NEW APPROACH TO PARALLEL MESH GENERATION AND PARTITIONING PROBLEM*

Nikos Chrisochoides†
College of William and Mary, USA

Abstract In this chapter, we present a new approach for parallel generation and partitioning of 3-dimensional unstructured meshes. The new approach couples the mesh generation and partitioning problems into a single optimization problem. Traditionally these two problems are solved separately with I/O and data movement overheads that exceed 90% of the total execution time for generating, partitioning, and placing very large meshes on distributed memory parallel computers. The new approach minimizes the I/O and data-movement overheads by eliminating redundant memory operations (loads/stores) from and to cache, local & remote memory, and discs. Our preliminary results show that the new approach is nine times faster than the traditional approach for generating, partitioning, and distributing very large 3-dimensional unstructured meshes.

Keywords: Parallel mesh generation, Partitioning, Delaunay triangulation

1. INTRODUCTION

Parallel mesh generation is important for three reasons: (1) *memory constrains* —the memory capacity of high-end workstations does not permit the generation of 3-dimensional unstructured meshes with more than few tens of millions of elements; (2) *execution time* for partitioning (even in parallel) of few tens of millions elements is prohibitively high —recent advancement in parallel PDE solvers and preconditioners shifted the performance bottleneck from the solution phase to the mesh generation and partitioning phases; (3) *scalability of adaptive numerical*

*Dedicated to John R. Rice on the occasion of his 65th birthday.

†This work was supported by NSF Grants: Career Award #CCR-0049086, CISE Challenge # EIA-9726388, and Information Technology Research grant # ACI-0085969, and the IBM Shared University Research Program.

simulations —the I/O overhead between parallel adaptive field solvers and sequential mesh generators is high despite the recent progress in parallel I/O hardware and software technology. For example, it took 19.1 hours to generate and partition[1] a 77 million tetrahedral mesh on DEC8400 with 7.7 Gb memory. It takes only 6 hours on a 256 node Cray T3D for executing 40,000 time steps of a parallel explicit wave propagation code for studying the earthquake-induced dynamics of the San Fernando Valley in South California [1]

Parallel mesh generation codes should: (i) be *efficient and scalable* (ii) be *stable* —parallel meshes should retain the good quality of the elements and properties of the sequentially generated meshes and (iii) *re-use scalar codes* —in order to leverage the ever evolving and maturing core scalar meshing techniques.

In this chapter, we present a new approach for developing parallel mesh generation methods that are stable and efficient. The new approach improves efficiency by simultaneously generating, partitioning, and placing the mesh on the nodes of distributed memory parallel computers. We target parallel computers with large numbers of very fast processors. These parallel computers suffer from very large latency gap between memory (local or remote) and CPU. Hardware studies show [27] that for future Teraflops and Petaflops machines this gap will become even wider. Therefore in order to fully utilize the computational capabilities of these machines we need to develop *pure parallel* approaches that minimize data-movement for memory intensive applications like mesh generation. The new parallel approaches should help minimize or even eliminate unnecessary and expensive memory accesses to and from cache, local/remote memory and minimize load imbalances in the system.

In order to identify opportunities for eliminating or minimizing I/O and overhead due to data movement in the mesh generation process, first, we study the mesh generation in the context of existing parallelization approaches for PDE simulations. Specifically, in Section 2, we identify the weaknesses of the traditional approach for placing (generating, partitioning, and storing) large 3D unstructured meshes on parallel platforms for PDE simulations; and then in Section 3, we examine in detail the steps required to generate 3D unstructured Delaunay meshes. The careful analysis of both mesh generation and partitioning/distribution phases leads to a new approach which we describe in Section 4. The new approach, Simultaneous Mesh Generation and Partitioning (SMGP), is based on the coupling of the mesh generation and partitioning phases.

[1]The I/O time for loading the submeshes onto the 256 nodes of T3D is not included.

In Section 5, we present preliminary data that demonstrate the potential of the SMGP approach; the SMGP approach is up to nine times faster than the traditional mesh generation, partitioning, and placement methods on distributed memory parallel computers. Moreover, our preliminary data indicate that the SMGP approach can generate very good partitions i.e., submeshes with very small surface to volume ratio (i.e., of the order of O(0.01)) and good equi-distribution of elements (i.e., imbalance is of the order of O(0.001)).

2. BACKGROUND

There are two approaches for developing scalable PDE solvers: (A) the *decompose and then discretize* approach and (B) the *discretize and then decompose approach*[2]. The former class of parallel PDE solvers can handle mesh inconsistencies on the inter-subdomain boundary, and thus the parallel mesh generation problem is reduced to a sequential mesh generation problem. However, the *discretize and then decompose* class of parallel PDE solvers use a global mesh which for very large meshes has to be generated in parallel. Specifically the *discretize and then decompose* PDE solvers perform the following steps: (1) discretize the domain Ω using a mesh M, (2) decompose the mesh M into N_s submeshes, M_i, (see Figure 1) such that $M = \cup_i^{N_s} M_i$, (3) use a finite element formulator on each of the submeshes in order to generate in parallel a distributed linear system of equations (see Figure 1, Right), and finally (4) solve the distributed linear system in parallel using an iterative solver —direct solvers are not usually preferred because of *fill*. Figure 1 depicts the partition of the mesh M and the structure of the global algebraic system for a specific indexing[3] of the unknowns.

Parallel iterative solvers (step 4) are computation intensive. Each processor P_i computes the unknowns $\underline{x}_i$ of the linear subsystem $A_i\underline{x}_i = \underline{b}_i$, communicates with other processors each time it needs nonlocal degrees of freedom from the outer-interface nodes, edges or faces (see Figure 1, Right), and synchronizes with the rest of the processors when global data are needed. Thus, the execution time, T_{solver} is given by:

$$T_{solver} = \max_{1 \le i \le \mathbf{P}} \{T_{i,calc} + T_{i,commun} + T_{i,sync}\} \qquad (1)$$

assuming that numerical calculations and network latency overlap. Equation (1) is particularly relevant for the loosely synchronous class of iter-

[2]Professor John Rice suggested this taxonomy in late 80's

[3]For each submesh M_i, first, we index all the nodal degrees of freedom (dof) that correspond to the interior mesh points or edges of M, then we index all the dof on inner interfaces, and at the end we index all the dof on the outer interfaces [2].

