Dynamic Load Balancing for Parallel Adaptive Mesh Refinement*

Xiang-Yang Li[1] and Shang-Hua Teng[1]

Department of Computer Science and Center for Simulation of Advanced Rockets, University of Illinois at Urbana-Champaign, Urbana, IL 61801.

Abstract. Adaptive mesh refinement is a key problem in large-scale numerical calculations. The need of adaptive mesh refinement could introduce load imbalance among processors, where the load measures the amount of work required by refinement itself as well as by numerical calculations thereafter. We present a dynamic load balancing algorithm to ensure that the work load are balanced while the communication overhead is minimized. The main ingredient of our method is a technique for the estimation of the size and the element distribution of the refined mesh before we actually generate the refined mesh. Base on this estimation, we can reduce the dynamic load balancing problem to a collection of static partitioning problems, one for each processor. In parallel each processor could then locally apply a static partitioning algorithm to generate the basic units of submeshes for load rebalancing. We then model the communication cost of moving submeshes by a condensed and much smaller subdomain graph, and apply a static partitioning algorithm to generate the final partition.

1 Introduction

Many problems in computational science and engineering are based on unstructured meshes in two or three dimensions. An essential step in numerical simulation is to find a proper discretization of a continuous domain. This is the problem of *mesh generation* [1, 9], which is a key component in computer simulation of physical and engineering problems. The six basic steps usually used to conduct a numerical simulation: **1: Mathematical modeling**, **2: Geometric modeling**, **3: Mesh generation**, **4: Numerical approximation**, **5: Numerical solution**, **6: Adaptive refinement**.

In this paper, we consider issues and algorithms for adaptive mesh refinement, (cf, Step 6 in the paradigm above). The general scenario is the following. We partition well shpaed mesh M_0 and its numerical system and map the submeshes and their fraction of the numerical system onto a parallel machine. By solving the numerical system in parallel, we obtain an initial numerical solution S_0. An error-estimation of S_0 generates a refinement spacing-function h_1 over the domain. Therefore, we need properly refine M_0 according to h_1 to generate well shaped mesh M_1. The requirement of refinement introduces load imbalance among processors in the parallel

* Supported in part by the Academic Strategic Alliances Program (ASCI) of U.S. Department of Energy, and by an NSF CAREER award (CCR-9502540) and an Alfred P. Sloan Research Fellowship.

machine. The work-load of a processor is determined by refining its submesh and solving its fraction of the numerical system over the refined mesh M_1. In this paper, we present a dynamic load balancing algorithm to ensure that the computation at each stage of the refinement is balanced and optimized. Our algorithm estimates the size and distribution of M_1 before it is actually generated. Based on this estimation, we can compute a quality partition of M_1 before we generate it. The partition of M_1 can be projected back to M_0, which divides the submesh on each processor into one or more subsubmeshes. Our algorithm first moves these subsubmeshes to proper processors before performing the refinement. This is more efficient than moving M_1 because M_0 is usually smaller than M_1. In partitioning M_1, we take into account of the communication cost of moving these subsubmeshes as well as the communication cost in solving the numerical system over M_1.

This paper is organized as following. Section 2 introduces an abstract problem to model parallel adaptive mesh refinement. Section 3 presents an algorithm to estimate the size and distribution of the refined mesh before its generation. It also presents a technique to reduce dynamic load balancing for mesh refinement to a collection of static partitioning problems. Section 4 extends ours algorithm from the abstract problem to unstructured meshes. Section 5 concludes the paper with a discussion of some future research directions.

2 Dynamic Balanced Quadtrees

In this section, we present an abstract problem to model the process of parallel adaptive mesh refinement. This abstract problem is general enough; it uses balanced quadtrees and octrees to represent well-shaped meshes; it allows quadtrees and octrees to grow dynamically and adaptively to approximate the process of adaptive refinement of unstructured meshes. This model is also simple enough geometrically to provide a good framework for the design of mesh refinement algorithms.

