first, a Solaris workstation network using TCP/IP, and secondly, an IBM SP2 running dedicated compute nodes.

Performance data is presented in two forms: first as a function of MPI_Wtime vs. problem size, and secondly as efficiency vs. problem size. MPI_Wtime is an MPI [38] function which returns the number of seconds since some arbitrary point in time in the past. The point is guaranteed not to change during the lifetime of the process, therefore successive calls to MPI_Wtime can be used to compute process duration time. This computed duration time in machine seconds is used in our results. Speedup and efficiency are defined in Equation (3) and Equation (4) as follows:

$$\text{speed\_up} = (\text{time for 1 processor})/(\text{time for n processors}) \quad (3)$$

$$\text{efficiency} = (\text{speed\_up} * 100\%)/n \quad (4)$$

where time for a single processor is calculated as the process duration with MPI_Wtime described above, and time for n processors is calculated as the maximum process duration over all the processes in the parallel run (one process for each processor) in Equation (3). In Equation (4), n is the total number of processors. Thus an efficiency of 100% would indicate a linear speed up, and greater that 100% efficiencies indicate super-linear speed up.

Another important consideration is the issue of competing effects in the computation. Even if results seem to indicate linear or super-linear speedups, it is critical to understand the combination of factors which is responsible. Reference [40] encapsulates this idea into a "parallel effectiveness" quantity which factors in efficiency with other quantities. Our results can be understood more appropriately with an effectiveness quantity such as this as opposed to a true "efficiency", since several effects are present in our data. This will be discussed more in Section 4.3.4 on page 46.

4.3.1 Example Problem Definitions

Two problems will be investigated for paving performance in the parallel environments. The spline geometry was constructed to meet a need for simple geometry that was not so trivial that it would have symmetric meshes generated. The spline problem is treated as single material with four node quadrilateral, uniformly-sized, two dimensional elements. Figure 4-1 shows a representation

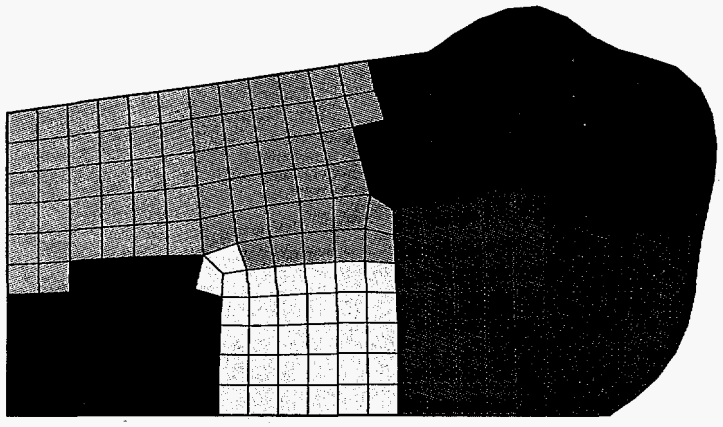


Figure 4-1 Eight processor decompositions of spline coarse mesh

of the partitioned planar spline geometry for an 8 processor decomposition. Decompositions for 2 and 4 processor runs were also constructed.The mesh in this figure is the initial coarse background mesh for the spline geometry.

The second problem is a nose cone model with a three dimensional surface that was investigated for generating shell models with parallel paving. This model contains four separate

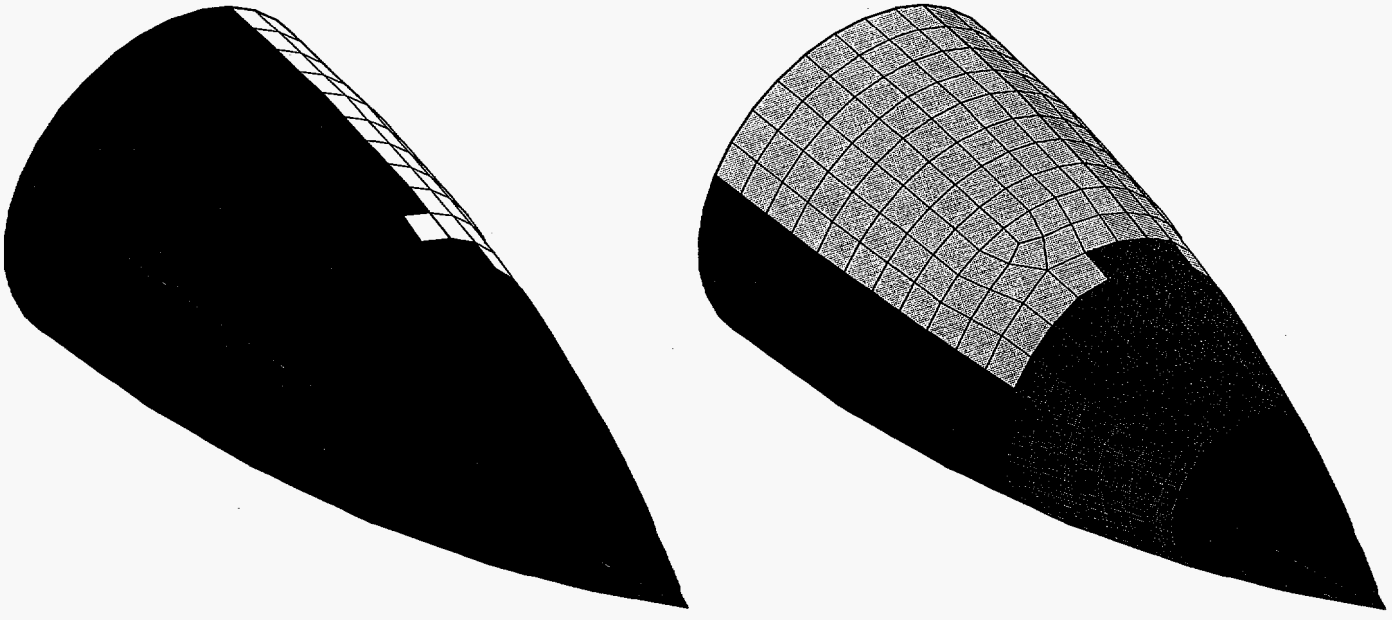


Figure 4-2 Four and eight node decomposition of nose cone geometry

materials, with four node, variably sized, three dimensional shell elements. Figure 4-2 shows the four and eight processor decomposition of the nose cone geometry. A two processor decomposition

was also performed for this geometry. The displayed mesh is the initial coarse background mesh for the nose cone.

4.3.2 Performance on a TCP/IP Network

Within the TCP/IP environment, several issues must be clarified prior to the understanding and interpretation of numerical results. First, the only hardware which could be dedicated for parallel performance measurements in this environment was a dual CPU sparc20 platform (two dedicated HyperSparc 150 MHz chips). Numerous experiments were run across the network computing resources, yet because the non-local desktop CPUs were not dedicated or of equal speed, the standard deviation of subsequent runs using this combination of platforms could be quite significant. This environment was capable enough to allow the parallel code development to progress and to also provide a useful scoping study into parallel performance trends.

To perform reproducible results within the TCP/IP environment, all the recorded performance measurements were tested exclusively on the two dedicated 150 sparc20 chips on the local desktop machine. Even so, it is important to realize that with only two CPUs available, a serial (single node) run would enjoy 100% of one CPU, the other remaining idle, while a 4 node run would at best achieve four separate 50% CPU resource processes, and similarly for 8 and 16 node runs. Therefore, absolute performance comparisons between runs of the TCP/IP data are not relevant, but trends can be readily identified. For the same reason, speedup and hence efficiency plots will only be generated for the IBM SP2 runs, since the TCP/IP data cannot be compared absolutely between runs with different numbers of processors.

For the performance data under the TCP/IP implementation, Figure 4-3 displays the timing plots for conventional (serial) paving and 4, 8, and 16 processor parallel paving for the planar spline model. From the timing data observed for the serial run in Figure 4-3, a quadratic growth pattern is discernible. The timing curves for the parallel run in Figure 4-3 were more reasonable, but if taken out far enough, will continue to trend upward following the conventional paver because they are still the same algorithm intrinsically. Conclusions regarding this growth pattern will be discussed in Section 4.3.4 on page 46. The small advantage the parallel approach has for this problem is that because the effective size of the problem is reduced via decomposition, the growth of the timing curve is shifted back to the left for parallel runs. This is only a temporary reprieve from the onset of the eventual problem, however, and this observation should be used to investigate the conventional paver carefully to address the scaling problem at its source. Since there were only two CPUs available for the TCP/IP runs, no speedup plots will be presented here because they would be meaningless, as mentioned earlier (comparing 100% CPUs to 50% and 12.5% CPUs).