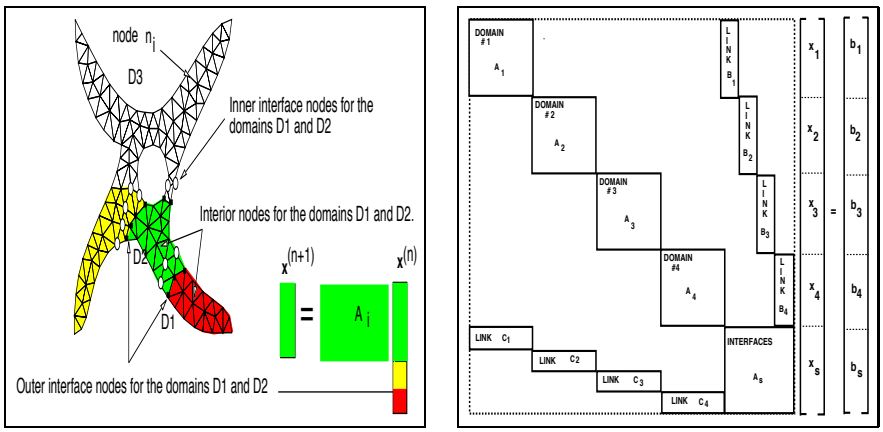

Figure 1 Left) 4-way mesh partition. Right) Global Linear System of Equations that corresponds to a 4-way partition of a mesh

ative solvers which are widely used in large scale PDE simulations. In the loosely synchronous model, computations are carried out in phases. Each phase consists of computations on the local linear subsystem followed by interprocessor communication for nonlocal data [13]. The execution time, T_{solver}, of parallel iterative PDE solvers is minimized when: (1) the processors' workload (calculation and communication) is evenly distributed and (2) communication (of nonlocal unknowns) and synchronization (of global parameters) overheads are minimized.

The problem of finding mesh distributions that minimize T_{solver} is a difficult optimization problem and many partitioning heuristics have been proposed for finding good suboptimal solutions. The partitioning heuristics are classified into two categories: (A) global or direct methods [26, 15] and local or incremental methods [4, 18]. Global or direct heuristics are successful in solving the load-balancing problem, for static PDE computations. However, they are not so efficient for adaptive PDE computations because they require a global knowledge of the mesh-graph (eg. the element-dual graph) which changes any time h-refinement is performed. In addition, some of these methods, at least those based on spectral techniques like Recursive Spectral Bisection (RSB), are sensitive to small perturbations in the graph (characteristic of h-refinement methods), and often lead to heavy data migration [30, 28].

Incremental partitioning methods, on the other hand, start with an initial partition and then iteratively improve it by using profit functions [23] that guide the optimization process. Incremental methods are easy to parallelize and they are scalable. In [10] the RPI group has shown

that incremental methods are very successful in load-balancing the computation of parallel adaptive PDE solvers. The incremental partitioning methods like GGP [4] and PGK [18] have been studied extensively during the last ten years. Today, in software packages such as Chaco [15] and Metis [18], one can find very efficient implementations of incremental methods for the solution of the graph partitioning problem. Table 1 shows that incremental methods based on local optimization techniques are as good as global partitioning methods in terms of equi-distribution and quality of separators. Sequential implementations of both global and incremental methods yield partitions with very good balancing with respect to the number of elements per submesh (equi-distribution). However, the incremental methods independently of the implementation are an order of magnitude faster than the global methods (see Table 2). We adopt, for simultaneously solving the parallel mesh generation and partitioning problem, the basic optimization techniques used in incremental methods. In the next section we describe these techniques.

Table 1 Quality of separators of a global RSB method from Chaco and incremental PGK method from Metis. The quality of separators is measured in terms of the number of faces on the largest separator, for a 16-way partition of 200K and 500K element meshes. The numbers in parenthesis depict another measure, the largest surface to volume ratio of faces among the 16 submeshes.

Mesh Size	RSB	PGK
200K	1836 (.036)	1629 (.032)
500K	3047 (.024)	3246 (.026)

Table 2 Time in seconds for a 16-way partition of 200K and 500K size meshes using RSB from Chaco, PGK, and parallel PGK from Metis and Parallel Metis respectively.

Mesh Size	RSB	PGK	Parallel PGK
200K	65	4	3.2
500K	164	11	3.4

2.1. INCREMENTAL METHODS

A typical formulation of an objective function that represents the computation and communication costs in equation (1), for a parallel

iterative PDE solver over a P-way decomposition of a mesh M can be:

$$OF_{typ} = \max_{1 \le i \le \mathbf{P}} \{W(m(M_i)) + \sum_{M_j \in \kappa_{M_i}} C(m(M_i), m(M_j))\} \qquad (2)$$

where $m : M_i \rightarrow P_i, 1 \le i \le P$ is a function that maps the submeshes M_i to the processors P_i, $W(m(M_i))$ represents the computational load of the processor $m(M_i)$ per iteration, $W(m(M_i))$ is proportional to the number of elements (or degrees of freedom) in M_i, $C(m(M_i), m(M_j))$ represents the communication cost per iteration between the processors $m(M_i)$ and $m(M_j)$, and κ_{M_i} is the set of submeshes that are adjacent to M_i.

OF_{typ} approaches its minimum when the computation load $W(P_i)$ is almost evenly distributed across the processors and the communication cost of the processors is minimum. Clearly, such conditions are also necessary for minimizing T_{solver}. Note that the synchronization term (T_{sync}) of T_{solver} in equation (1) is not explicitly reflected in OF_{typ}. The synchronization cost is a nonlinear function of communication, computation, and overlapped communication and computation. Another term that is not explicitly reflected is the network or bus contention; this term appears implicitly in the communication term, $\sum_{M_j \in \kappa_{M_i}} C(m(M_i), m(M_j))$. The network contention, among other factors, depends on $|\kappa_{M_i}|$, which sometimes is minimized in the effort to minimize the overall communication of the processor $m(M_i)$. Nevertheless, OF_{typ} is considered in the literature to be a good representation for T_{solver}.

Incremental methods for the minimization of OF_{typ} are based on local optimization algorithms. These algorithms search a set of finite perturbations in the solution space (i.e., the space of all mesh partitions) for a given initial solution (i.e., partition) until a perturbation with a lower cost can be found. Examples of such perturbations for the graph partitioning problem appear in the literature [19, 23][4].

