Parallel Incremental Graph Partitioning

Chao-Wei Ou and Sanjay Ranka

Northeast Parallel Architecture Center
and
School of Computer and Information Science
Syracuse University
Syracuse, NY 13244-4100

Email: *cwou@npac.syr.edu, ranka@top.cis.syr.edu*
Phone: (315) 443-4890, (315) 443-4457
FAX: (315) 443-1122

August 1994

Abstract

Partitioning graphs into equally large groups of nodes while minimizing the number of edges between different groups is an extremely important problem in parallel computing. For instance, efficiently parallelizing several scientific and engineering applications requires the partitioning of data or tasks among processors such that the computational load on each node is roughly the same, while communication is minimized. Obtaining exact solutions is computationally intractable, since graph-partitioning is an NP-complete.

For a large class of irregular and adaptive data parallel applications (such as adaptive meshes), the computational structure changes from one phase to another in an incremental fashion. In incremental graph-partitioning problems the partitioning of the graph needs to be updated as the graph changes over time; a small number of nodes or edges may be added or deleted at any given instant.

In this paper we use a linear programming-based method to solve the incremental graph partitioning problem. All the steps used by our method are inherently parallel and hence our approach can be easily parallelized. By using an initial solution for the graph partitions derived from recursive spectral bisection-based methods, our methods can achieve repartitioning at considerably lower cost than can be obtained by applying recursive spectral bisection. Further, the quality of the partitioning achieved is comparable to that achieved by applying recursive spectral bisection to the incremental graphs from scratch.

1 Introduction

Graph partitioning is a well-known problem for which fast solutions are extremely important in parallel computing and in research areas such as circuit partitioning for VLSI design. For instance, parallelization of many scientific and engineering problems requires partitioning data among the processors in such a fashion that the computation load on each node is balanced, while communication is minimized. This is a graph-partitioning problem, where nodes of the graph represent computational tasks, and edges describe the communication between tasks with each partition corresponding to one processor. Optimal partitioning would allow optimal parallelization of the computations with the load balanced over various processors and with minimized communication time. For many applications, the computational graph can be derived only at runtime and requires that graph partitioning also be done in parallel. Since graph partitioning is NP-complete, obtaining suboptimal solutions quickly is desirable and often satisfactory.

For a large class of irregular and adaptive data parallel applications such as adaptive meshes [2], the computational structure changes from one phase to another in an incremental fashion. In incremental graph-partitioning problems, the partitioning of the graph needs to be updated as the graph changes over time; a small number of nodes or edges may be added or deleted at any given instant. A solution of the previous graph-partitioning problem can be utilized to partition the updated graph, such that the time required will be much less than the time required to reapply a partitioning algorithm to the entire updated graph. If the graph is not repartitioned, it may lead to imbalance in the time required for computation on each node and cause considerable deterioration in the overall performance. For many of these problems the graph may be modified after every few iterations (albeit incrementally), and so the remapping must have a lower cost relative to the computational cost of executing the few iterations for which the computational structure remains fixed. Unless this incremental partitioning can itself be performed in parallel, it may become a bottleneck.

Several suboptimal methods have been suggested for finding good solutions to the graph-partitioning problem. For many applications, the computational graph is such that the vertices correspond to two- or three-dimensional coordinates and the interaction between computations is limited to vertices that are physically proximate. This information can be exploited to achieve the partitioning relatively quickly by clustering physically proximate points in two or three dimensions. Important heuristics include recursive coordinate bisection, inertial bisection, scattered decomposition, and index based partitioners [3, 6, 12, 11, 14, 16]. There are a number of methods which use explicit graph information to achieve partitioning. Important heuristics include simulated annealing, mean field annealing, recursive spectral bisection, recursive spectral multisection, mincut-based methods, and genetic algorithms [1, 4, 5, 7, 8, 9, 10, 13]. Since, the methods use explicit graph information, they have wider applicability and produce better quality partitioning.

In this paper we develop methods which use explicit graph information to perform incremental graph-partitioning. Using recursive spectral bisection, which is regarded as one of the best-known methods for graph partitioning, our methods can partition the new graph at considerably lower cost. The quality of partitioning achieved is close to that achieved by applying recursive spectral bisection from scratch. Further, our algorithms are inherently parallel.

The rest of the paper is outlined as follows. Section 2 defines the incremental graph-partitioning problem. Section 3 describes linear programming-based incremental graph partitioning. Section 4 describes a multilevel approach to solve the linear programming-based incremental graph partitioning. Experimental results of our methods on sample meshes are described in Section 5, and conclusions are given in Section 6.