Figure 4-4 shows the same plots for the nose cone model under TCP/IP Again, there is a second order growth trend in the serial version, although Figure 4-4 appears linear. The trend emerges more slowly for this geometry since the overhead (non paving work) for managing many

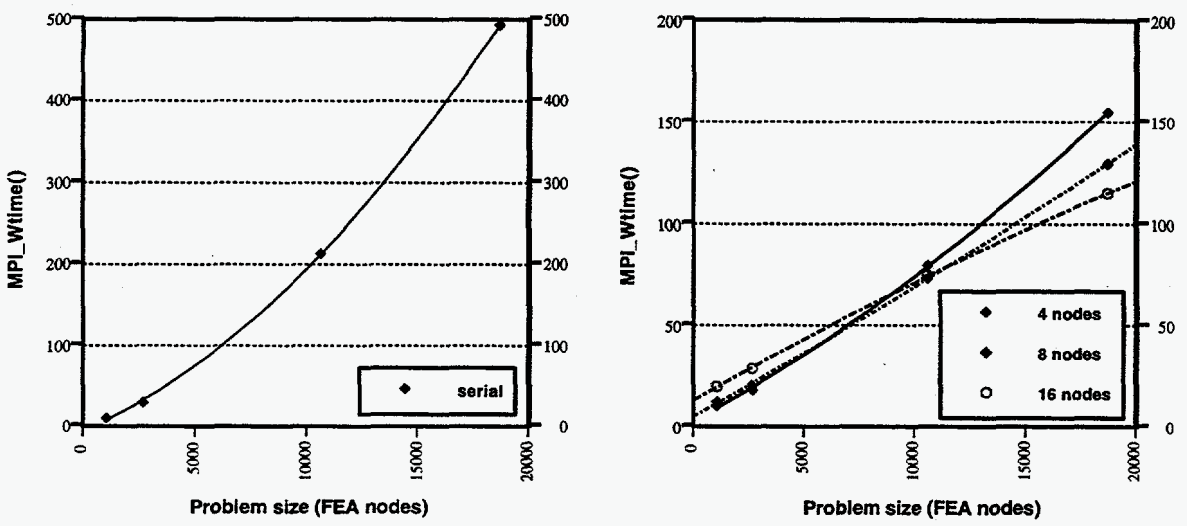


Figure 4-3 Two dimensional spline timing plot for serial and parallel runs - TCP/IP

more surfaces scales in a more linear fashion. If larger problems could be run for this serial case to ensure the paving portion of the work load dominated, the quadratic growth would be more apparent. Machine limitations constrained any larger nose cone models for the TCP/IP serial case.

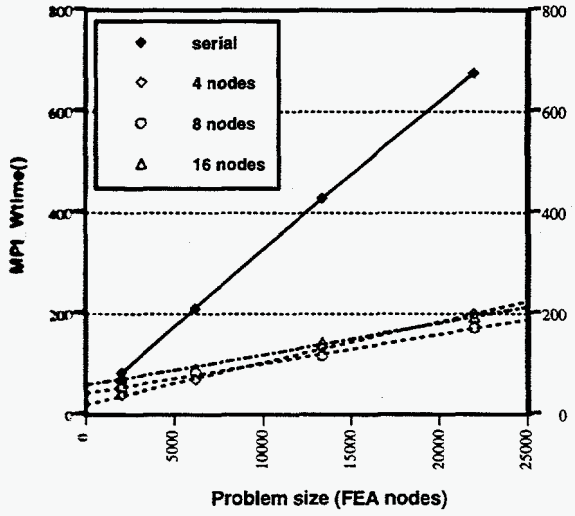


Figure 4-4 Nose cone timing plot for serial and parallel runs - TCP/IP

This growth trend exists as well within the parallel runs, just a bit more subtle yet.. Note that the curves on this plot should not be compared to each other due to the CPU mismatch issue, or in other words, the actual y-intersect positions would vary from Figure 4-4 had all 4, 8, and 16 nodes been

of equal capability from run to run. The IBM SP2 comparisons do have compute nodes of equal capability and thus are appropriate for these comparisons.

4.3.3 Performance on an IBM SP2

IBM SP2 performance data for the two dimensional spline problem is displayed in Figure 4-5. Again the second order growth trend is visible, and the apparent super-linear speedups are a

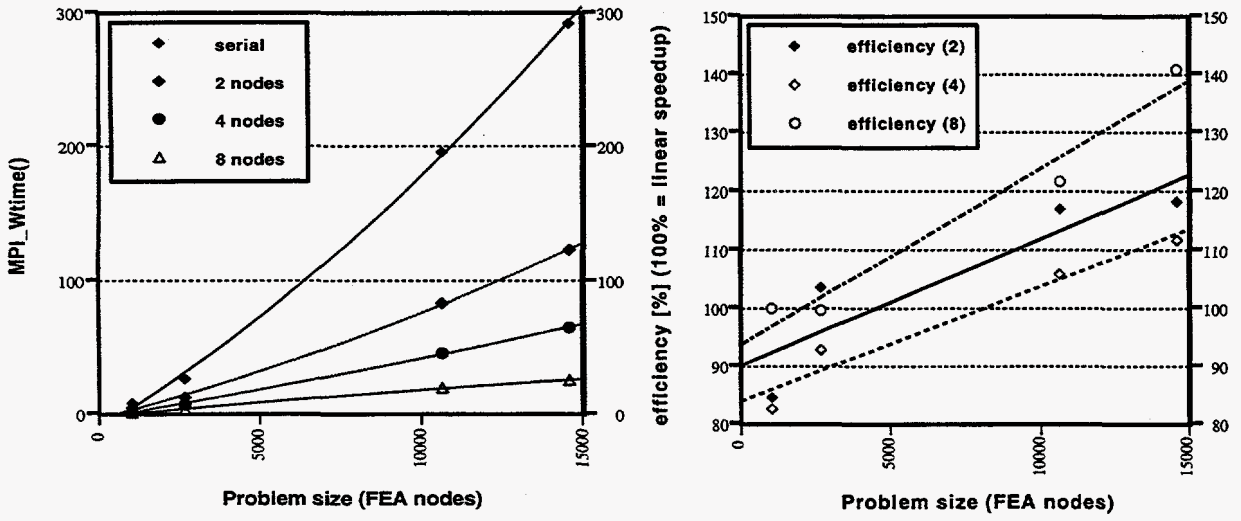


Figure 4-5 Two dimensional spline timing plot for serial and parallel runs - SP2

beneficial result of the effect the decomposition has on parallel paving's performance. This will be discussed in more detail in the following section. Similarly, nose cone plots are seen in Figure 4-6. With this geometry the timings were not quite so exciting, although as more processors were added, things tended to improve. As a result of the noise present in the smaller models, the efficiency calculation does not show a clear trend in this case. One factor that probably affected the nose cone geometry with an increased overhead load is the issue of many local subdomains being managed per processor. Because the number of surfaces on the nose cone was already significant, then increased once the decomposition was made, each processor has to sort through all the surfaces, mesh one, find the next, mesh, etc. With the spline example, the entire timing study was able to focus completely on the operation of the parallel paver with no intruding overhead tasks taking up cycles.

4.3.4 Performance Interpretation

Several competing effects must be understand to properly assess the data on parallel paving. Three effects will be described: 1) divide and conquer, a beneficial effect, 2) general parallel overhead, a negative effect, and 3) problem change effect, nature unknown at this time.

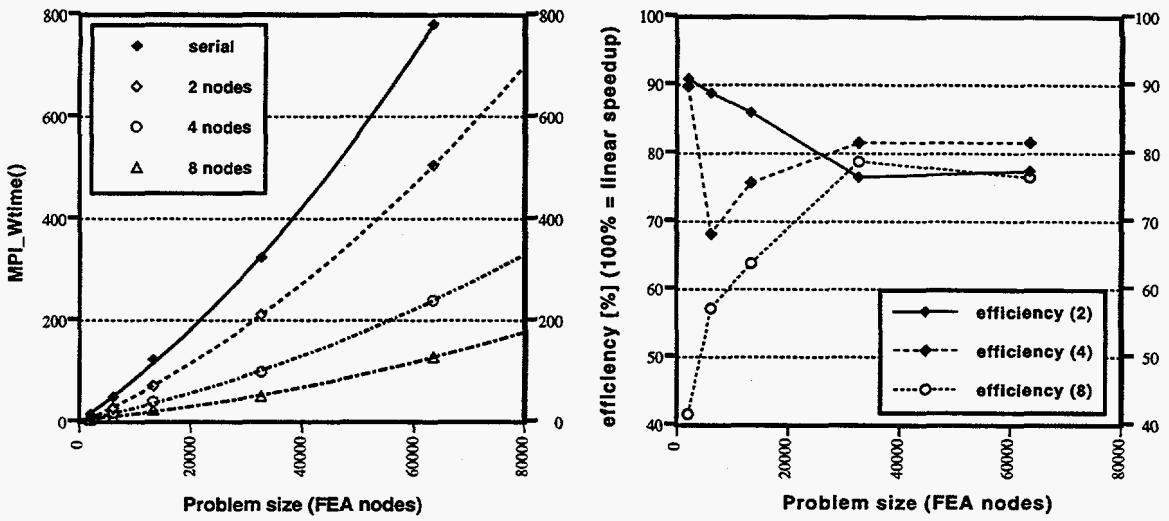


Figure 4-6 Nose cone timing plot for serial and parallel runs - SP2

Divide and conquer is by far the most powerful effect, and since it is a beneficial effect resulting from the mere decomposition of the domain, it is the cause behind the data which suggests super-linear speedup has occurred. The limitations of the conventional paving algorithm have been known for some time, principally that paving is very inefficient on large problems. This timing curve indicates a scaling problem with the conventional paver, a characteristic that is consistent with the serial paver's intersection detection scheme. At the time of this writing, conventional paving performed intersection checking for each newly created element against every other element edge on the front instead of in a local area. The result of this is a performance problem as the compute requirements for conventional paving increase on the order of n^2 with a problem size of n finite element nodes. This characteristic results in the observed quadratic growth patterns in the paving timing curves in Figure 4-3 through Figure 4-6. A natural solution to this is the use of a bin sorting technique to eliminate intersection checking other than in the local area of a new element or node placement. Thus the end result of the divide and conquer effect is a speedup that appears super-linear, not from a fortuitous algorithm implementation, but simply from the application of a decomposition to the problem domain.