2.1 Balanced Space Decomposition

The basic data structure for quad-/oct-tree based adaptive mesh refinement is a *box*. A box is a d-dimensional cube embedded in an axis-parallel manner in $\mathbb{R}^d$ [9]. Initially, there is a large d-dimensional box, we call it the *top-box*, which contains the interior of the domain, and a neighborhood around the domain. The box may be split, meaning that it is replaced by 2^d equal-sized boxes. These smaller boxes are called the *children boxes* of the original box. A sequence of splitting starting at the *top-box* generates a 2^d*-tree*, i.e., a *quadtree* in two dimensions, and an *octree* in three dimensions. *Leaf-boxes* of a 2^d-tree are those that have no child. Other boxes are *internal-boxes*, i.e., have children. The size of 2^d-tree T, denoted by $size(T)$, is the number of the leaf-boxes of T. The *depth* of a box b in T, denoted by $d(b)$, is the number of splittings needed to generate b from the top box. The depth of the top box, hence is 0. Two leaf-boxes of a 2^d-tree are *neighbors* if they have a $(d-1)$ dimensional intersection. A 2^d-tree is balanced iff for any two neighbor leaf-boxes b_1, b_2, $|d(b_1) - d(b_2)| \leq 1$. For convenience, leaf-boxes b and b_1 of a quadtree are called *edge neighbors* if they intersect on an edge; and they are called *corner neighbors* if they share only a vertex.

[htp]

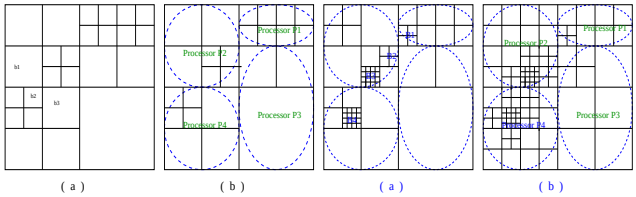

Fig. 1. A balanced quadtree T in two dimensions, a 4-way partition, refinement and balancing. Assume the splitting depth of b_1, b_2, b_3 and b_4 are 1, 1, 2 and 2 respectively.

2.2 Modeling Adaptive Refinement with Dynamic Quadtree

A balanced 2^d-tree can be viewed as a well-shaped mesh in $\mathbb{R}^d$. In fact, most quad-/oct-tree based mesh generation algorithms first construct a balanced 2^d-tree over an input domain, and then apply a local warping procedure to build the final triangular mesh [1, 9]. When the accuracy requirement of a problem is changed during the numerical simulation, we need refine the mesh accordingly. In particular, an error estimation of the computation from the previous stage generates a new spacing-function over the input domain. The new spacing-function defines the expected size of mesh elements in a particular region for the formulation in the next stage. In the context of a 2^d-tree, the refinement requires that some leaf-boxes be split into a collection of boxes of a certain size while globally maintains that the resulting 2^d-tree is still balanced. We model the refinement of a 2^d-tree as following:

Definition 1 Adaptive Refinement of 2^d-trees. A balanced 2^d-tree T and a list of non-negative integers δ, one for each leaf-box, i.e., associated with each leaf-box b is an integer $\delta(b)$, generate a new balanced tree T^* as the following.

1. Construct T' by dividing each leaf-box b of T into $2^{d\delta(b)}$ equal sized boxes.
2. Construct T^* by balancing T'.

The refinement most likely introduces load imbalance among processors, which reflects in the work for both refinement and computations thereafter. Therefore, as an integral part of parallel adaptive computation, we need to dynamically repartition the domain for both refinement and computations of the next stage. To balance the work for refinement, we need to partition a 2^d-tree before we actually refine it. In the next section, we present an efficient method to estimate the size and element distribution of a refined 2^d-tree without actually generating it.

3 Reduce Dynamic Load Balancing to Static Partitioning

The original 2^d-tree T is distributed across a parallel machine based on a partition of T. Assume we have k processors, and we have divided T into k-subdomains $S_1 \ldots, S_k$, and have mapped S_i onto processor i. A good partition is in general *balanced*, i.e., the sizes of $S_1, \ldots, S_k$ are approximately the same size. In addition,

the number of *boundary boxes*, the set of leaf-boxes that have neighbors located at different processors should be small.

A simple minded way to refine a 2^d-tree (or a mesh in general) for a new spacing-function is to have each processor refine its own subdomain to collectively construct T^*. Note that the construction of T^* from T' needs communication among processors. The original k-way partition of T naturally defines a k-way partition $(S'_1, \ldots, S'_k)$ for T' and a k-way partition $(S^*_1 \ldots, S^*_k)$ for T^*. However, these partitions may not longer be balanced. In addition, the computation of the next stage will no longer be balanced either. Note also that the set of boundary boxes will change during the construction of T' and T^*. The number of the boundary boxes may not be as small as it should be. In this approach, to balance the computation for the next stage, we could repartition T^* and distribute it according to the new partition. One shortcoming of this approach is that T^* could potentially be larger than T, and hence the overhead for redistributing T^* could be more expensive.