2 Problem definition

Consider a graph $G = (V, E)$, where V represents a set of vertices, E represents a set of undirected edges, the number of vertices is given by $n = |V|$, and the number of edges is given by $m = |E|$. The graph-partitioning problem can be defined as an assignment scheme $M : V \longrightarrow P$ that maps vertices to partitions. We denote by $B(q)$ the set of vertices assigned to a partition q, i.e., $B(q) = \{v \in V : M(v) = q\}$.

The weight w_i corresponds to the computation cost (or weight) of the vertex v_i. The cost of an edge $w_e(v_1, v_2)$ is given by the amount of interaction between vertices v_1 and v_2. The weight of every partition can be defined as

$$W(q) = \sum_{v_i \in B(q)} w_i. \tag{1}$$

The cost of all the outgoing edges from a partition represent the total amount of communication cost and is given by

$$C(q) = \sum_{v_i \in B(q),\, v_j \notin B(q)} w_e(v_i, v_j). \tag{2}$$

We would like to make an assignment such that the time spent by every node is minimized, i.e., $\min_q \ (W(q) + \beta C(q))$, where β represents the ratio of cost of unit computation/cost of unit communication on a machine. Assuming computational loads are nearly balanced ($W(0) \approx W(1) \approx \cdots \approx W(p-1)$), the second term needs to be minimized. In the literature $\sum C(q)$ has also been used to represent the communication.

Assume that a solution is available for a graph $G(V, E)$ by using one of the many available methods in the literature, e.g., the mapping function M is available such that

$$B(1) \approx B(2) \approx B(3) \approx \cdots \approx B(q-1), \tag{3}$$

and the communication cost is close to optimal. Let $G'(V', E')$ be an incremental graph of $G(V, E)$

$$V' = V \cup V_1 - V_2 \quad \text{where} \quad V_2 \subseteq V, \tag{4}$$

i.e., some vertices are added and some vertices are deleted. Similarly,

$$E' = E \cup E_1 - E_2 \quad \text{where} \quad E_2 \subseteq E, E_1 \cap E_2 \neq \phi; \tag{5}$$

i.e., some edges are added and some are deleted. We would like to find a new mapping $M' : V' \longrightarrow P$ such that the new partitioning is as load balanced as possible and the communication cost is minimized.

The methods described in this paper assume that $G'(V', E')$ is sufficiently similar to $G(V, E)$ that this can be achieved, i.e., the number of vertices and edges added/deleted are a small fraction of the original number of vertices and edges.

3 Incremental partitioning

In this section we formulate incremental graph partitioning in terms of linear programming. A high-level overview of the four phases of our incremental graph-partitioning algorithm is shown in Figure 1. Some notation is in order.

Let

1. P be the number of partitions.
2. $B'(i)$ represent the set of vertices in partition i.

3. μ represent the average load for each partition $\mu = \frac{\sum_i |B'(i)|}{P}$.

The four steps are described in detail in the following sections.

Step 1: Assign the new vertices to one of the partitions (given by M').

Step 2: Layer each partition to find the closest partition for each vertex (given by L').

Step 3: Formulate the linear programming problem based on the mapping of Step 1 and balance loads (i.e., modify M') minimizing the total number of changes in M'.

Step 4: Refine the mapping in Step 2 to reduce the communication cost.

Figure 1: The different steps used in our incremental graph-partitioning algorithm.

3.1 Assigning an initial partition to the new nodes

The first step of the algorithm is to assign an initial partition to the nodes of the new graph (given by $M'(V)$). A simple method for initializing $M'(V)$ is given as follows. Let

$$M'(v) = M(v) \quad \text{for all} \quad v \in V - V_1. \tag{6}$$

For all the vertices $v \in V_1$,

$$M'(v) = M(x) \quad \text{where} \quad \min_{x \in V - V_2} (d(v, x)), \tag{7}$$

$d(v, x)$ is the shortest distance in the graph $G'(V', E')$. For the examples considered in this paper we assume that G' is connected. If this is not the case, several other strategies can be used.