The effect of parallel overhead is an inescapable effect which simply states that given a perfectly balanced and 100% efficient process, the actual speedup will be slightly less than 1.0 from the small overhead that even minimal communication will incur. This effect is more than counterbalanced by the divide and conquer effect for parallel paving.

The final effect is the issue of problem change, which has influence but should not be especially significant for these problems. Note that paving, either conventional or parallel is a non-deterministic algorithm. While it will consistently deliver high quality meshes, it can produce

different meshes depending on which curve is selected to begin laying in elements for meshing a region, at which point the process becomes asymmetrical.

Because of this, the normal idea of scaling and efficiency calculations for parallel performance are more challenging to assess because paving cannot operate within another grid or topology, and cannot be induced to generate a specific number of elements. Thus when the graphs show that the number of processors was increased and the model size increased, the plotted number of mesh nodes is not exact but is a close estimate to the true number of finite element nodes present. These estimates have always been within two percent or less of the actual total finite element nodes for these examples, such that the measurements for efficiency are quite close to the true efficiency values if they could be measured. For this reason also, we cannot strictly state that super-linearity was achieved since the technical definition of speedup was not completely adhered to.

These effects will all factor together to shape the data trends which are seen in the parallel paving results. As noted previously, the plots for efficiency would be more appropriately presented as "parallel effectivity" plots, considering the number of effects being combined.

In summary, one practical way of assessing the parallel paver's performance is to simply observe a generation rate of nodes for given problem sizes and decompositions. Figure 4-7 shows this comparison, where most of the data is quite logarithmic and the rate is increasing regularly with the number of processors.

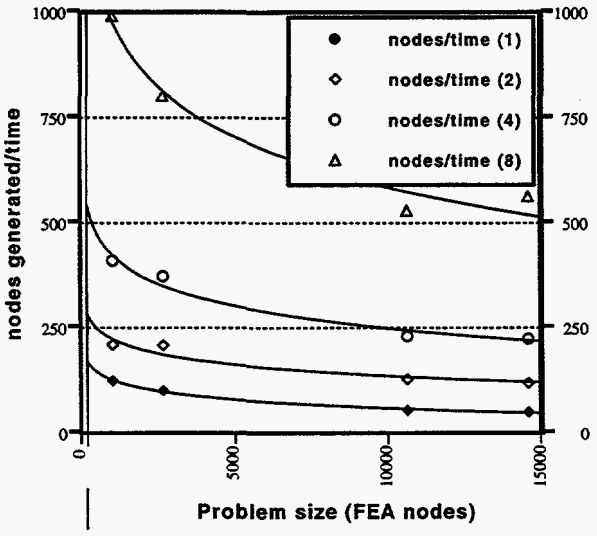


Figure 4-7 Node generation rate for two dimensional spline - SP2

4.4 Mesh Quality Assessments

Mesh quality statistics were measured for the spline body using the NUMBERS [41] code. NUMBERS computes element statistics and shape data, and uses the techniques from

Reference [42] to attain the average, minimum, and maximum aspect ratio, skew, and taper. Figure 4-8 shows a distribution of the average aspect ratio for the spline body on the SP2 machine. The

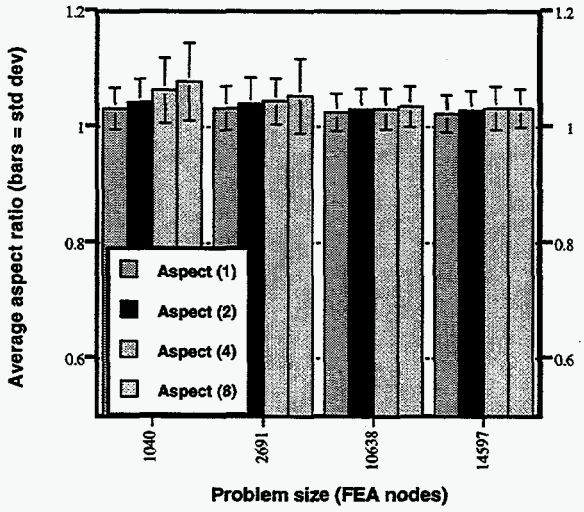


Figure 4-8 Aspect ratio distributions for spline body - SP2

label "Aspect (1)" indicates aspect ratio as calculated from a mesh generated with a single processor, "Aspect (2)" refers to a 2 processor run, and similarly for the remaining values.

The error bars on Figure 4-8 are the standard deviations for average aspect ratio as calculated by NUMBERS. Skew and taper statistics were also collected, but the standard deviation on all of these was larger than the average values indicating data that is simply random background noise, thus no insight is to be gained with it other than that it did not significantly change regardless of how many processors were used. In Figure 4-8, notice that the standard deviation is the largest for relatively coarse and highly decomposed models. This is to be expected because paving is being asked to generate elements in smaller regions (result of the geometric artifacts being introduced by the virtual geometry boundaries in the center of the mesh) at coarse mesh sizes, which is a combination that will always result in more deformed elements since paving has less flexibility in a small coarse mesh to finely control the element quality. As the mesh density increases, the standard deviation decreases as more and more of the elements are being created in the interior of the geometry, allowing more freedom for smoothing and placing well shaped elements.

In closing, Figure 4-9 shows a snapshot of the nose cone after it has been refined to approximately 13375 nodes.

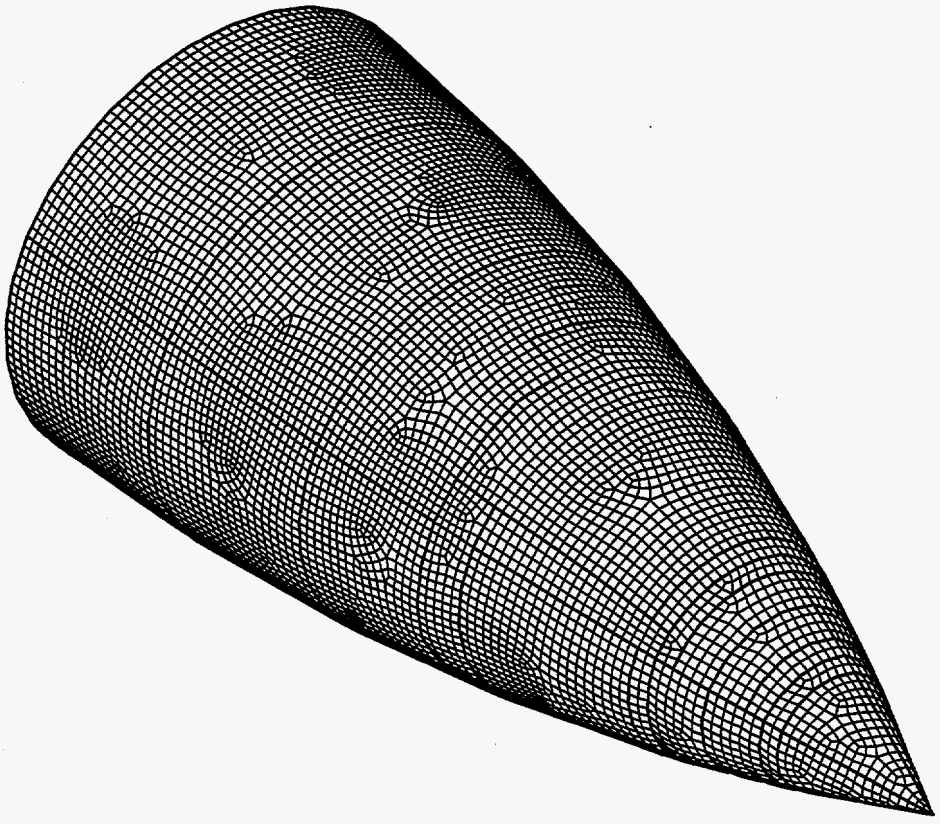


Figure 4-9 Nose cone final mesh - 13375 nodes

5. Conclusions and Remaining Issues

Parallel computing is required to achieve large scale adaptive analysis capability, an extremely compute-intensive operation. Some of the difficult bottlenecks in performing this analysis in a parallel environment include generating a very large mesh model and distributing the model over the compute nodes, and then regenerating and redistributing the model each time the analysis "adapts". To be successful, the parallel methods must accomplish this and compare favorably with their serial counterparts for performance and quality. This research addressed these issues by prototyping a parallel version of the paving meshing algorithm.