A simple procedure for systematically searching the set of perturbations in the solution space is described in [23]. The procedure, in each step of the search, constructs a set of new feasible solutions, t_{new}, from an existing feasible solution t_o. The set of new feasible solutions is called the neighborhood structure of t_o and it is denoted by $N(t_o)$. Subsequently, two feasible solutions t and t' are called neighbors if and only if t' is the result of a finite number of consecutive perturbations (element exchanges or assignments in the case of the graph partitioning) on t. A

[4]These algorithms have a long history, some were discussed in the elementary text Introduction to Computer Science, John R. Rice, 1969 and were analyzed mathematically in the early 1960's by Stanley Reiter.

local optimization algorithm for a given initial solution t_o and neighborhood structure $N(t_o)$ performs a local search of the neighborhood $N(t_o)$, and replaces the current solution t_o with a neighbor solution $u \in N(t_o)$ that minimizes the cost function OF_{typ}. This process is repeated until a *locally optimal* solution has been identified.

The simplest neighborhood structure for the graph partitioning problem and a given feasible solution (i.e., partition (A_o, B_o) of a graph G) is given by $N(A_o, B_o) = \{$ all partitions A^*, B^* that are obtained from the partition A_o, B_o by a single swap operation $\}$. The *swap* or assignment operations for forming A^*, B^* are defined by: $A^* = (A_o \setminus \{a\}) \cup \{b\}$ and $B^* = (B_o \setminus \{b\}) \cup \{a\}$, where a, b are vertices and $a \in A_o$ and $b \in B_o$.

In summary, incremental partitioning heuristics based on local optimization algorithms iteratively improve the partition of the mesh by applying local transformations. Local transformations use element exchanges or assignments of elements near by the inter-submesh interfaces (separators). Local transformations use profit functions [4] to guide the selection process —there are many choices on the subset of the elements which can be considered for an exchange or re-assignment; however, the profit functions can get their maximum values for elements that are in the neighborhood of the mesh separators.

3. PARALLEL MESH GENERATION

Parallel mesh generation methods decompose the original meshing problem into smaller subproblems that can be solved in parallel. In [11] parallel mesh generation methods for the distributed memory model are classified in terms of the *way* and the *order* the artificial boundary surfaces (interfaces) of the subproblems are meshed. Specifically, in [11] the parallel methods are classified into three large categories: (i) methods, like the SMGP approach we present here, that mesh the subproblem interfaces in parallel as they mesh the individual subproblems [7, 3], (ii) a priori methods, that first mesh the interfaces of the subproblems and then mesh in parallel the individual subproblems [31, 24], and (iii) posteriori methods, that first solve the meshing problem in each of the subproblems in parallel and then mesh the interfaces so that the global mesh is consistent [12]. Each of the three categories is further classified in sub-categories based on the means like discrete or continuous representation of the geometry that are used to implement the decomposition of the original problem into subproblems.

3.1. PARALLEL DELAUNAY MESHING

A triangle or tetrahedral mesh, M, is called Delaunay if for each element $e \in M$ the open circumscribed circle or sphere of e is empty i.e., none of the grid points of the mesh are contained within the circumcircle (or circumsphere) triangles (or tetrahedra) of the mesh. This criterion is called the *empty sphere* or *Delaunay* criterion. The Delaunay criterion has been used successfully, for sequential mesh generation of complex geometries, since the late 80's. There are many different Delaunay triangulation methods based on divide-and-conquer and gift-wrapping methods [14]. However, the most popular Delaunay meshing techniques are the incremental methods [14]. Incremental methods start with an initial mesh (usually a boundary conforming mesh) which is refined incrementally by inserting new points (one at a time) using a spatial distribution technique. Each new point is re-connected with the existing points of the mesh in order to form a new triangulation or a new mesh. The difference between the various Delaunay incremental algorithms is due to: (1) different spatial point distribution methods for creating the points and (2) different local re-connection techniques for creating the triangles or tetrahedra.

The most popular local re-connection methods are the flip edge/face methods [20] and the BW kernel [2, 29]. The flip edge/face methods are easier to implement on single CPU computers using simple and efficient data structures. However, the disadvantage of the flip methods is that the code complexity of the parallelization is increasing and performance is decreasing. In [22] we show that setbacks —due to the undoing (or rollback) of face flipping from more than one processors— can increase from 7% of the total number of updates on 2 processors to 29% on 16 processors for a mesh containing one million tetrahedra. In contrast, the BW kernel is easier and more efficient to parallelize.

3.1.1 Bowyer-Watson kernel. The BW kernel is an iterative procedure; each iteration performs two basic operations: (i) *point insertion*, where a new point is inserted using an appropriate spatial distribution technique; and (ii) *element creation*, where existing triangles that violate the empty sphere criterion are removed and new triangles are built by connecting the newly inserted point with old points.

Given an existing Delaunay triangulation, T_i, a new triangulation, T_{i+1}, can be incrementally constructed by inserting a new point into T_i and recovering the Delaunay property of the triangulation through local transformations. This process can be viewed as an iterative procedure, $for\ i = 0, N\ do: T_{i+1} = T_i - C_i + B_i$, where C_i is defined as the the

union of all tetrahedra $t \in T_i$ whose circumsphere encloses the new point p_i and ball B_i is defined to be the set of all new tetrahedra that include p_i as a vertex. This algorithm is described bellow:

1 **point insertion**: the creation of a new point, $p_{i+1} \notin T_i$,

2 **point location**: the identification of the tetrahedron, $t \in T_i$ containing p_{i+1},

3 **cavity construction**: the computation of the set, C_i, of tetrahedra $t \in T_i$ that violate the Delaunay property

4 **cavity re-triangulation**: the deletion of the tetrahedra in C_i from T_i, and the creation of the ball B_i.

Figure 2 depicts a single iteration of the BW kernel, starting from an initial boundary conforming mesh.

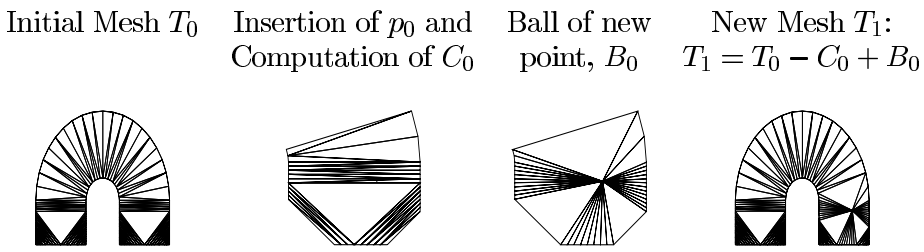

Figure 2 Single iteration from Bowyer-Watson algorithm.

The efficiency of the parallel BW kernel depends on the efficiency of the searching procedure for finding the first triangle in conflict (i.e., triangle that violates the Delaunay property) for each new point insertion. Also, it depends on the efficiency of the cavity computation and element creation and distribution (or allocation). In this chapter we concern about the efficiency of the parallel implementation and element creation and distribution steps.