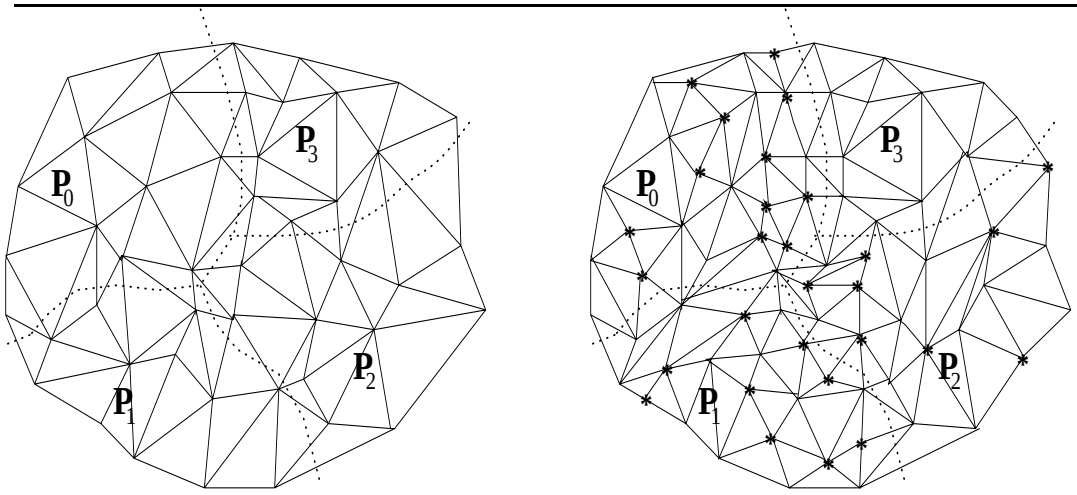
- If $G''(V \cup V_1, E \cup E_1)$ is connected, this graph can be used instead of G for calculation of $M'(V)$.
- If $G''(V \cup V_1, E \cup E_1)$ is not connected, then the new nodes that are not connected to any of the old nodes can be clustered together (into potentially disjoint clusters) and assigned to the partition that has the least number of vertices.

For the rest of the paper we will assume that $M'(v)$ can be calculated using the definition in (7), although the strategies developed in this paper are, in general, independent of this mapping. Further, for ease of presentation, we will assume that the edge and the vertex weights are of unit value. All of our algorithms can be easily modified if this is not the case. Figure 2 (a) describes the mapping of each the vertices of a graph. Figure 2 (b) describes the mapping of the additional vertices using the above strategy.

3.2 Layering each partition

The above mapping would ordinarily generate partitions of unequal size. We would like to move vertices from one partition to another to achieve load balancing, while keeping the communication cost as small as possible. This is achieved by making sure that the vertices transferred between two partitions are close to the boundary of the two partitions. We assign each vertex of a given partition to a different partition it is close to (ties are broken arbitrarily).

$$L'(v) = M(x) \tag{8}$$



(a) (b)

Figure 2: (a) Initial Graph (b) Incremental Graph (New vertices are shown by "*").

where x is such that

$$\min_{x \notin B'(M(v))} (d(v,x)) \tag{9}$$

is satisfied; $d(v,x)$ is the shortest distance in the graph between v and x.

A simple algorithm to perform the layering is given in Figure 3. It assumes the graph is connected. Let α_{ij} represent the number of such vertices of partition i that can be moved to partition j. For the example case of Figure 3, labels of all the vertices are given in Figure 4. A label 2 of vertex in partition 1 corresponds to the fact that this vertex belongs to the set that contributed to α_{12}.

3.3 Load balancing

Let l_{ij} represent the number of vertices to be moved from partition i to partition j to achieve load balance. There are several ways of achieving load balancing. However, since one of our goals is to minimize communication cost, we would like to minimize $\sum_i \sum_j l_{ij}$, because this would correspond to a minimization of the amount of vertex movement (or "deformity") in the original partitions. Thus the load-balancing step can be formally defined as the following linear programming problem.

Minimize

$$\sum_{0 \le i \ne j \le P} l_{ij} \tag{10}$$

subject to

$$0 \le l_{ij} \le \alpha_{ij} \le |B'(i)| \tag{11}$$

$$\sum_{0 \le i < P} (l_{ij} - l_{ji}) = |B'(j)| - \mu \quad 0 \le j < P. \tag{12}$$

Constraint 12 corresponds to the load balance condition.

The above formulation is based on the assumption that changes to the original graph are small and the initial partitioning is well balanced, hence moving the boundaries by a small amount will give balanced partitioning with low communication cost.

{ $map[v[j]]$ represents the mapping of vertex j. }

{ $adj_i[j]$ represents the j^{th} element of the local adjacent list in partition i. }

{ $xadj_i[v[j]]$ represents the starting address of vertex j in the local adjacent list of partition i. }

{ $S_i^{(j,k)}$ represents the set of vertices of partition i at a distance k from a node in partition j.