We would like to have a mechanism to simultaneously balance the work for refinement and for the computation of the next stage. To do so, we need to properly partition T^* before we actually generate it. Furthermore, we need a dynamic load balancing scheme that is simple enough for efficient parallel implementation. In this section, we present an algorithm that effectively reduce the dynamic load balancing problem to a collection of static partitioning problems. We first give a high-level description of our approach. Details will be given in subsequent subsections.

Repartitioning Method

Input (1) a balanced 2^d-tree T that is mapped onto k processor according to a k-way partition $S_1, \ldots, S_k$, and (2) a list of non-negative integers δ, one for each leaf-box.

1. In parallel, processor i estimates the size and the element distribution of its subdomains S'_i and S^*_i without constructing them.
2. Collectively, all processors estimate the size of T^*. Assume this estimation is N. Let $W = \alpha(N/k)$ for a predefined positive constant $\alpha < 1$.
3. In parallel, if the estimated size of S^*_i is more than W, then processor i applies the geometric partitioning algorithm of Miller-Teng-Thurston-Vavisis [6, 3] to implicitly partition S^*_i into a collection of subsubdomains $S^*_{i,1}, \ldots, S^*_{i,L_i}$. We can naturally project this partition back to S_i to generate subsubdomains $S_{i,1}, \ldots, S_{i,L_i}$.
4. We remap these subsubdomains to generate a partition of T so that the projected work for the refinement and computations thereafter at each processor is balanced. We would also like to minimize the overhead in moving these subsubdomains. We introduce a notion of subdomain graph over these subsubdomains to do dynamic balancing.
5. We construct a k-way partition of the subdomain graph using a standard static graph partitioning algorithms such as provided in Chaco and Metis [4, 5]. The partition of the subdomain graph defines a new distribution for T^* before its refinement.
6. Move each subsubdomain to its processor given by the partition of the subdomain graph.
7. In parallel, each processor refines and balances its subdomain.

8. Solve the resulting numerical system for the next stage.

3.1 Subdomain Estimation

We now estimate the size of the quadtree T^* after refining and balancing quadtree T. Our technique can be directly extended to general 2^d-trees. For each leaf-box b of a balanced 2^d-tree, the *effect region* of b, denoted by $region(b)$, is defined to be the set of all boxes that share at least one point with b. We mainly concern about the two dimensions case during the later discussion.

Lemma 2. *For any leaf-box b of a quadtree T, the size of $region(b)$ satisfies $|region(b)| \leq 12$. The region of b can be computed in a constant time.*

The *pyramid* of a leaf-box b of a quadtree T, denoted by $pyramid(b)$, is defined as following: 1:if a leaf-box $b_1 \notin region(b)$ shares a $(d-1)$ dimensional face with a box $b_2 \in region(b)$, $\delta(b_1)$=0, and $d(b_2) = d(b_1) + 1$, then $b_1 \in pyramid(b)$. 2:if a leaf-box $b_3 \notin region(b)$ shares a $(d-1)$ dimensional face with a leaf-box b_1 in $pyramid(b)$, $\delta(b_3) = 0$, and $d(b_1) = d(b_3) + 1$, then $b_3 \in pyramid(b)$.

[htop]

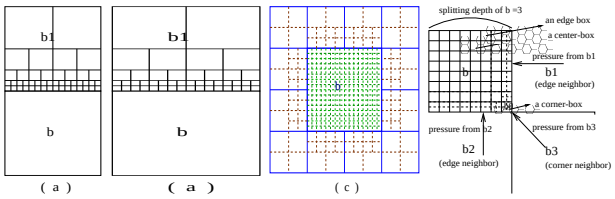

Fig. 2. The two templates for splitting boxes in $region(b)$. Fig (a): the refinement of b causes the edge neighbor b_1 to be split, Fig (b): it causes corner neighbor b_1 to be split, Fig (c): how the refinement of b influence the splitting of leaf-boxes in $region(b)$.

Lemma 3. *The set of boxes that we need to split in the construction of T^* due to the refinement of a leaf-box b is contained in $region(b) \cup pyramid(b)$.*

If b_1 is a leaf-box contained in $pyramid(b)$, then we need to split b_1 at most once. If b_1 is a leaf-box in $region(b)$, then b_1 can only be split *geometrically away* from the face shared between b_1 and b (see Figure 2 (a)) or geometrically from a corner shared by b_1 and b (see Figure 2 (b)). The depth of splitting is a function of $\delta(b)$ and $\delta(b_1)$. As shown in Figure2(c), after the refinement, the impact of b geometrically weakens away from b in $region(b)$.