A first conclusion is that the paving algorithm itself can be used serially to generate a coarse background mesh which provides a topology to be decomposed onto the parallel domain. Since the paver naturally transitions and produces well shaped elements, the background coarse meshes produced by paving yield very well formed topologies for use in parallel operation after decomposition. The final dense mesh is then created in parallel by paving each subdomain independently once the subdomain boundary discretizations have been negotiated between adjacent subdomains. This process was exercised successfully with numerous geometry and mesh density combinations. Both the initial coarse background mesh and the final dense mesh can be generated adaptively if desired (although the adaptive meshing of the fine mesh in parallel has not yet been demonstrated), where the local element size is determined by a querying a field function (typically an error function from a previous run), yielding an appropriately sized mesh which captures error gradients for finer resolution and increased solution accuracy.

Our second conclusion is that the generation of a paved mesh in parallel is very feasible from both a performance and a mesh quality standpoint. Performance data on an IBM SP2 parallel machine yielded super-linear speedups over a variety of problem sizes (1040 to 63517 finite element nodes) with parallel runs using 2, 4, and 8 compute nodes. Efficiencies from these runs (speedup/number of processors) ranged from 42 to 141 percent, becoming more efficient as more parallel compute nodes are employed in general.

The wide range of efficiencies can be attributed to the overhead incurred when a larger number of subdomains must be managed by each processor (in the case of the nose cone example, up to 20 separate meshing operations could be assigned to a single processor for a two processor decomposition). As the number of subdomains per processor increases, so does the overhead of model management at each processor. This is mainly a problem when the overall mesh is coarse, since a greater proportion of the compute time is allocated to geometry management as opposed to parallel paving. This effect causes some of the efficiencies to be quite low. When parallel paving dominates the compute time, the efficiencies are significantly higher, in some cases greater than 100 percent due to the speedups that paving undergoes. This effect is the divide and conquer effect

which paving receives via domain decomposition, and is the reason for the highest efficiency values.

Differences in mesh quality between the serially paved mesh and the 8-node parallel equivalent were acceptable: element aspect ratio degraded by 9 percent going from the serial to the parallel version. These performance conclusions are applicable to coarse grain parallel architectures, but fine grain environments may yield different results.

Conclusions regarding the performance of the dynamic load balancer are incomplete, as time limitations precluded the testing of this capability sufficiently. When the mesher begins to generate adaptively, the parallel subdomains will go out of balance rapidly, necessitating a capable parallel load balancer. Ideally the load balancing code ought to be linked together with the meshing code to share common data structures for efficiency.

Recommendations are to continue an investigation into parallel load balancing which must be proved viable for final conclusions on parallel adaptive meshing, and to further investigate the performance of the serial paver with the new insight provided by the apparent super-linear speedup results from the parallel paver. These speedups indicate that the serial paver has a scaling problem which ought to be addressed. It is our opinion that a form of bin sorting prior to paving initiation would help the conventional paver's speed significantly.

In addition, a topic which did not get addressed during the project was the relaxing of interior (non-geometric) subdomain boundaries for inter-processor mesh smoothing to reduce geometric artifacts. This may help improve the quality of the mesh significantly. An additional topic which ought to be pursued is the influence that subdomain shape has on final mesh quality.

References

1. Blacker, T. D., and Stephenson, M. B., "Paving: A New Approach to Automated Quadrilateral Mesh Generation", *Int. j. numer. methods eng.*, **Vol. 32**, 1991, pp. 811 - 47.
2. Blacker, T. D., et al., *CUBIT Mesh Generation Environment, Volume 1: User's Manual*, SAND94-1100, Sandia National Laboratories, Albuquerque, New Mexico, 1994.
3. Wheat, S. R., Devine, K. D., and Maccabe, A. B., "Experience with Automatic, Dynamic Load Balancing and Adaptive Finite Element Computation", Proceedings of the *27th Hawaii International Conference on System Sciences*, **Vol. II**, 1994, pp. 463-72.
4. Devine, K. D., Flaherty, J. E., Wheat, S. R., and Maccabe, A. B., "A Massively Parallel Adaptive Finite Element Method with Dynamic Load Balancing", Proceedings of *Supercomputing '93*, IEEE Computer Society Press, 1993, pp. 2-11.
5. Taylor, L. M., and Flanagan, D. P., *PRONTO 3D A Three-Dimensional Transient Solid Dynamics Program*, SAND87-1912, Sandia National Laboratories, Albuquerque, New Mexico, 1989.
6. Attaway, S. W., *Update of PRONTO 2D and PRONTO 3D Transient Solid Dynamics Program*, SAND90-0102, Sandia National Laboratories, Albuquerque, New Mexico, 1990.
7. Hughes, T. J. R., *The Finite Element Method*, Prentice-Hall, Inc., Englewood Cliffs, New Jersey, 1987, pp. 242-54.
8. Jones, R. E., *Users Manual for QMESH, a Self-Organizing MESH Generation Program*, SLA-74-0239, Sandia National Laboratories, Albuquerque, New Mexico, 1974.
9. Blacker, T. D., Stephenson, M. B., Mitchiner, J. L., Phillips, L. R., and Lin, Y. T., "Automated Quadrilateral Mesh Generation: A Knowledge System Approach", *ASME Paper No. 88-WA/CIE-4*, 1988.
10. Chavez, P. F., *Automatic 3-D Finite Element Modeling*, SAND89-0047C, Sandia National Laboratories, Albuquerque, New Mexico, 1989.
11. Blacker, T. D., *FASTQ Users Manual Version 2.1*, SAND88-1326, Sandia National Laboratories, Albuquerque, New Mexico, 1988.
12. Gilkey, A. P., and Sjaardema, G. D., *GEN3D: A GENESIS Database 2D to 3D Transformation Program*, SAND89-0485, Sandia National Laboratories, Albuquerque, New Mexico, 1989.
13. Sjaardema, G. D., *GJOIN: A Program for Merging Two or More GENESIS Databases*, SAND90-0566, Sandia National Laboratories, Albuquerque, New Mexico, 1990.
14. Sjaardema, G. D., *GREPOS: A GENESIS Database Repositioning Program*, SAND92-2290, Sandia National Laboratories, Albuquerque, New Mexico, 1992.
15. Schoof, L. A., and Yarberry, V. R., *EXODUS II: A Finite Element Data Model*, SAND92-2137, Sandia National Laboratories, Albuquerque, New Mexico, 1994.
16. Gartling, D. K., and Hogan, R. E., *Coyote II: A Finite Element Computer Program for Nonlinear Heat Conduction Problems Part II - User's Manual*, SAND94-1179, Sandia National Laboratories, Albuquerque, New Mexico, 1994.
17. Schunk, P. R., Sackinger, P. A., Rao, R. R., Chen, K. S., and Cairncross, R. A., *GOMA - A Full-Newton Finite Element Program for Free and Moving Boundary Problems with Coupled Fluid/Solid Momentum, Energy, Mass, and Chemical Species Transport: User's Guide*, SAND95-2937, Sandia National Laboratories, Albuquerque, New Mexico, 1996.
18. Spatial Technology, Inc., *ACIS Geometric Modeler Application Guide Version 1.7*, Spatial Technology, Inc., Three-Space, Ltd., and Applied Geometry, Corp., 1995.
19. Tautges, T. J., Blacker. T. D., and Mitchell, S. A., "The Whisker Weaving algorithm: A Connectivity-based Method for Constructing All-Hexahedral Finite Element Meshes", *Int. j. numer. methods eng.*, **Vol. 39**, No. 19, 1996, pp. 3327 - 50.
20. Hipp, J. R., Lober, R. R., Blacker, T. D., and Meyers, R. J., "Plastering: Automated All-hexahedral Mesh Generation of General Geometries through Connectivity Resolution", to be published.
21. Blacker. T. D., and Meyers, R. J., "Seams and Wedges in Plastering: A 3-D Hexahedral Mesh Generation Algorithm", *Eng. comp.*, **Vol. 9**, 1993, pp. 83 - 93.

22. Lober, R., R., Tautges, T. J., and Cairncross, R. A., "The Parallelization of an Advancing-front, All-quadrilateral Meshing Algorithm for Adaptive Analysis", proceedings of the *4th International Meshing Roundtable,* Albuquerque, New Mexico, SAND95-2130, 1995, pp. 59 - 70.

23. Cook, W. A., and Oakes, W. R., "Mapping Methods for Generating Three-dimensional Meshes", *Comp. mech. eng.*, **Vol. 1**, 1982, pp. 67 - 72.

24. Whiteley, M., White, D. R., Benzley, S. E., and Blacker. T. D., "Two and Three-Quarter Dimensional Meshing Facilitators", *Eng. comp.*, **Vol. 12**, 1996, pp. 144 - 54.

25. White, D. R., Mingwu, L., Benzley, S. E., and Sjaardema, G. D., "Automated Hexahedral Mesh Generation by Virtual Decomposition", proceedings of the *4th International Meshing Roundtable*, Albuquerque, New Mexico, SAND95-2130, 1995, pp. 165 - 76.

26. Mingwu, L., Benzley, S. E., Sjaardema, G. D., and Tautges, T. J., "A Multiple Source and Target Sweeping Method for Generating All Hexahedral Finite Element Meshes", proceedings of the *5th International Meshing Roundtable*, Pittsburgh, Pennsylvania, SAND96-2301, 1996, pp. 217 - 25.