Parallelizing the Bowyer-Watson kernel. The BW kernel, for each new point insertion, performs purely local transformations on an existing Delaunay triangulation to maintain locally the Delaunay property. It is therefore possible to apply the BW kernel concurrently in many areas of the mesh, without disrupting the *global* Delaunay property of the triangulation [9].

Without lost of generality in our approach we can assume that the input to the parallel BW kernel is T_0, an initial Delaunay tetrahedraliza-

tion of a set of points, $S \in E^3$. We also assume that T_0 has been partitioned into N_s submeshes $M_k \subseteq T_i, k = 1, 2, 3, \ldots, N_s$. When $N_s >> P$, P is the number of processors, we have an *over-decomposition*; this case is especially attractive when the runtime system implicitly balances the processors' workloads. In the rest of this chapter we assume, for simplicity, that $N_s = P$.

The decomposition of T_0 into submeshes induces a separator, $I_{j,k}$, between submeshes M_j and M_k if there exists at least one common face between them. $I_{j,k}$ consists of triangular *interface faces*, edges, and vertices, which are replicated in all submeshes sharing them. Two submeshes M_j and M_k are called *adjacent* if $I_{j,k} \neq \emptyset$ and are called *neighbors* if $M_k \cap M_j \neq \emptyset$.

New points are inserted concurrently into submeshes and the resulting cavities are computed and re-triangulated independently. When tetrahedra in one of the submeshes are non-Delaunay with respect to a point insertion in another submesh, the cavity may extend across the interfaces between these two submeshes (an *interface* cavity); otherwise, the cavity is *local*, and is constructed and re-triangulated "atomically," without the intervention of other processors that handle adjacent submeshes. Figure 3 depicts an interface mesh.

A simple method for computing a cavity, C_p, is to perform a depth-first search over the data structures to find the tetrahedra that are non-Delaunay (the *encroached* tetrahedra, see Figure 3, Left) with respect to the point p, starting from some initial encroached tetrahedron. In the parallel BW kernel, the construction (or *expansion*) of an interface cavity is distributed across possibly many submeshes. A more detailed description is given in [9]

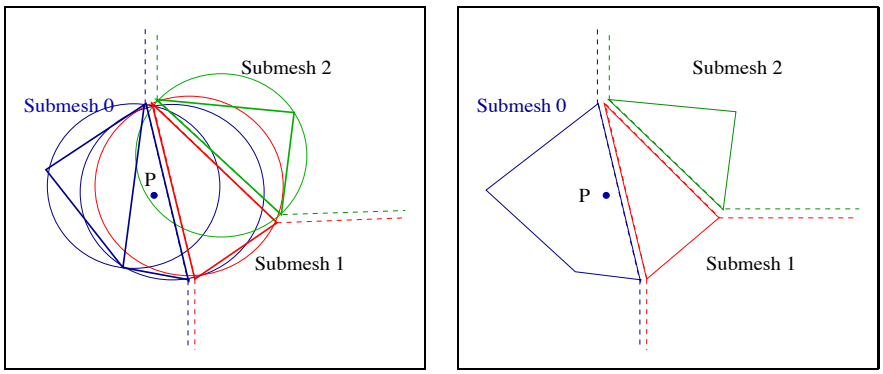

Figure 3 Cavity expansion over more than one submeshes across the inter-submesh interfaces. Dashed line show that inter-submesh interfaces.

4. SIMULTANEOUS MESHING AND PARTITIONING

Traditionally, mesh partitioners decompose the mesh into P submeshes such that the following two criteria are approximately satisfied:

(i) *equi-distribution*: minimize maximum difference in the number of elements between the submeshes,

$$\min \max_{1 \leq i,j \leq P} ||M_i| - |M_j|| \tag{3}$$

(ii) *quality of the separators*: minimize the surface to volume ratio of tetrahedral faces for each submesh,

$$\min_{1 \leq i \leq P} \frac{|S_i|}{|V_i|} \tag{4}$$

where $|M_i|$ is the number of elements of the submesh M_i, $|S_i|$ is the number of faces on the interfaces of the submesh M_i, and $|V_i|$ is the total number of faces of the submesh M_i. Next, we describe the distributed element creation step of the parallel mesh generation process so that these two criteria are satisfied while the submeshes are generated concurrently.

In Section 2, we have seen that local optimization methods apply transformations locally on a small subset of elements at a time. Also, we have seen that there is significant freedom on the choice of the subset of elements that are selected to be assigned to different submeshes; the criterion (ii) is satisfied as long as these elements are in the neighborhood of the existing mesh separators. Similarly, in Section 3 we have seen that the BW kernel for Delaunay triangulation applies local transformations on a small subset of elements (the cavity of a newly inserted point). These transformations are applied in order to improve the mesh quality while preserving the Delaunay properties of the mesh. The basic idea of simultaneous mesh generation and partitioning is *to distribute the newly created elements from the re-triangulation of the interface cavities in a way that the criteria (i) and (ii) are optimize and data migration of elements is minimized.*

From the definition of the interface cavity, the newly created elements are in the neighborhood of the mesh separators and thus are very good candidates for new assignment —in the sense of incremental partitioning methods. The newly created elements involve the most recently accessed data. Their data structures are still in cache or memory, so by deciding at creation time which processor should "own" these new

elements, we minimize both local/remote memory and disk access operations. This will have a lasting impact in the performance of parallel meshing as long as we prevent future migrations of elements due to load imbalances. Next, we describe a prediction method, for the final number of elements per processor, that is used to prevent imbalances and thus data migration. The prediction method is using information on the current state of the mesh (i.e., average quality of existing elements, number of "bad" elements to be removed, etc.) and the pre-defined final quality of elements.

4.1. EQUI-DISTRIBUTION

Instead of waiting to complete the mesh generation phase and then re-distribute the newly created elements, we propose to re-distribute the new elements as they are created. It is important to re-distribute only the elements that are generated from interface cavities, since these elements are "close" to separators of the partitions and maximize the profit functions of incremental partitioning algorithms —element re-distribution from local cavities will result into disconnected submeshes and it will increase the size of the separators or the surface to volume ratio. The simplest option for re-distributing elements from interface cavities is to place the new elements on the processor that is responsible for their creation. Intuitively, and as the performance data suggest, some of the processors may end up with many more elements, depending on the initial distribution of the geometry (i.e., number of elements with large circumradius-to-shortest edge ratio). This approach leads into workload imbalances that subsequently affect the performance of both the parallel mesh generation and the solution phases, if re-partitioning is not applied. Of course, re-partitioning will introduce undesired overhead due to global synchronization and data-movement.