Let $\theta(b, b_1) = \max([\delta(b) - \delta(b_1)], 0)$. Let $\beta(b, b_1) = d(b) - d(b_1)$. Let $pressure(b, b_1)$ be the needed split depth of leaf-box b_1, due to the imbalance caused by the refinement of leaf-box b. Let $\gamma(b, b_1) = 1$ or 2 if b and b_1 are edge neighbors or corner neighbors. Then we have $pressure(b, b_1) = \theta(b, b_1) - \beta(b, b_1) - \gamma(b, b_1)$.

The *unit boxes* of leaf-box b are the smaller box uniformly split in b by depth $\delta(b)$, i.e., the side length of this kind boxes is $2^{-\delta(b)}$ of that of b. There are three kinds of unit boxes. One is *corner-boxes*, which locates at the four corners of b. The other is *edge-boxes*, which intersect b with only one edge. All other boxes are *center-boxes*. Figure 2 shows an example of these unit boxes.

We now consider how the refinement of one leaf-box may influence the splitting of its neighboring leaf-boxes. Suppose b and b_1 are two edge neighbors in T. We have three cases:$d(b) - d(b_1) = 1, 0, -1$. Assume the refinement of b_1 causes an imbalance between b and b_1. Note that there are $2^{\delta(b)-2}$ edge-boxes need be split along one edge of b. Let a_k be the number of smaller boxes introduced in splitting an edge-box of b into depth k using the template shown at Figure 2(a). Then a_k can be computed as following.

Lemma 4. *The number of smaller boxes introduced in splitting an edge-box of b into depth k is $a_k = 3 * 2^k - 3$.*

Proof: We have $a_0 = 0$. Splitting each small box will introduce four smaller boxes. Then we have $a_k = a_{k-1} + 3 * 2^{k-1}$, which implies that $a_k = 3 * 2^k - 3$. □

We now consider the splitting of a corner-box of b to eliminate the imbalance caused by the refinement of boxes in $region(b)$. Let b_3 be a corner neighbor of b. Let b_0 be the corner-box of b which shares a vertex v with b_3, as shown in Figure 2. Let s_k be the number of boxes introduced in b_0 if we split it by depth k according to the pressure of b_3. Then we have the following lemma to compute s_k.

Lemma 5. *The number of smaller boxes introduced in splitting b_0 by depth k from pressure of b_3 is $s_k = 3 * k$.*

Proof: Clearly $s_0 = 0$. And $s_k = s_{k-1} + 3$, which implies that $s_k = 3 * k$. □

We now consider the case that two edge neighbors of b cause the corner-box to be split. W.l.o.g., let b_1 and b_2 be the two neighbors of b. Boxes b, b_1 and b_2 intersect on the vertex v of b. Let b_0 be the corner-box which also intersect on v. Let $k_1 = pressure(b_1, b)$. Let $k_2 = pressure(b_2, b)$. Let $c(k_1, k_2)$ be the number of smaller boxes split in b_0 by pressure k_1 and pressure k_2. We do not consider the corner-pressure from b_3 in $c(k_1, k_2)$ now. Then we have the following lemma to compute $c(k_1, k_2)$.

Lemma 6. *The number of smaller boxes split in b_0 by pressure k_1 and pressure k_2, is $c(k_1, k_2) = 3 * (2^{k_1} + 2^{k_2}) - 3k_2 - 6$. where we assume $k_1 \geq k_2$.*

Proof: Clearly $c(k_1, k_2) = c(k_2, k_1)$. If $k_1 = k_2$, then we have $c(0, 0) = 0, c(k, k) = c(k-1, k-1) + 3 * (2^k - 1)$. Hence, we have $c(k, k) = 3 * 2^{k+1} - 3k - 6$.

If $k_1 \neq k_2$, then w.l.o.g, we assume $k_1 > k_2$. The splitting of the corner unit box b_0 can be viewed as two steps: (1) split it by depth k_2 in both directions of b_1 and b_2, (2) split the much smaller boundary boxes generated in (1) into depth $k_1 - k_2$ along the common edge of b and b_1. Note that there are 2^{k_2} much smaller boundary boxes needed to be split in step (2). The number of total smaller boxes split is $c(k_1, k_2) = c(k_2, k_2) + 2^{k_2} * a_{k_1 - k_2}$. Generally, we have $c(k_1, k_2) = 3 * (2^{k_1} + 2^{k_2}) - 3k_2 - 6$. □

We now take into account the corner pressure from corner neighbor b_3 and the edge pressures from two edge neighbors b_1 and b_2 together, as shown in Figure 2.