27. Cass, R. J., Benzley, S. E., Meyers, R. J., and Blacker. T. D., "Generalized 3D Paving: An Automated Quadrilateral Surface Mesh Generation Algorithm", *Int. j. numer. methods eng.*, **Vol. 39**, No. 9, 1996, pp. 1475 - 90.

28. Blacker. T. D., Jung, J., and Witkowski, W. R., "An Adaptive Finite Element Technique Using Element Equilibrium and Paving", ASME Paper No. 90-WA/CIE-2, November, 1990.

29. Zienkiewicz, O. C., and Zhu, R. J. Z., "A Simple Error Estimator and Adaptive Procedure for Practical Engineering Analysis", *Int. j. numer. methods eng.*, **Vol. 24**, No. 2, 1987, pp. 337 - 57.

30. Pelletier, D., and Illinca, F., "Adaptive Finite Element Method for Mixed Convection", *J. thermophys. heat trans.*, **Vol. 9**, No. 4, 1995, pp. 708 - 14.

31. Tautges, T. J., Lober, R., R., Vaughan, C. T., and Jung, J., "The Design of a Parallel Adaptive Paving All-Quadrilateral Meshing Algorithm", proceedings of the *Sixth International Symposium on Computational Fluid Dynamics*, Lake Tahoe, Nevada, Vol. 3, 1995, pp. 1255 - 60.

32. Wu, P. and Houstis, E. N., "Parallel Adaptive Mesh Generation and Decomposition", *Eng. comp.*, **Vol. 12**, 1996, pp. 155 - 67.

33. LaCourse, D. E., ed., *Handbook of Solid Modeling*, McGraw-Hill, Inc., New York, 1995, p. 4.7.

34. Mortenson, M. E., *Geometric Modeling*, John Wiley & Sons, Inc., New York, 1985, p. 94.

35. Sjaardema, G. D., et. al., *CUBIT Mesh Generation Environment, Volume 2: Developers Manual*, SAND94-1101, Sandia National Laboratories, Albuquerque, New Mexico, 1994.

36. Hendrickson, B. and Leland, R., *The Chaco User's Guide Version 1.0*, SAND93-2339, Sandia National Laboratories, Albuquerque, New Mexico, 1993.

37. Summers, R. M., et. al., "Recent Progress in ALEGRA Development and Applications to Ballistic Impacts", to be published in the *Proceedings to the 1996 Hypervelocity Impact Symposium, Int. j. impact eng.*, **Vol. 20**, 1997.

38. Gropp, W., Lusk, E., and Skjellum, A., *Using MPI: Portable Parallel Programming with the Message-Passing Interface*, The MIT Press, Cambridge, Massachusetts, 1994, p. 28 - 9.

39. International Business Machines Corporation, *IBM Parallel Environment for AIX: Operation & Use Version 2.1.0*, International Business Machines Corporation, Document No. GC23-3891-00, 1995.

40. Leland, R. W., *The Effectiveness of Parallel Iterative Algorithms for Solution of Large Sparse Linear Systems*, PhD. Thesis, University of Oxford, Oxford, England, 1989, p. 2.

41. Sjaardema, G. D., *NUMBERS: A Collection of Utilities for Pre- and Postprocessing Two- and Three-Dimensional EXODUS Finite Element Models,* SAND88-0737, Sandia National Laboratories, Albuquerque, New Mexico, 1989.

42. Robinson, J., "CRE Method of Element Testing and the Jacobian Shape Parameters", *Eng. comp.*, **Vol. 4**, 1987, pp. 113 - 18.

43. Gamma, E., et. al., *Design Patterns: Elements of Reusable Object-oriented Software*, Addison-Wesley, Reading, Massachusetts, 1995, pp. 127 - 34.

Appendix A Parallel Meshing Example Files

A.1 Syntax of Selected Commands

The *merge* command exists in both CUBIT and CUBITP to allow for non-manifold (see Section 2.2.1 on page 18) model construction. This command is necessary to combine all the redundant reference entity data structures, or structures that represent spatially equivalent ACIS® entities. The syntax of this command is:

```
merge all
```

As a help to visualize these relationships, CUBIT assigns separate colors on a RefVolume basis, such that all reference entities below the RefVolume level for a given volume accept the color of their higher topological level parent (all the surfaces, curves, and vertices of the volume share the volume's assigned color). When a pair of entities is merged, the entity surviving the merge retains its previous color assignment, and as such when entities are displayed, verification of the proper merged relationship can be viewed by observing if the shared entity appears in the same color when either of its two owning parents are plotted. For the example in Figure 2-7, the survivor of the RefEdge merge would have come from RefFace 1 (since its id value is lower), and would retain the original color from RefFace 1. Then when either RefFace 1 or RefFace 2 was displayed, the RefEdge with the color from RefFace 1 would be drawn both times, visually indicating the shared relationship and parentage of the merged RefEdge.

The need to preserve geometric topology between parallel remeshing steps is met with the *nodeset associativity* command. The user is required to generate the topology records for storage into the ExodusII [15] file format. The need for and the mechanism of storing these records is addressed in Section 3.3 on page 36. The command syntax is:

```
nodeset associativity on
```

which will generate the necessary data for export to the ExodusII file format.

The following several commands exist specifically for parallel operations.

The *import subdomains* command is used to construct the virtual geometry structures by simply the importing the decomposed subdomain mesh files. The command syntax is:

```
import subdomains '<filename_prefix>'
```

The filename_prefix is the root file name of the suite of subdomain files produced by EDDT. CUBITP follows the EDDT parallel file naming convention, adding the suffixes of "gen.<processor_id>" to the root file name for each refined mesh file. Thus for a coarse model file named nose_cone.gen, the processor files would be named nose_cone.gen.0, nose_cone.gen.1, etc.

Analogously, EDCT expects names like nose_cone.exo.0, and will concatenate the files into a single file named nose_cone.exo.

The *mesh subdomains* command is used to initiate the parallel meshing process. The command syntax is:

```
mesh subdomains
```

This command invokes the parallel mesh tool, which ensures that subdomains can in fact close with all-quadrilateral meshes. It also manages the boundary negotiation and mesh generation processes.

The *export subdomains* command is used to complete the finite element model generation process and write out the suite of refined mesh data files. The command syntax is:

```
export subdomains '<filename_prefix>'
```

where the filename_prefix is again the same type of variable that was described above.

A.2 Journal File for Initial Coarse Spline Mesh

```
## Cubit Version 1.11.5-geom-05, Run on 09/05/96 at 03:47:28 PM
## Command Options:
## -warning = On
## -debug =
## -information = On
## -initfile /home/rrlober/.cubit
# General debugging
set info on

# Geometry creation
import acis 'spline.sat'

# Meshing scheme and typical element size settings
surface 5 size 13
surface 5 scheme pave

# Store topology records for Exodus II
nodeset associativity on

# Initiate timing for performance measurement
set start time

# Generate the coarse mesh
mesh surface 5

# Stop timer
set end on

# Assign the element block extent and element type
block 1 surface 5
block 1 element type QUAD4

# Write out the file
export genesis 'spline.gen'

# Terminate mesher
quit
```

A.3 Journal File for Parallel Meshing of Spline

```
## Cubit Version 1.11.5-geom-05, Run on 09/05/96 at 04:00:31 PM
## Command Options:
## -warning = On
## -debug =
## -information = On
## -initfile /home/rrlober/.cubit
# General debugging
set info on

# Geometry creation
```

```
import acis 'spline.sat'
display

# Read in background mesh decomposition and finite element model
import subdomains 'spline'

# Meshing scheme and typical element size settings
surf all sche pave
surf all size 3.3

# Initiate timing for performance measurement
set start time

# Parallel pave
mesh subdomains

# Stop timer
set end time

# Write out the new, refined mesh data file series (not the coarse background mesh)
exp subdomains 'pspl'

# Terminate mesher
quit
```

A.4 Journal File for Initial Coarse Nose Cone Mesh

```
## Cubit Version 1.11.5-geom-05, Run on 09/05/96 at 05:18:12 PM
## Command Options:
## -warning = On
## -debug =
## -information = On
## -initfile /home/rrlober/.cubit
# General debugging turned off for performance
set info off
journal off

# Aprepro variables for mesh size control
{large_size=.12}
{small_size=0.75*large_size}

# Geometry creation
import acis "nose.sat"

# Resolve non-manifold model for this initial mesh
merge all

# Meshing scheme setting
surf 1 to 45 sche pave

# Set typical element size using Aprepro variables, large_size for larger surfaces
surf 1 to 4 size {large_size} even
```

```
surf 13 to 16 size {large_size} even
surf 25 to 28 size {large_size} even
surf 37 to 40 size {large_size} even

surf 5 to 9 size {small_size} even
surf 17 to 21 size {small_size} even
surf 29 to 33 size {small_size} even
surf 41 to 45 size {small_size} even

# Store topology records for Exodus II
nodeset associativity on

# Initiate timing for performance measurement
set start on

# Mesh all the surfaces
mesh surf 5 to 9
mesh surf 17 to 21
mesh surf 29 to 33
mesh surf 41 to 45

mesh surf 1 to 4
mesh surf 13 to 16
mesh surf 25 to 28
mesh surf 37 to 40

# Stop timer
set end on

# Assign the element block extent and element type
block 100 surf 5  to 9
block 200 surf 17 to 21
block 300 surf 29 to 33
block 400 surf 41 to 45

block 100 surf 1 to 4
block 200 surf 13 to 16
block 300 surf 25 to 28
block 400 surf 37 to 40

block 100 element type shell4
block 100 attribute 0.2
block 200 element type shell4
block 200 attribute 0.2
block 300 element type shell4
block 300 attribute 0.2
block 400 element type shell4
block 400 attribute 0.2

# Write out the file
export genesis 'nose.gen'

# Terminate mesher
quit
```