A simple solution to this problem is to continuously re-distribute the new elements as they are created among the processors that are participating in the creation of interface cavities; the processor with the smallest number of elements at the time of element creation is a good candidate to receive the new elements. An example of interface cavity expansion is shown in Figure 4. The cavity is expanded into submeshes 0, 1, and 2, then the cavity is re-triangulated, and the new elements are redistributed to the participating submeshes. As it is expected, this approach leads to unnecessary movement of elements among the processors, because the number of elements on the processors is changing (in a non-monotonic way which depends on the current quality of elements) during the parallel mesh generation. The temporal changes in the size

of the submeshes will trigger continuous element migration between the submeshes in order to continuously maintain processors' workload balances.

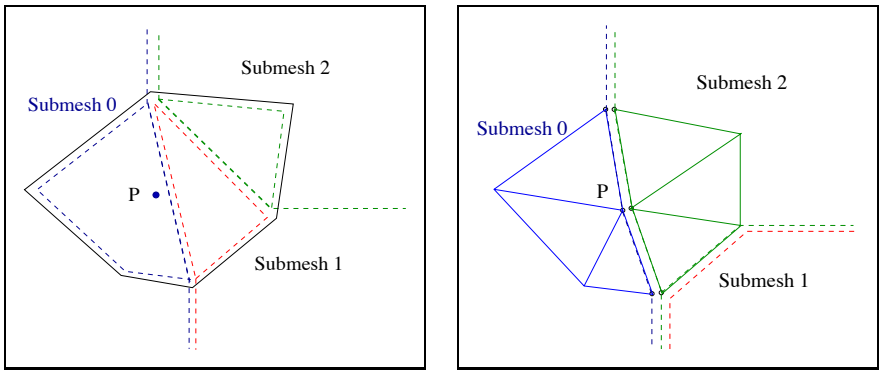

Figure 4 An interface cavity is expanded into 3 submeshes, re-triangulated, and redistributed among the participating submeshes.

We try to minimize the unnecessary data-movement by considering the final number of elements per processor in our decision making on the allocation of new elements. However, since we do not know the final number of elements per processor we use a prediction formula to approximate it. The prediction formula uses information on the current "state" of the mesh: (1) the current number of elements, (2) the current number of "bad" elements, and (3) the current mesh quality for each of the processors participating in the creation of an interface cavity —the number of participating processors per interface cavity is small. It is inexpensive for each processor to maintain (locally) information about the current state of the processors that handle adjacent or neighboring submeshes. The information on the current state of the mesh is captured in a simple formula that at any time approximates the final number of elements in each of the submeshes:

$$N_{predicted} = N_{elms} + (\frac{c_{ne/c} - c_{oe/c}}{c_{b/c}}) \times N_{bad\_elms} \times w(q_{cur}, q_{targ}) \qquad (5)$$

where $c_{ne/c}, c_{oe/c}$, and $c_{b/c}$ are constants; $c_{ne/c}$ is the average number of new elements that are created per cavity, $c_{oe/c}$ is the average number of old elements per cavity that are removed, and $c_{b/c}$ is the average number of "bad" elements per cavity that are destroyed. The constants q_{cur}, q_{targ} are the average current and target mesh quality for a given submesh. The $N_{predicted}$ is computed in parallel for each submesh.

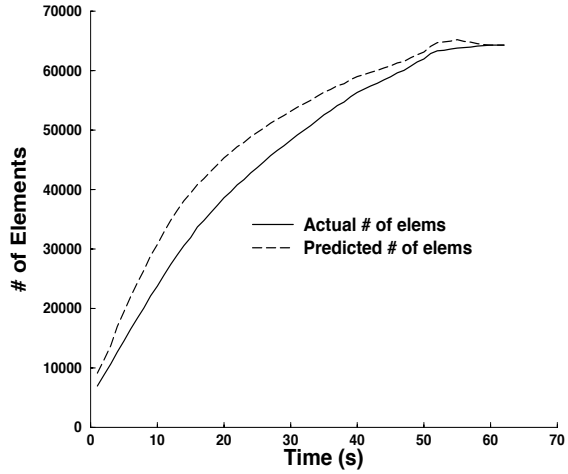

Figure 5 Actual and predicted number of elements versus time for one region of a 16 region mesh.

Figure 5 tracks the progress of a prediction formula of equation (5) in a single submesh over the mesh generation course. The output of the prediction function at any instance in time is the number of elements expected in the *next* instance. This graph shows that the prediction function we use is reasonably accurate and does not vary widely from the actual number of elements over the course of the mesh generation.

5. PERFORMANCE EVALUATION

We use as Model Problem, Ω, a unit cube, within which is suspended a regular octahedral hole centered at the centroid of the cube. The vertices of the octahedron are positioned .25 units from the cube's centroid along the perpendicular bisectors of the cube's faces, such that vertices along the same bisector are .5 units apart. We use Ω for the evaluation of both the traditional approach, based upon partitioning algorithms implemented by libraries like Chaco and Metis, and the SMGP approach. The domain Ω is discretized with three different meshes of 200,000, 500,000, and 1,000,000 elements each. The sequential mesh was generated on a Wide SP node. A Wide SP node is a 135MHz Power2 SuperChip (P2SC) with 2 GB memory, 128K cache, and 256bit memory bus. Parallel Metis (ParMetis) and SMGP were run on 16 Thin SP nodes. A Thin SP node is a 120MHz P2SC with 1GB memory, 128K cache, 256bit memory bus. The Thin SP nodes are connected via the SP switch; the SP switch is a TB3 switching fabric with 150MB/s peak

hardware bandwidth. The SP machine was located at Cornell Theory Center.

The meshes were generated both sequentially and in parallel using the following two quality criteria: (1) minimize the maximum element volume, and (2) minimize the largest element circumradius-to-shortest-edge ratio (AR). The maximum element volume criterion controls the size of the mesh, while the AR bounds the "size" of the worst element in the mesh.

Next, we present performance data from traditional state-of-the-art libraries for partitioning medium size meshes on the nodes of distributed memory parallel machines like the SP. Then we present data from four different versions of the SMGP algorithm and compare them with ParMetis —a widely used parallel graph partitioning package.