Let k_1 and k_2 be the edge pressure from two edge neighbors b_1 and b_2 respectively. And let k_3 be the corner pressure from a corner neighbor b_3 of b. And let v be the common vertex of b, b_1, b_2 and b_3. Let $g(k_1, k_2, k_3)$ be the number of much smaller boxes introduced in corner-box b_0 of b due to the refinement of b_1, b_2 and b_3. Then from the above lemmas, we have the following lemma.

Lemma 7. *The number of smaller boxes introduced in b_0 due to the corner pressure from corner neighbor b_3 and the edge pressures from two edge neighbors b_1 and b_2 is $g(k_1, k_2, k_3) = c(k_1, k_2) + s_{k_3 - k_1}$ if $k_3 \geq k_1$, otherwise $g(k_1, k_2, k_3) = c(k_1, k_2)$.*

The proof is omitted here. Let $F(T, \delta) = \cup_{b, \delta(b) > 0} pyramid(b) - \cup_{b, \delta(b) > 0} region(b)$. Let $f(T, \delta) = |F(T, \delta)|$. In other words, $f(T, \delta)$ counts the leaf-boxes b that have to be split due to the imbalance introduced by the refinement of the boxes b_1 which are not in the region of b. Then by computing $pyramid(b_1)$ for $\delta(b_1) > 0$, we can compute $f(T, \delta)$ in $size(T)^2$ time.

For leaf-box b, let $V(b) = \{v_1, v_2, v_3, v_4\}$ be the vertex set of b. Let $Intr(v_i)$ be the number of smaller boxes introduced in corner-box b_0, which shares v_i with b. $Intr(v_i)$ can be computed in constant time according to analysis before. And let $E(b) = \{e_1, e_2, e_3, e_4\}$ be the edge set of b. Let $Intr(e_i)$ be the number of smaller boxes introduced in edge-boxes which intersect b on e_i. $Intr(e_i)$ can also be computed in constant time according to analysis before. Then we have the following theorem to compute $size(T^*)$.

Theorem 8. *Suppose T is a balanced quadtree and δ is a list of non-negative integers for its leaf-boxes. Then after balancing and refining T,*

$$size(T^*) = \sum_{b \in T} \left(\sum_{v_i \in V(b)} Intr(v_i) + \sum_{e_i \in E(b)} Intr(e_i) + 2^{2\delta(b)} \right) + 4 * f(T, \delta).$$

Proof: The elements of T^* has three resources. The first contribution is from the refinement of each box b(there are $2^{2\delta(b)}$ small boxes constructed). The second is from the re-refinement of edge neighbors in $region(b)$. There are $Intro(e_i)$ introduced from pressure an edge neighbor sharing e_i with b. And there are no overlaps when we do $\sum_b Intro(e_i)$. And the last is from the re-refinement of boxes in $pyramid(b)$. There are at most $4 * pyramid(b)$ small boxes introduced, because each box in $pyramid(b)$ is split to 4 smaller boxes. Note that if $b_1 \in region(b)$ then $b \in region(b_1)$. □

Lemma 9.

$$size_{app}(T^*) = \sum_{b \in T} \left(\sum_{v_i \in V(b)} Intr(v_i) + \sum_{e_i \in V(b)} Intr(e_i) + 2^{2\delta(b)} \right)$$

approximates $size(T^*)$ very well. If there are ϵ portion of boxes of T such that $\delta(b) = 0$, then $size_{app}(T^*) \geq (1 - 4\epsilon/(4 - 3\epsilon)) size(T^*)$.

Proof: $size_{app}(T^*) \leq size(T^*)$. We have $size(T^*) = size_{app}(T^*) + 4 * f(T, \delta) \leq size_{app}(T^*) + 4\epsilon * size(T) \leq size_{app}(T^*) + 4\epsilon/(4 - 3\epsilon) * size(T^*)$. It implies that $size_{app}(T^*) \geq (1 - 4\epsilon/(4 - 3\epsilon)) size(T^*)$. □

Our algorithm for estimating the size of T^* runs in linear in the $size(T)$.