A.5 Journal File for Parallel Meshing of Nose Cone

```
## Cubit Version 1.11.5-geom-05, Run on 09/05/96 at 05:56:29 PM
## Command Options:
## -warning = On
## -debug =
## -information = On
## -initfile /home/rrlober/.cubit
# General debugging turned off for performance
set info off
journal off

# Geometry creation
import acis 'nose.sat'

# Resolve non-manifold model for this initial mesh
merge all

# Read in background mesh decomposition and finite element model
import subdomains 'nose'

# Meshing scheme and typical element size settings
surf all sche pave
surf all size 0.12

# Initiate timing for performance measurement
set start time

# Parallel pave
mesh subdomains

# Stop timer
set end time

# Write out the new, refined mesh data file series (not the coarse background mesh)
export subdomains 'pnose'

# Terminate mesher
quit
```

Appendix B Selected Class Headers

B.1 MeshTopologyVertex Class Declaration

```
//- Class:       MeshTopologyVertex
//- Owner: Tim Tautges
//- Description: MeshTopologyVertex is a vertex based on the position of a node
//- Checked by:
//- Version: $Id: MeshTopologyVertex.hpp,v 1.2 1995/07/11 13:35:10 gdsjaar Exp $

#ifndef MESHTOPVERTEX_HPP
#define MESHTOPVERTEX_HPP

#include "VertexEval.hpp"

class PRefVertex;

class MeshTopologyVertex : public VertexEval
{
private:

  PRefVertex *refVertex;
  //- ref vertex to which this evaluator applies

  CubitNode *vertexNode;
  //- node which determine's this vertex's position

public:

  MeshTopologyVertex(PRefVertex *vertex, CubitNode *node);
  //- Constructor assumes vertex is already meshed

  ~MeshTopologyVertex();
  //- Destructor

  CubitNode *vertex_node();
  void vertex_node(CubitNode *node);
  //- get/set vertexNode

  PRefVertex *ref_vertex() {return refVertex;};
  void ref_vertex(PRefVertex *vertex) {refVertex = vertex;};
  //- get/set refVertex

  int contains_node_location ( CubitNode *node );
  //- returns true if the location of this vertex matches that of the node

  CubitVector coordinates();
  //- returns the coordinates of this vertex

  int type ();
  //- return the type of evaluator this is
};
```

```
inline int MeshTopologyVertex::type ()
{
  return MESH_VERTEX;
}

#endif // MESHTOPVERTEX_HPP
```

B.2 MeshTopologyCurve Class Declaration

```
//- Class:       MeshTopologyCurve
//- Owner: Tim Tautges
//- Description: MeshTopologyCurve is a curve whose geometry is based on ACIS
//-              but whose topology is based on an underlying mesh
//- Checked by:
//- Version: $Id: MeshTopologyCurve.hpp,v 1.4 1996/01/05 23:04:42 rrlober Exp $

#ifndef MESHTOPOLOGYCURVE_HPP
#define MESHTOPOLOGYCURVE_HPP

#include "CurveEval.hpp"
#include "CubitVector.hpp"

class RefEntity;
class RefVertex;
class PRefCurve;
class MeshData;

class MeshTopologyCurve : public CurveEval
{
private:

  RefEntity *parentGeom;
  //- relies on another entity for geometric operations like move_to

  PRefCurve *refCurve;
  //- refcurve to which this evaluator belongs

  MeshData *meshData;
  //- mesh data items making up this curve

  double curveLength;
  //- summed length of edges along this curve

  RefEntity *parent_geom() {return parentGeom;};
  void parent_geom(RefEntity *geom) {parentGeom = geom;};
  //- Get/set parentGeom

public:

  MeshTopologyCurve (PRefCurve *ref_curve, RefEntity *parent_geom,
                                     DLCubitNodeList &node_list);
  //- Constructor
```

```
~MeshTopologyCurve();
//- Destructor

MeshData *mesh_data() {return meshData;};
void mesh_data(MeshData *data) {meshData = data;};
//- get/set meshData

PRefCurve *ref_curve() {return refCurve;};
void ref_curve(PRefCurve *curve) {refCurve = curve;};
//- get/set refcurve

double start_param();
double end_param();
double param_range();
//- parameter values (inlined below)

int contains (const CubitVector &location);
//- return TRUE if location falls on this curve, FALSE otherwise

int sense();
//- return the sense for (0 = CUBIT_FORWARD, 1 = CUBIT_REVERSED) this curve
//- this uses the same style as ACIS (REVBIT) in acis1.6/kerndata/top/tophdr.hxx

double position_to_param (const CubitVector &position);
//- return parameter for this position on the curve. Note that the position
//- must lie on the curve - if not, param of 0.0 will be returned and error
//- message will result

double parameter_at_base_node (const CubitNode &node);
//- return parameter for this node on the curve. Note that the node
//- must be part fo the definition of the curve (owned by one of the
//- CubitEdges in the meshData object)

double length(double num1 = NULL, double num2 = NULL);
//- arc length for this curve

double calculate_length();
//- calculate the length for this curve

double length_to_param(double param_low, double arc_length);
//- return the parameter at a distance along this curve, starting from param_low

CubitVector param_to_position ( double param, int, int );
//- return the position of this parameter value

CubitVector mid_point ();
//- return the midpoint position of this curve

RefVertex *start_ref_vertex ();
//- return the starting ref vertex for a curve

RefVertex *end_ref_vertex ();
//- return the ending ref vertex for a curve
```

```
  void move_to ( CubitNode *node );
  //- move the node to this entity

  void move_to ( CubitVector &location );
  //- move the location to this entity

  int type ();
  //- return the type of evaluator this is

  int contains_as_base ( const CubitNode &node );
  // Returns TRUE/FALSE if the input node was used in the definition of
  // the MeshTopologyCurve

  int about_equal (const CubitVector& Vec1,
                                    const CubitVector& Vec2,
                                    double factor = 1.0);
//- Return CUBIT_TRUE if the the 2 CubitVectors are within factor*CUBIT_RESABS
//- of each other

  int intersect_fraction ( const CubitVector& Vec1,
                           const CubitVector& Vec2,
                           const CubitVector& Vec3,
                           double &arc_fraction      );
  //- Returns TRUE if Vec3 lies on the line segment formed between Vec1 and Vec2
  //- and puts the percent distance from Vec1 into arc_fraction. Returns FALSE if
  //- Vec3 does NOT lie on the line segment between Vec1 and Vec2

};

inline double MeshTopologyCurve::start_param()
{
  return 0.0;
}

inline double MeshTopologyCurve::end_param()
{
  return 1.0;
}

inline double MeshTopologyCurve::param_range()
{
  return 1.0;
}

inline double MeshTopologyCurve::length(double, double)
{
  if (curveLength < 0.0) calculate_length();
  return curveLength;
}

inline int MeshTopologyCurve::type()
{
  return MESH_TOPOLOGY_CURVE;
}
#endif // MESHTOPOLOGYCURVE_HPP
```

B.3 MeshTopologySurface Class Declaration

```
//- Class:       MeshTopologySurface
//- Owner: Tim Tautges
//- Description: MeshTopologySurface is a surface whose geometry is based on ACIS
//-              but whose topology is based on an underlying mesh
//- Checked by:
//- Version: $Id: MeshTopologySurface.hpp,v 1.4 1995/12/04 17:49:33 rrlober Exp $

#ifndef MESHTOPSURFACE_HPP
#define MESHTOPSURFACE_HPP

#include "SurfaceEval.hpp"
#include "DLRefEdgeList.hpp"
#include "DLRefVertexList.hpp"
#include "DLCubitFaceList.hpp"
#include "DLCubitNodeList.hpp"
#include "CubitVector.hpp"

class RefFace;
class PRefSurface;
class MeshData;


class MeshTopologySurface : public SurfaceEval
{
private:

  RefFace *parentGeom;
  //- relies on another entity for geometric operations like move_to

  DLRefEdgeList curveList;
  //- list of curves bounding this surface

  RefFace *parent_geom() {return parentGeom;};
  void parent_geom(RefFace *geom) {parentGeom = geom;};
  //- Get/set parentGeom

public:

  MeshTopologySurface(RefFace *parent_geom, DLRefEdgeList &curve_list);
//- Constructor

  ~MeshTopologySurface();
//- Destructor

  CubitVector normal_at ( CubitVector location, RefVolume *volume = NULL);
  //- returns the normal at the given location

  void move_to ( CubitNode *node );
  //- relax point to lie on this curve

  void move_to ( CubitVector &vector );
```

```
  //- relax point to lie on this curve

  void ref_surfaces ( DLPRefSurfaceList &surface_list );
  //- return the curves on this surface

  void ref_curves ( DLRefEdgeList &curve_list );
  //- return the curves on this surface

  void ref_vertices ( DLRefVertexList &vertex_list );
  //- return the vertices on this surface

  int node_loops ( DLNodeLoopList &node_list );
  //- return the node loops for this surface

  int type ();
  //- return the type of evaluator this is
};

inline int MeshTopologySurface::type ()
{
  return MESH_TOPOLOGY_SURFACE;
}

inline void MeshTopologySurface::ref_surfaces ( DLPRefSurfaceList &surface_list )
{
  return;
}


#endif // MESHTOPSURFACE_HPP
```