5.1. TRADITIONAL APPROACH

Our goal is to deliver finite element meshes for parallel PDE solvers. The meshes should be stored on the processors in a format that is suitable for parallel finite element PDE formulators. In our evaluation we include the time it takes to generate, deliver, and place the actual elements of the submeshes on the processors in a way that they can be used without any further processing by the PDE solvers. The reason we insist on this evaluation is because a large percent of the total simulation time is spent in delivering the elements to the processors, for parallel finite element formulators.

Chaco and Metis are two well known libraries that are used routinely by the scientific community for mesh partitioning. Tables 3 and 4, for the Model Problem Ω, verify that both Chaco and Metis work well in terms of both mesh equi-distribution and quality of separators. We do not have data for a million element mesh for Chaco and Metis, because of memory limitations imposed by the memory requirements of the sequential mesh generation, translation of mesh data structures to CSR format, and partitioning libraries.

Table 5, indicates that ParMetis generates partitions which do not distribute the elements as evenly as the partitions generated by Metis. Of course, small imbalances, are unlikely to have a major impact in the overall performance of the parallel PDE solver. However, in the case of ParMetis we observe a trend, the load imbalance doubles as the mesh size doubles. For meshes of hundreds of millions of elements such imbalances are expected to have a negative impact on the performance of the parallel PDE solvers.

Table 3 Equi-distribution criterion measured with respect to the difference in the number of elements between the largest and the smallest submesh, for a 16-way partition.

Mesh Size	Chaco (MKL)	Metis	ParMetis
200K	0	5	316
500K	1	4	623
1000K	—	—	1298

Table 4 Average size of separators measured by the number of interface faces per submesh for a 16-way partition.

Mesh Size	Chaco (MKL)	Metis	ParMetis
200K	1366	1366	1343
500K	2623	2587	2538
1000K	—	—	4105

Table 5 Equi-distribution criterion for ParMetis. Again, it is measured with respect to the difference in the number of elements between the largest and the smallest submesh for a 16-way partition.

Mesh Size	ParMetis
25,000	50
50,000	123
100,000	202
250,000	348
500,000	623
1,000,000	1298
2,000,000	2009

The total execution time of the traditional approach for generating, partitioning, and placing unstructured meshes onto the nodes of a parallel machine (in a form that can be used by parallel finite element formulators) is equal to the sum of the execution times: (1) for generating the mesh sequentially or in parallel, (2) for transforming the mesh data structures into the CSR[5] format that generic graph partitioning libraries like Chaco and Metis require, (3) for loading the CSR data on processors, (4) for partitioning in parallel the mesh graph, (5) for migrating the element as it is dictated by partitioners, and (6) for generating the submeshes on the processors in way that they can be used by FE formulators. The total execution time for all these phases using Chaco, Metis, and ParMetis is shown in Table 6.

Table 6 Total execution time in seconds for generation, partitioning, and loading a mesh onto 16 nodes of an IBM SP/2.

Mesh Size	Chaco (MKL)	Metis	ParMetis
200K	110	97	90
500K	277	237	215
1000K	—	—	439

Table 7 shows the percentage of the overhead for I/O (reading and writing the mesh to be partitioned) and data movement in the case of ParMetis. ParMetis uses an initial partition which is further optimized in parallel with respect to a number of criteria [25] similar to ones we have listed in Section 4. For example, ParMetis in order to optimize the initial partitions, for the 200K, 500K, and 1000K meshes, performs data-movement of more than 191K, 473K, and 934K elements, respectively —almost all of the elements were re-assigned.

In summary, the traditional approach for generating, partitioning, and placing very large meshes in an optimal way (w.r.t criteria i and ii, Section 4) has two weaknesses:

1 I/O and data movement due to re-partitioning is prohibitively expensive, for very large simulations. Despite the progress in parallel I/O hardware and software technologies the I/O for loading

[5]The CSR format is a simple and widely used format for efficiently storing sparse graphs. The structure of a graph, **G**, is represented with two arrays: (a) the *adjacency* array, **E**, and (b) the *range* array, **N**. For a graph with v vertices and m edges, $|\mathbf{E}| = 2m$, and $|\mathbf{N}| = v + 1$. **N** maps the nodes $n_i \in \mathbf{G}$ to the range $\mathbf{E}[\mathbf{N}_i, \mathbf{N}_{i+1}]$, which contains the indices of the nodes adjacent to n_i. $|\mathbf{E}| = 2m$ since, for nodes n_i and n_j, both edge (n_i, n_j) and edge (n_j, n_i) are stored.

Table 7 Percentage of I/O required to generate, partition and load the mesh for the PDE solver. In the case of ParMetis the percentage includes the time for data movement due to partitioning.

Mesh Size	Chaco (MKL)	Metis	ParMetis
200K	84 %	96 %	97 %
500K	82 %	95 %	98 %
1000K	—	—	98 %

hundreds of millions to a billion element mesh will remain a challenging problem. It needs to be addressed algorithmically in order to achieve scalability and utilize future Teraflops and Petaflops supercomputers.

2 The solution of the equi-distribution problem using parallel partitioners is expensive; there is a trade-off between the quality of the resulting partition and the performance of the partitioners. The compromise in equi-distribution is not noticeable for small meshes, but for very large problems, and for the extremely fast processors of the near future, such a trade-off will affect the overall performance of the parallel PDE simulations[6].

5.2. SMGP APPROACH

Next, we present preliminary data that compare the SMGP approach with the traditional approach using the best known parallel partitioner (ParMetis). Also, we present data from four SMGP algorithms. The first algorithm, SMGP0, generates and partitions the mesh in a very efficient way (Table 8); it is five times faster than the traditional approach. However, it does nothing to improve the element distribution (Table 9) and the quality of the separators (Table 10). New elements remain where they are created. Although SMGP0 is very efficient it suffers from severe imbalances which in turn affect its performance, since some of the processors spend more time than others on parallel meshing. Of course, a separate mesh re-partitioning phase can correct the imbalancing problems, but again such a re-partitioning phase will introduce synchronization and data movement overheads.

[6]Unless implicit methods are used to address this problem [8] during the mesh generation and PDE solution phase.

Table 8 Total execution time in seconds on 16 processors. The total execution time includes both data redistribution and SMGP times.

Mesh Size	ParMetis	SMGP0	SMGP1	SMGP2	SMGP3
200K	90	42	42	42	42
500K	215	65	87	64	62
1000K	439	97	160	91	94

Table 9 Balance measure as the difference between the submesh with the largest number of elements and the submesh with smallest number of elements, for a 16-way partition.

Mesh Size	ParMetis	SMGP0	SMGP1	SMGP2	SMGP3
200K	316	2937	60	2177	2090
500K	623	11845	181	3830	2065
1000K	1298	30632	96	11485	7293

Table 10 Maximum size of separators measured in terms of number of interface faces, for a 16-way partition.