3.2 Sampling Boxes from T^*

According to the size estimation of mesh T^*, we can approximately sample a random leaf-box of T^*. For a leaf-box b of mesh T, let k_e, k_s, k_w and k_n be the pressure of b from four edge-neighbor leaf-boxes respectively. Let k_{se}, k_{sw}, k_{nw} and k_{ne} be the pressure of b from four corner-neighbor leaf-boxes respectively. Then according to the lemmas of size estimation, we can compute the number of introduced splitting boxes in b due to the refinement of the neighbor leaf-boxes. Let c_1, c_2, c_3 and c_4 be the number of splitting boxes introduced in four corner-boxes of b. Let c_5, c_6, c_7 and c_8 be the number of splitting boxes introduced in edge-boxes of b. And let c_9 be the number of center-boxes splitting in b. The set of small boxes, which c_i is counted from, is called *block i*. Let (x, y) be the geometric center point of b. Let h be the side length of b. Let $h_0 = h/2^{\delta(b)}$, the side length of the split boxes in b according to the refinement of depth $\delta(b)$.

For sampling a leaf-box in T^*, we uniformly generate a random positive integer r, which is not larger than $\sum_{i=1}^{9} c_i$. W.l.o.g. we assume that $\sum_{j=1}^{i-1} c_j < r \leq \sum_{j=1}^{i} c_j$. The value r specifies the block i at which the random small box will be located. If the object block i is center block, i.e., $r > \sum_{i=1}^{8} c_i$. Let $t = r - \sum_{i=1}^{8} c_i$. Let $e = 2^{\delta(b)} - 2$, the number of small split boxes per row at the center block c_9. Let m,n be the integer such that $m \leq e-1$, $n \leq e-1$ and $t = m*e+n$. In other words, the object small box will locate the mth row and nth column at the center block of b. The coordinates of left-up corner point of center block of b is $(x - h/2 + h_0, y + h/2 - h_0)$. Then the center point of the object small box is $(x - h/2 + (n+3/2)h_0, y + h/2 - (m+3/2)h_0)$. And the side length of the sampled small box is h_0.

For the cases that the object block is an edge-block or corner-block, we have similar sampling methods. Detail of the sampling is omitted here.

3.3 Subdomain Partitioning

We first review the basic concepts of graph partitioning. Suppose we have a weighted graph $G = (V, E, w)$, where V is the set of vertices and E is the set of edges, and w assigns a positive weight to each vertex and each edge. A *k-way* partition of G is a division of its vertices into k subsets $V_1, \ldots, V_k$. The set of edges whose endpoints are in two different subsets are call the *edge-separator* of the partition. The goal of graph partitioning is to find a k-way partition such that (1) V_i has approximately equal total weight, and (2) the separator is small. There are several available software for graph partition [4, 5]. However, most of these algorithms requires the full combinatorial description of an input graph.

To partition each subdomain according to its size and distribution in T^*, we do not have its final combinatorial structure available before the refinement is actually performed. What do we have is a geometric approximation of its size and element distribution. Fortunately, the geometric information is sufficient for us to use the geometric partitioning algorithm of Miller-Teng-Thurston-Vavisis [6, 7].

Recall that the original k-way partition of T defines a k-way partition $(S'_1, \ldots, S'_k)$ for T' and a k-way partition $(S^*_1, \ldots, S^*_k)$ for T^*. However, these partitions may not longer be balanced. What we are going to do is to use the estimation of the element distribution of T^*, to implicitly divide each subdomain from $(S^*_1, \ldots, S^*_k)$

into subsubdomains of approximately equal size. The subsubdomain decomposition is described explicitly using T and its initial partition $S_1, \ldots, S_k$. The subsubdomains will be the units for the final partition.

In particularly, we use the size estimation algorithm presented in the previous section to estimate the size and element distribution of each leaf-box in T. This estimation allows us to sample a random leaf-box of T^* in each leaf-box of T. By doing so, we can obtain a sample of random leaf-boxes of S_i^*. We then apply the geometric mesh partitioning algorithm to this sample to obtain a proper multiway partition of S_i^*. This multiway partition is described as a partition tree of separating spheres and hence we can use this set of separating spheres to build a multiway partition $(S_{i,1}, \ldots, S_{i,L_i})$ of S_i. Details of the geometric mesh partitioning algorithm uses samples can be found in [6].

After the size estimation of each subdomain, we use the sampling technique to uniform and randomly select leaf-boxes in T^*. We can use the sphere based technique to partition the sampled leaf-boxes. The partition of the sampling leaf-boxes in T^* implied a partition of subdomain in T.

3.4 Subdomain Redistribution

After we have divided each subdomain of T into a collection of subsubdomains, we need to redistribute them to proper processors to balance the load and minimize the communication requirement. We introduce a *subdomain graph* SG to model the redistribution of these subsubdomains. This graph is a weighted graph and its node set contains two parts. The first part has one node for each subsubdomain that we have generated. These nodes will be referred as *subdomain nodes*. The weight of each subdomain node is equal to the estimated size of the subsubdomain in T^*. The second part has one node for each processor. We will call these nodes *processor nodes*. We will discuss the weight of processors nodes later.