B.4 ParallelMeshTool Class Declaration

```
//- Class: ParallelMeshTool
//- Description: ParallelMeshTool class - tool for meshing in parallel
//-              This class is implemented as a {singleton} pattern. Only
//-              one instance is created and it is accessed through the
//-              {instance()} static member function.
//-              This class stores information that is needed to manage
//-              the interfaces between mesh regions which are shared
//-              by multiple processors.
//- Owner: Randy Lober
//- Checked by:
//- Version: $Id: ParallelMeshTool.hpp,v 1.2 1995/05/10 15:39:31 rrlober Exp $

#ifndef PARALLELMESHTOOL_HPP
#define PARALLELMESHTOOL_HPP

#include "MeshTool.hpp"
#include "RefEntity.hpp"
#include "DLRefEntityList.hpp"
#include "CubitLogical.hpp"
```

```
class ParallelMeshTool : public MeshTool
{

private:

  static ParallelMeshTool* instance_;

  DLRefEntityList local_subdomains;  // List of entities (PRefFaces, etc.)
                                     //
  ExodusMesh *exodusMeshPtr; // Storage for sizing function

protected:

  ParallelMeshTool(); //(Not callable by user code. Class is constructed
                      //- by the {instance()} member function.

  void draw_mesh();

  int mesh_subdomain_boundaries();
//- Resolves the master-slave relationship on boundaries which are shared by
//- 2 processors, then meshes the owned boundaries and sends/receives results
//- between master and slave processors.

  int resolve_curve_mesh( RefEdge * loc_ref_edge );
//- Checks the master_slave_status and if master, calls pack_and_send_mesh.
//- If slave, it calls receive_and_unpack_mesh for mesh from the master,
//- Returns TRUE if successful, FALSE otherwise.

  int pack_and_send_mesh ( RefEdge * loc_ref_edge );
//- Treats the input RefEdge * as a master and packs and sends the
//- list of nodes in order to the partner processor.

  int receive_and_unpack_mesh ( RefEdge * loc_ref_edge );
//- Treats the input RefEdge * as a slave and unpacks the
//- list of nodes from the master processor and completes the curve mesh.

public:

  ~ParallelMeshTool ();

  static ParallelMeshTool* instance();

  int mesh_subdomains ();
  // Meshes each subdomain owned by this processor

  void add_subdomain ( RefEntity *prefentity_ptr ) { local_subdomains.append
                                                                          (
prefentity_ptr );  }
  // Add a new subdomain to the list for this processor

  int get_global_element_id_offset ( int local_number_elements,
                                     int &offset );
  // This routine computes the id offsets for each processor,
  // with processor 0 having an offset of 0, i.e., proc 0 id's
```

```
  // are the same locally and globally.Proc 1's id's are all
  // incremented globally by the number of elements on proc 0,
  // etc.

  int get_global_node_id_offset ( int local_number_nodes,
                                  int &offset );
  // This routine computes the id offsets for each processor,
  // with processor 0 having an offset of 0, i.e., proc 0 id's
  // are the same locally and globally.Proc 1's id's are all
  // incremented globally by the number of nodes on proc 0,
  // etc.

};

#endif
```

Appendix C Singleton Pattern

C.1 Singleton Pattern Use in CommunicationTool Class

A number of design patterns for object-oriented software design have emerged recently [43], providing simple and elegant solutions to specific problems specific to object-oriented design. The parallel paving project has some specific needs for communication software design, and as such has elected to use the singleton design pattern as part of the implementation of the CommunicationTool class. A description of this pattern will be given.

The intent of the singleton method is to guarantee that a class has but a single instance, and then provide a global point of access to that instance. Our motivation for this choice is to ensure that there can be no more than one object attempting to coordinate and execute communication requests in our parallel implementation. The design of the CommunicationTool class itself using this pattern ensures that there is only one instance of the object.

Benefits of the singleton pattern are:

1. Controlled access to sole instance
2. Reduced name space
3. Permits refinement of operations and representation
4. Permits a variable number of instances if desired
5. More flexible than class operations

The following is our implementation of the CommunicationTool class using the singleton pattern.

```
//- Class:       CommunicationTool
//- Description: Class which wrappers the communication calls needed for
//-              parallel processing.
//- Owner:       Randy Lober
//- Checked by:
//- Version: $Id: CommunicationTool.cc,v 1.1.2.1 1996/03/06 23:08:18 rrlober Exp $

#include "CommunicationTool.hpp"
#include "NoCommunication.hpp"

#if defined( MPI_COMMUNICATIONS_ENABLED )
#include "MPICommunication.hpp"
#endif

CommunicationTool* CommunicationTool::instance_ = 0;
```

```
CommunicationTool::CommunicationTool()
{
  startTime = NULL;
  endTime = NULL;
}

CommunicationTool::~CommunicationTool()
{
  instance_ = NULL;
}

CommunicationTool * CommunicationTool::instance()
{
  if (instance_ == 0)
    {
#if defined( MPI_COMMUNICATIONS_ENABLED )
// For MPI runs
      instance_ = new MPICommunication;
#else
// default
      instance_ = new NoCommunication;
#endif
    }
  return instance_;
}
```

A concrete instance of either the MPICommunication or NoCommunication is instantiated when CommunicationTool::instance() is called for the first time. Note that because instance_ is a pointer to a singleton object, the instance member function can assign a pointer to a subclass of singleton to this variable. This is what we do in the MPICommunication and NoCommunication classes. Here is the definition of our MPICommunication class.

```
//- Class:       MPICommunication
//- Description: Concrete class which wrappers the communication calls needed for
//-              MPI calls.
//- Owner:       Randy Lober
//- Checked by:
//- Version: $Id: MPICommunication.cc,v 1.1.2.1 1996/03/06 23:08:10 rrlober Exp $

#include "MPICommunication.hpp"
#include "mpi.h"
#include <iostream.h>

int MPICommunication::communicationsActive = CUBIT_FALSE;

MPICommunication * MPICommunication::cast_to_mpi_comm() { return this; }

MPICommunication::MPICommunication()
{
}

MPICommunication::~MPICommunication()
```

```
{
}

void MPICommunication::initialize( int &argc,
                                   char **&argv )
{
int info = 0;

info = MPI_Init(&argc,&argv);

  cout << " Just did init for proc " << processor_id() << ", info = "
       << info << endl << flush;

  // Set up an error handler so if MPI bails we get some clues
  // This next code changes the default MPI behavior to return
  // an error condition instead of terminating,
  // The error messages are printed out in ::handle_error with
  // a string that describes the condition. See p.179-80 of Using MPI,
  // Author(s) W. Gropp, E. Lusk, & A. Skjellum
  MPI_Errhandler_set ( MPI_COMM_WORLD,
                       MPI_ERRORS_RETURN );

// Since we're using Bsend's (aka JP), we need to attach a buffer to begin with
int init_mpi_buf_len = 100000 * sizeof(double);
char* init_mpi_buf = new char[init_mpi_buf_len];
handle_error ( MPI_Buffer_attach( init_mpi_buf,
                                  init_mpi_buf_len ));

  active ( CUBIT_TRUE );
}

void MPICommunication::finalize()
{
  active ( CUBIT_FALSE );
  MPI_Finalize();
}

int MPICommunication::processor_id() const
{
  int myid;
  handle_error ( MPI_Comm_rank( MPI_COMM_WORLD,
                                &myid) );
  return myid;
}

int MPICommunication::number_of_processors() const
{
  int numprocs;
  handle_error ( MPI_Comm_size ( MPI_COMM_WORLD,
                                 &numprocs) );
  return numprocs;
}

void MPICommunication::get_processor_name( char *name ) const
{
```

```
  int loc_nam_len = 0;
  handle_error ( MPI_Get_processor_name ( name,
                          &loc_nam_len ) );
}

void MPICommunication::mark_start_wtime()
{
  start_time ( MPI_Wtime () );
}

void MPICommunication::mark_end_wtime()
{
  end_time ( MPI_Wtime () );
}

double MPICommunication::calculate_total_time () const
{
  double total_time = end_time() - start_time();
  return total_time;
}

int MPICommunication::send_data(int processor,
                                int* data,
                                int len,
                                int mesg)
{
  int info = 0;

  info = MPI_Bsend((void*) data,
                   len,
                   MPI_INT,
                   processor,
                   mesg,
                   MPI_COMM_WORLD);

  handle_error ( info );


  if (info != MPI_SUCCESS) return CUBIT_FALSE;

  return CUBIT_SUCCESS;
}

int MPICommunication::recv_data(int processor,
                                int* data,
                                int len,
                                int mesg)
{
  int info = 0;

  MPI_Status status;
  info = MPI_Recv((void*) data,
                  len,
                  MPI_INT,
                  processor,
```

```
                          mesg,
                          MPI_COMM_WORLD,
                          &status);

  handle_error ( info );

if (info != MPI_SUCCESS) return CUBIT_FALSE;

return CUBIT_SUCCESS;
}

int MPICommunication::send_data(int processor,
                                double* data,
                                int len,
                                int mesg)
{
  int info = 0;

  info = MPI_Bsend((void*) data,
                          len,
                          MPI_DOUBLE,
                          processor,
                          mesg,
                          MPI_COMM_WORLD);

  handle_error ( info );

if (info != MPI_SUCCESS) return CUBIT_FALSE;

return CUBIT_SUCCESS;
}

int MPICommunication::recv_data(int processor,
                                double* data,
                                int len,
                                int mesg)
{
  int info = 0;

MPI_Status status;
  info = MPI_Recv((void*) data,
                          len,
                          MPI_DOUBLE,
                          rocessor,
                          mesg,
                          MPI_COMM_WORLD,
                          &status);

  handle_error ( info );

  if (info != MPI_SUCCESS) return CUBIT_FALSE;

return CUBIT_SUCCESS;
}
```

```
void MPICommunication::handle_error ( int error_code ) const
{
  if ( error_code != MPI_SUCCESS )
    {
      int error_class;
      MPI_Error_class ( error_code,
                        &error_class );

      char buffer[MPI_MAX_ERROR_STRING];
      int resultlen;

      MPI_Error_string ( error_code,
                         buffer,
                         &resultlen );

      PRINT_ERROR ( "MPICommunications Error: %s\n",
                            buffer );

}
}

int MPICommunication::global_sum(double value,
                                      double &sum)
{
  int info = 0;

  info = MPI_Allreduce(&value,
                           &sum,
                           1,
                           MPI_DOUBLE,
                           MPI_SUM,
                           MPI_COMM_WORLD);

  handle_error ( info );

  if (info != MPI_SUCCESS) return CUBIT_FALSE;

  return CUBIT_SUCCESS;
}

int MPICommunication::global_sum(int value,
                                      int &sum)
{
  int info = 0;

  info = MPI_Allreduce(&value,
                           &sum,
                           1,
                           MPI_INT,
                           MPI_SUM,
                           MPI_COMM_WORLD);

  handle_error ( info );

  if (info != MPI_SUCCESS) return CUBIT_FALSE;
```

```
  return CUBIT_SUCCESS;
}

int MPICommunication::global_maximum(double value,
                                     double &max)
{
  int info = 0;

  info = MPI_Allreduce(&value,
                       &max,
                       1,
                       MPI_DOUBLE,
                       MPI_MAX,
                       MPI_COMM_WORLD);

  handle_error ( info );

  if (info != MPI_SUCCESS) return CUBIT_FALSE;

  return CUBIT_SUCCESS;
}

int MPICommunication::global_maximum(int value,
                                     int &max)
{
  int info = 0;

  info = MPI_Allreduce(&value,
                       &max,
                       1,
                       MPI_INT,
                       MPI_MAX,
                       MPI_COMM_WORLD);

  handle_error ( info );

  if (info != MPI_SUCCESS) return CUBIT_FALSE;

  return CUBIT_SUCCESS;
}
```