Mesh Size	ParMetis	SMGP0	SMGP1	SMGP2	SMGP3
200K	1343	2136	6933	1774	1822
500K	2538	5692	28739	3482	3561
1000K	4105	10224	64518	5570	5785

The second algorithm, SMGP1, uses the prediction formula of equation (5) to estimate the final number of elements per submesh. Each processor uses the prediction formula in order to decide *when* and *where* to distribute the newly created elements. Table 9 indicates that SMGP1 reduces imbalances by more than ten times, for large meshes, compared to the traditional approach and it is still more than two times faster than the traditional approach. SMGP1 like SMGP0, does nothing to improve the quality of the separators and therefore it does not return good quality partitions in terms of the separator size (see Table 10), as it is expected. A post-processing for the optimization of the separators (using local exchange optimization methods like GGP [6]) needs to be applied in order to minimize the communication overheads during the parallel mesh and PDE solution phases.

Table 11 Ratio of number of moved elements to mesh size for a 16-way partition.

Mesh Size	ParMetis	SMGP1	SMGP2	SMGP3
200K	.96	.03	.07	.08
500K	.95	.38	.14	.12
1000K	.93	.52	.14	.12

The third algorithm, SMGP2, uses the equi-distribution scheme from SMGP1 and a simplified version of the Kernighan-Lin [19] algorithm to improve the quality of the mesh separators. The Kernighan-Lin (KL) algorithm improves the separators of a given graph (element-dual graph) by considering sequences of vertex *swaps* between partitions of the graph. A vertex of the graph may be swapped from one partition to another if the *gain* (measured in terms of improvement on the size of the separators) in performing the swap is larger than the gain of swapping other candidate vertices. The SMGP2 algorithm uses a restricted form of the KL algorithm, where only a single sequence of swaps is considered and only elements of interface cavities can be candidates for swapping. SMGP2 generates partitions whose average size of separators is close to the average size of separators generated by ParMetis —Table 10 depicts the maximum size of separators. The simplifications and restrictions we impose on SMGP2 algorithm, for performance reasons, prevent SMGP2 from improving the separators further. Moreover, SMGP2 like KL algorithm, is local in nature and it can not compete in terms of the quality of the solution (partition) with the more powerful multi-level partitioning algorithms [16] used by Chaco and ParMetis.

The last algorithm, SMGP3, tries to combine the effectiveness of SMGP1 on the equi-distribution problem while minimizing data movement and the effectiveness of SMGP2 on the reduction of separator sizes. SMGP3 generates submeshes whose separators are good compared to separators generated by ParMetis. Moreover SMGP3 is two to three times faster than the traditional approach. SMGP3 and SMGP1 have the same time complexity, but the constant term for SMGP3 is much larger because of the local transformation for improving the quality of the separators. However, SMGP3 performs 75% less data-movement than SMGP1 (see Table 11) and thus minimizes communication overhead. SMGP3 uses a weighted profit function that combines the profit functions from SMGP1 and SMGP2.

The development and evaluation of the SMGP3 algorithm has not been completed yet. SMGP3 does not completely capture the effec-

Table 12 Execution time is the time in seconds that it takes to place a two million element mesh on the 16 nodes of an SP machine, the balance measure the equi-distribution of the submeshes which is measured as the difference between the smaller and larger submesh, and the surface to volume ration (S/V) is used to quantify the quality of the separators for a 16-way partition generated by ParMetis, m-SMGP and SMGP3 methods.

Statistics	ParMetis	m-SMGP3	SMGP3
Execution Time	1232	168	135
Balance	2009	1529	21053
S/V Ratio	0.0387	0.0332	0.0618

tiveness of either SMGP1 or SMGP2 in terms of equi-distribution and quality of separators, respectively. SMGP3 reduces imbalance by four times compared to SMGP0 and lowers the size of separators by half compared to SMGP0, while at the same time is as slightly faster than SMGP0. SMGP3 reduces the data-movement overheads by more than four times (see Table 11) compared to SMGP1 and more than 7.5 times compared to traditional approach.

6. CONCLUSIONS AND FUTURE WORK

We have presented a new approach, Simultaneous Mesh Generation and Partitioning, for solving the generation, partitioning, and distribution problems of unstructured guaranteed quality Delaunay meshes on parallel platforms. The coupling of these problems maximizes processor and memory utilization by eliminating unnecessary and expensive access operations to and from cache, local/remote memory, and discs. Our preliminary results suggest that the SMGP approach is nine times faster than the traditional approach for parallel mesh generation and partitioning problem, while at the same time it generates and maintains an optimal distribution of the data structures (mesh) for parallel adaptive PDE solvers. Finally, the SMGP approach is leads to stable parallel mesh generation methods, while the traditional approaches impose "hard" bounds on the quality of the final mesh and thus do not maintain the same quality as the sequential mesh generators. For example, the quality of a mesh obtained by parallel refinement of coarse submeshes, using some form of constrained meshing [3, 24], will be limited by "small" angles that appear in the interfaces of the coarse submeshes. SMGP eliminates these problems.

Future Work. The SMGP approach needs more work in order to be: *ten times faster than the the traditional way of generating and maintaining the distributed data structure for parallel adaptive PDE solvers.* The performance data from Section 5 indicate that this goal is feasible. Table 12 suggests that by applying graph contraction techniques [21] similar to multi-level partitioning methods [16, 17] we can simultaneously generate, partition, and place the submeshes on the processors by seven times faster than the traditional approaches. And at the same time we can improve the equi-distribution by about 25% and achieve better quality of separators compared to the traditional partitioning methods. Our next objective is to achieve the *ten-fold improvement in performance and preserve the quality of partitions by tightly coupling the graph contraction (m-SMGP) techniques with mesh generation process.*

Acknowledgments

Many colleagues and friends helped us in many different ways during the time we were developing these ideas, we are thankful to all of them. The idea of considering the simultaneous mesh generation and partition of meshes initially surfaced in our discussions with Geoffrey Fox in early 90's. Tim Baker gave us the model problem Ω and an initial Delaunay triangulation. My colleagues and students from Cornell and Notre Dame, especially Induprakas Kodukula, Chris Hawblitzel, and Kevin Barker with whom we've build the software infrastructure that helped us ease the task of implementing parallel mesh generation codes. My students Florian Sukup and Demian Nave for implementing the parallel mesh generation codes and collecting the performance data we present in Section 5. IBM's Research Program and Marc Snir for helping us to acquire an extremely useful, for this project, SP machine. Keshav Pingali and Paul Stodghil for their continuous support and help to integrate our parallel meshing work in the crack propagation test-bed at Cornell.