Two subdomain nodes are connected in SG if they are directly connected by boundary boxes. The weight of the edge between them is equal to the the number of shared boundary leaf-boxes times a scaler which is determined by the communication cost in solving the numerical system in the parallel computer.

Each processor node is connected in SG with all subdomain nodes of its subsubdomains. The weight on the edge between a processor node and a subdomain node is the cost of moving the subsubdomain to any other processor.

We now come back to the issue of the weight of a processor node. Let W be the total weight of all subsubdomain nodes in SG. Let $w = (1/2 + \epsilon) * W/k$ for a predefined positive constant ϵ. The constant ϵ is also a function of the constant α used in the repartition method of Section 3. For example $\epsilon = \alpha/2$. The choice of the weight of processor nodes is to ensure that in the subsequent partition of SG, no two processor nodes will be assigned to the same partition. That is why we choose the weight larger than $0.5W/k$. However, if the weight is too large, then it might disturb the balance of the final partition of some static partitioning algorithm. An example of a subdomain partition and its subdomain graph is given in Figure 3.

In Figure 3, we use the follow notation. Node p_i denotes the processor i. Node S_{ij} denotes the jth subsubdomain of subdomain associated with processor i, generated by the subdomain partition algorithm.

[htp]

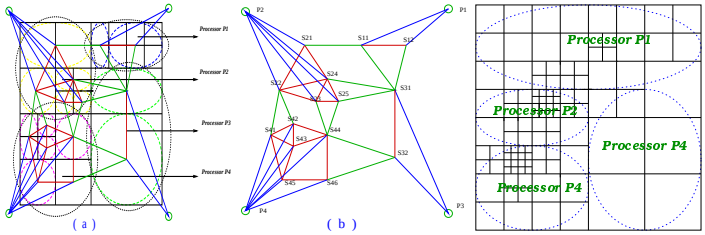

Fig. 3. constructing subdomain graph and redistribution of subdomain.

Note that the subdomain graph is very small. It has only $\Theta(k)$ nodes. So the cost for partitioning subdomain graph will be small as well.

We can use any static graph partitioning algorithm such as those provided in Chaco and Metis on SG to divide its nodes into k subsets of roughly equal total weights. It follows from the weight that we assigned to processor nodes, in the k-way partition, each subset contains exactly one processor node. Hence this partition generates a redistribution map of subsubdomains among processors in the parallel computer: a subsubdomain will be moved to the processor whose processor node is in the same subset in the k-way partition. Therefore the weight on the edge between a processor node and a subdomain node faithfully includes the communication cost in the partition.

Figure 3 gives an example of a quadtree T and its redistribution over the four processors. After the redistribution, each processor then refines and balances its new subdomain and solves its fraction of the numerical system for the next stage.

4 Remeshing Unstructured Meshes

Our parallel adaptive 2^d-tree refinement algorithm can be extended to general unstructured meshes. In this section, we outline our approach. It follows from a series of work by Bern-Eppstein-Gilbert [1], Mitchell-Vavasis [9], and Miller-Talmor-Teng-Walkington [8] that given a well-shaped mesh M in $\mathbb{R}^d$, there is a balanced 2^d-tree T_M that approximates M. In particularly, T_M has the property that there are three positive constants $c > 1$, $\beta_1 < 1$ and $\beta_2 > 1$ such that (1) For each element e in M, the number of leaf-boxes of T_M that intersect e is at most c. (2) For each leaf-box b in T_M, the number of elements of M that intersect b is at most c. (3) In addition, if a leaf-box b of T_M intersects e, then $\beta_1 area(e) \leq area(b) \leq \beta_2 area(e)$.

Given a well-shaped mesh M, we can construct T_M in time linear in the size of M. Moreover, such computation can be optimally speeded-up if we have a multiple number of processors. We can then use the following strategy to design a parallel adaptive refinement algorithm for unstructured meshes.

Parallel Refinement Method

Input (1) a well-shaped mesh M that is mapped onto k processor according

to a k-way partition $M_1, \ldots, M_k$, and (2) a spacing-function f defining the new spacing at each vertices of M.