Sandia Internal Distribution

1	MS-0188	D. L. Chavez,	LDRD Office
1	MS-0321	W. J. Camp,	09200
1	MS-0441	CUBIT Report File,	09226
1	MS-0441	B. W. Clark,	09226
1	MS-0441	R. W. Leland,	09226
1	MS-0441	S. A. Mitchell,	09226
1	MS-0441	R. W. Ostensen,	09226
1	MS-0441	R. W. Leland,	09226
1	MS-0441	L. A. Schoof,	09226
5	MS-0441	T. J. Tautges,	09226
1	MS-0441	D. R. White,	09226
1	MS-0443	S. W. Key,	09117
1	MS-0443	H. S. Morgan,	09117
1	MS-0443	G. D. Sjaardem,	09117
1	MS-0447	S. W. Attaway,	09118
1	MS-0437	J. Jung,	09118
1	MS-0437	R. K. Thomas,	09118
1	MS-0437	L. M. Taylor,	09118
1	MS-0439	W. R. Witkowski,	09234
1	MS-0471	P. F. Chavez,	05135
1	MS-0819	J. S. Peery,	09231
1	MS-0825	B. Hassan,	09115
1	MS-0825	W. H. Rutledge,	09115
1	MS-0826	D. K. Gartling,	09111
1	MS-0826	P. A. Sackinger,	09111
1	MS-0826	R. C. Givler,	09111
1	MS-0826	M. W. Glass,	09111
1	MS-0826	S. N. Kempka,	09111
1	MS-0826	R. R. Rao,	09111
1	MS-0826	P. R. Schunk,	09111
1	MS-0826	J. A. Schutt,	09111
1	MS-0827	A. S. Geller,	09114
1	MS-0828	E. D. Gorham,	09104
1	MS-0828	R. D. Skocypec,	09102
1	MS-0833	J. H. Biffle,	09103
1	MS-0834	A. C. Ratzel,	09112
1	MS-0834	P. L. Hopkins,	09112
1	MS-0834	M. J. Martinez,	09112
1	MS-0835	T. C. Bickel,	09113
1	MS-0835	B. L. Bainbridge,	09113
1	MS-0835	B. F. Blackwell,	09113
1	MS-0835	S. P. Burns,	09113
1	MS-0835	R. J. Cochran,	09113
1	MS-0835	D. Dobranich,	09113
1	MS-0835	S. E. Gianoulakis,	09113

1	MS-0835	D. M. Hensinger,	09113
1	MS-0835	R. E. Hogan,	09113
20	MS-0835	R. R. Lober,	09113
1	MS-0835	V. J. Romero,	09113
1	MS-0835	T. E. Voth,	09113
1	MS-0836	C. W. Peterson,	09116
1	MS-0836	S. R. Tieszen,	09116
1	MS-0841	P. J. Hommert,	09100
1	MS-0865	J. L. Moya,	09735
1	MS-1010	A. L. Ames,	09622
1	MS-1010	M. E. Olson,	09622
1	MS-1111	S. S. Dosanjh,	09221
1	MS-1111	K. D. Devine,	09226
1	MS-1111	B. A. Hendrickson,	09226
1	MS-1111	S. J. Plimpton,	09226
10	MS-1111	C. T. Vaughan,	09226
1	MS-1138	T. L. Edwards,	09432
1	MS-1138	J. R. Hipp,	09432
1	MS-1177	W. J. Bohnhoff,	09661
1	MS-9401	M. L. Callabresi,	08743
1	MS-9018	Central Technical Files,	8940-2
5	MS-0899	Technical Library,	4414
2	MS-0619	Review & Approval Desk,	12690 for DOE/OSTI
3	MS-0161	Patents and Licensing,	11500

External Distribution:

1 Dan Anderson
CAE Systems Department, FORD
MD-10, ECC Building
20000 Rotunda Drive
P.O. Box 2053
Dearborn, MI 48121

2 Bob Benedict & Dave Wismer
D/431A
Goodyear Technical Center
P.O. Box 3531
Akron, OH 44309-3531

1 Dr. Steve Benzley & Research Assts.
Associate Dean, College of
Engineering & Technology
Brigham Young University
Provo, UT 84602

1 Don Dewhirst
Ford Research Laboratory
Ford Motor Company
Room/Mail Drop 2122 SRL
P.O. Box 2053
Dearborn, MI 48121-2053

1 Michael Engelman
Fluid Dynamics International
500 Davis Street, Suite 600
Evanston, IL 60201

1 Lori Frietag
MCS221/C232
Argonne National Laboratory
9700 South Cass Ave
Argonne, IL 60439-4844

1 Tom Canfield
MCS221/B255
Argonne National Laboratory
9400 South Cass Ave
Argonne, IL 60439-4844

1 Rob Kelly
Automesher Development
Ford Motor Company
MD-10 ECC
P.O. Box 2053
Dearborn, MI 48121-2053

1 Paul Kinney
Automesher Development
MD-10 ECC
P.O. Box 2053
Dearborn, MI 48121-2053

1 Malcolm J. Panthaki
7634-C Louisiana Blvd. NE
Albuquerque, NM 87109

1 Eric Hjelmfelt
Altair Computing
1757 Maplelawn Drive
Troy, MI 48084-4004

1 Bob Phillips
PDA Engineering
2975 Redhill Avenue
Costa Mesa, CA 92626

1 Bruce Johnston
The MacNeal-Schwendler Corp.
Aries Division
900 Chelmsford Street
Lowell, MA 01851-5198

1 Harold Trease, MS-B265
Los Alamos National Laboratory
Los Alamos, NM 87545

1 Phil Tuchinsky
Ford Research Laboratory
Ford Motor Company
Room/Mail Drop 2122 SRL
P.O. Box 2053
Dearborn, MI 48121-2053

1 Dennis Vasilopoulos
General Motors R&D
Bldg. 1-6, EM-15
Warren, MI 48090-9057

1 Ted Blacker
Fluid Dynamics International
500 Davis Street, Suite 600
Evanston, IL 60201