References

[1] H. Bao, J. Bielak, O. Ghattas, L. F. Kallivokas, D. R. O'Hallaron, J. R. Shewchuk, and J. Xu. Earthquake ground motion modeling on parallel computers. *Supercomputing '96*, Pittsburgh, PA, November 1996.

[2] A. Bowyer. Computing Dirichlet tessellations. *The Computer Journal*, 24(2):162–166, 1981.

[3] P. Chew, N. Chrisochoides, and F. Sukup. Parallel constrained Delaunay meshing. In *Proceedings of the 1997 ASME/ASCE/SES Summer Meeting, Special Symposium on Trends in Unstructured Mesh Generation*, Northwestern University, Evanston, IL, June

29–July 2, 1997.

[4] N. P. Chrisochoides, C. E. Houstis, S. K. Kortesis E. N. Houstis, and J. R. Rice. Automatic load balanced partitioning strategies for PDE computations. In (E. N. Houstis and D. Gannon, eds.), *Proceedings of International Conference on Supercomputing*, ACM Press, pp. 99–107, 1989.

[5] N. Chrisochoides, E. Houstis, S. B. Kim, M. Samartzis, and J. Rice. Parallel iterative methods. *Advances in Computer Methods for Partial Differential Equations VII*, (R. Vichnevetsky. D. Knight and G. Richter, eds.), IMACS, New Brunswick, NJ, pp. 134–141, 1992.

[6] N. P. Chrisochoides, E. N. Houstis and J. J. Rice. Mapping algorithms and software environment for data parallel pde iterative solvers. *Special Issue of the Journal of Parallel and Distributed Computing on Data-Parallel Algorithms and Programming*, 21(1):75–95, April 1994.

[7] N. Chrisochoides and F. Sukup. Task parallel implementation of the Bowyer-Watson algorithm. *Proceedings of Fifth International Conference on Numerical Grid Generation in Computational Fluid Dynamics and Related Fields*, pp. 773–782, 1996.

[8] N. Chrisochoides, K. Barker, D. Nave, and C. Hawblitzel. Mobile object layer: a runtime substrate for parallel adaptive and irregular computations. *Advances in Engineering Software*, 31(8-9):621–637, August 2000.

[9] N. Chrisochoides Parallel Delauany triangulation kernel. To be submitted in the *Journal of Parallel and Distributed Computing*, 2001.

[10] H. L. de Cougny, K. D. Devine, J. E. Flahery, R. M. Loy, C. Ozturan, and M. Shephard. Load balancing for parallel adaptive solutions of partial differential equations. *Appl. Numer. Math.*, 16(1-2):157–182, 1994.

[11] H. L. de Cougny and M. Shephard. *Parallel Unstructured Grid Generation.* CRC Handbook of Grid Generation, (J. F. Thompson, B. K. Soni, and N. P. Weatherill, eds.), CRC Press, Inc., Boca Raton, pp. 24.1–24.18, 1999.

[12] H. L. de Coughy and M. Shephard. Parallel volume meshing using face removals and hierarchical repartitioning. *Comput. Methods Appl. Mech. Engrg.*, 177:275–298, 1999.

[13] G. C. Fox, M. Johnson, G. Lyzenga, S. Otto, J. Salmon, and D. Walker. *Solving Problems on Concurrent Processors*. Prentice Hall, New Jersey, 1988.

[14] P. Frey and P. George. *Mesh Generation: Applications to Finite Elements*. Hermes Science Publishing, pp. 814, 2000.

[15] B. Hendrickson and R. Leland. The Chaco User's Guide, version 1.0. Technical Report SAND93-2339, Sandia National Laboratories, 1993.

[16] B. Hendrickson and R. Leland. A multilevel algorithm for partitioning graphs. SAND93-1301, 1993.

[17] G. Karypis and V. Kumar. Multi-level k-way: Partitioning Scheme for Irregular Graphs. *Journal of Parallel and Distributed Computing*, 48:71–95, 1998.

[18] G. Karypis and V. Kumar. A fast and highly quality multilevel scheme for partitioning irregular graphs. *SIAM Journal on Scientific Computing*, to appear.

[19] B. W. Kernighan and S. Lin. An efficient heuristic procedure for partitioning graphs. *The Bell System Technical Journal*, pp. 291–307, 1970.

[20] C. Lawson Transforming triangulations. *Discrete Mathematics*, 3:365–372, 1972.

[21] N. Mansour, R. Ponnusamy, A. Choudhary, and G. Fox. *Graph Contraction for Physical Optimization Methods: A Quality-Cost Tradeoff for Mapping Data on Parallel Computers*. International Supercomputing Conference, Japan, July 1993, ACM Press.

[22] D. Nave and N. Chrisochoides A local reconnection scheme for parallel Delaunay mesh generation, *Trends in Unstructured Mesh Generation*, Michigan, August 2001.

[23] C. H. Papadimitriou and K. Steiglitz. *Combinatorial Optimization: Algorithms and Complexity*. Prentice-Hall, Englewood Cliffs, NJ, 1982.

[24] R. Said, N. Weatherill, K. Morgan, and N. Verhoeven. Distributed parallel Delauany mesh generation. *Comp. Methods Appl. Mech. Engrg.*, 177:109–125, 1999.

[25] K. Schloegel, G. Karypis, and V. Kuma. Parallel multilevel diffusion schemes for repartitioning of adaptive meshes. Technical Report 97-014, University of Minnesota, 1997.

[26] H. D. Simon. Partitioning of unstructured problems for parallel processing. Technical Report RNR-91-008, NASA Ames Research Center, Moffet Field, CA, 1990.

[27] T. Sterling. *Petaflops Systems Operations Working Review.* Bodega Bay, CA, June 1998.

[28] C. Walshaw and M. Berzins. Dynamic load balancing for PDE solvers an adaptive unstructured meshes. University of Leeds, School of Computer Studies, Report 92.32, 1992.

[29] D. Watson. Computing the n-dimensional Delaunay tessellation with applications to Vornoi polytopes. *The Computer Journal*, 24(2):167–172, 1981.

[30] R. D. Williams. Performance of dynamic load balancing algorithms for unstructured mesh calculations. *Concurrency Practice and Experience*, 3(5):457-481, 1991.

[31] P. Wu and E. Houstis. Parallel adaptive mesh generation and decomposition. *Engrg. Comput.*, 12:155–176, 1996.