In parallel, we generate T_M. We project the k-way partition $M_1, \ldots, M_k$ to T_M to obtain a k-way partition of T_M. For each vertex $v \in M$, we compute the ratio r_m that is equal to the ratio of the current spacing at v to $f(v)$. For each box $b \in T_M$, let $\delta(b)$ be the logarithm of the average ratio of all vertices of M that lies inside b. We then apply our 2^d-tree load balancing algorithm to compute a k-way partition of T and project it back to M to obtain a new k-way partition of M. This k-way partition will be balanced for M^*, the refined mesh for M. Then permute M according this new partition and each processor applies a sequential mesh refinement algorithm to their own submesh and collaboratively refine the boundary elements among the submeshes.

In the full version of this paper, we will review some of sequential mesh refinement algorithms. We can support any sequential mesh coarsening algorithm.

5 Final Remarks

In this paper, we present a dynamic load balancing algorithm for parallel adaptive mesh refinement. The main objective of this research is to develop effective algorithms that are simple enough for implementation. We focus on reducing dynamic load balancing to static partitioning in a black-box fashion and on reducing parallel mesh refinement to a collection of traditional sequential mesh refinements. We show how the estimation of the size and element distribution of a refined mesh can be used for this objective. There are several directions that we can extend and improve the method presented in this paper.

In our abstract model for adaptive mesh refinement, we assume that each leaf-box will be uniformly split in T'. In practice, we may need to split each leaf-box according to a given pattern.

The scheme developed in this paper for unstructured mesh refinement first builds a balanced 2^d-tree to approximate the unstructured mesh. This could be cumbersome. It is desirable to have a more direct method to estimate the size and element distribution of unstructured meshes other than 2^d-tree.

In our current model for adaptive refinement, we assume that the mesh will be made finer at every region. For certain applications, some regions will be "de-refined", i.e., will be coarsened. We need to extend our adaptive refinement scheme to handle mixed adaptive refinement and coarsening.

We are in the process of implementing ideas and methods developed in this paper. Experimental results will be presented in subsequent writings. The full version of the paper is available at request.

References

1. M. Bern, D. Eppstein and J. R. Gilbert. Provably good mesh generation. In *31st Annual Symposium on Foundations of Computer Science, IEEE*, 231–241, 1990.

2. L. Paul Chew, Nikos Chrisochoides, Florian Sukup. Parallel constrained delaunay meshing. *Trends in Unstructured Mesh Generation* edited by S.A.Canann and S.Saigal, pp89-96, 1997.
3. J. R. Gilbert, G. L. Miller, and S.-H. Teng. Geometric mesh partitioning: Implementation and experiments. *SIAM J. Scientific Computing*, to appear, 1998.
4. B. Hendrickson and R. Leland. The Chaco user's guide, Version 1.0. Technical Report SAND93-2339, Sandia National Laboratories, Albuquerque, NM, 1993.
5. G. Karypis and V. Kumar. A fast and high quality multilevel scheme for partitioning irregular graphs. *SIAM J. Scientific Computing* to appear, 1997.
6. G. L. Miller, S.-H. Teng, W. Thurston, and S. A. Vavasis. Automatic mesh partitioning. In A. George, J. Gilbert, and J. Liu, editors, *Sparse Matrix Computations: Graph Theory Issues and Algorithms*, IMA Volumes in Mathematics and its Applications. Springer-Verlag, pp57–84, 1993.
7. G. L. Miller, S.-H. Teng, W. Thurston, and S. A. Vavasis. Geometric separators for finite element meshes. *SIAM J. Scientific Computing*, to appear, 1998.
8. G. L. Miller, D. Talmor, S.-H. Teng, and N. Walkington. A Delaunay based numerical method for three dimensions: generation, formulation, and partition. In *Proc. 27th Annu. ACM Sympos. Theory Comput.*, pages 683–692, 1995.
9. S. A. Mitchell and S. A. Vavasis. Quality mesh generation in three dimensions. Proc. ACM Symposium on Computational Geometry, pp 212-221, 1992.
10. T.Okusanya, J.Peraire. 3D parallel unstructured mesh generation. *Trends in Unstructured Mesh Generation* edited by S.A.Canann and S.Saigal, pp109-116, 1997.
11. M. L.Staten and S. A. Canann. Post refinement element shape improvement for quadrilateral meshes. *Trends in Unstructured Mesh Generation* edited by S.A.Canann and S.Saigal, pp9-16, 1997.
12. G. Strang and G. J. Fix. *An Analysis of the Finite Element Method*, Prentice-Hall, 1973.
13. S.-H. Teng. A geometric approach to parallel hierarchical and adaptive computing on unstructured meshes. In *Fifth SIAM Conference on Applied Linear Algebra*, pages 51–57, June 1994.