{ $Neighbor_i$ represents the set of partitions which have common boundaries with partition i. }

For each partition i **do**

  For vertex $v[j] \in V_i$ **do**

    For $k \longleftarrow xadj_i[v[j]]$ **to** $xadj_i[v[j+1]]$ **do**

      if $map[adj_i[k] \neq i$

        $Count_i[map[adj_i[k]]] := Count_i[map[adj_i[k]]] + 1$

    if $\sum_l Count[l] > 0$

      Add $v[j]$ into $S_i^{(tag,0)}$

      { where $Count[tag] = \max_l Count[l]$ }

      $V_i \leftarrow V_i - \{v[j]\}$

  $level := 0$

  repeat

    For $k \in Neighbor_i$ **do**

      For vertex $v[j] \in S_i^{(k,level)}$ **do**

        For $l \longleftarrow xadj_i[v[j]]$ **to** $xadj_i[v[j+1]]$ **do**

          if $adj_i[l] \notin S_i^{(k,level)}$

            $count_i[adj_i[l]][k] := count_i[adj_i[l]][k] + 1$

            Add $v[j]$ into tmp_S

    $level := level + 1$

    For vertex $v[j] \in tmp_S$ **do**

      Add $v[j]$ into $S_i^{(tag,level)}$

      { where $count_i[j][tag] = \max_l count_i[j][l]$ }

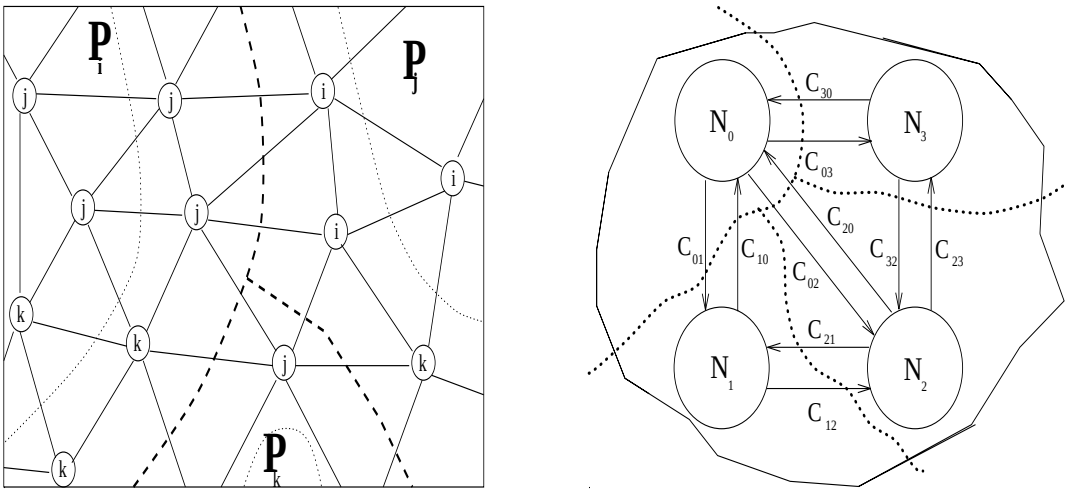
    $V_i \leftarrow V_i - \{v[j]\}$

  until $(V_i = \phi)$

  For $j \in Neighbor_i$ **do**

    $$\alpha_{ij} := \sum_{0 \leq k < level} |S_i^{(j,k)}|$$

Figure 3: Layering Algorithm



(a) (b)

Figure 4: Labeling the nodes of a graph to the closest outside partition; (a) a microscopic view of the layering for a graph near the boundary of three partitions; (b) layering of the graph in Figure 2 (b); no edges are shown.

Constraints in (11):

$$l_{01} \leq 9 \; l_{02} \leq 7 \; l_{03} \leq 12 \; l_{10} \leq 10 \; l_{12} \leq 11$$
$$l_{20} \leq 3 \; l_{21} \leq 7 \; l_{23} \leq 9 \; l_{30} \leq 7 \; l_{32} \leq 5$$

Constraints in (12):

$$l_{01} + l_{02} + l_{03} - l_{10} - l_{20} - l_{30} = 8$$
$$l_{10} + l_{12} - l_{01} - l_{21} = 1$$
$$-l_{20} - l_{21} - l_{23} + l_{02} + l_{12} + l_{32} = 1$$
$$-l_{30} - l_{32} + l_{03} + l_{23} = 8$$

Solution using the Simplex Method

$$l_{03} = 8, l_{12} = 1$$
all other values are zero.

Figure 5: Linear programming formulation and its solution, based on the mapping of the graph in Figure 2; (b) using the labeling information in Figure 4 (b).

There are several approaches to solving the above linear programming problem. We decided to use the simplex method because it has been shown to work well in practice and because it can be easily parallelized.[1] The simplex formulation of the example in Figure 2 is given in Figure 5. The corresponding solution is $l_{03} = 8$ and $l_{12} = 1$. The new partitioning is given in Figure 6.

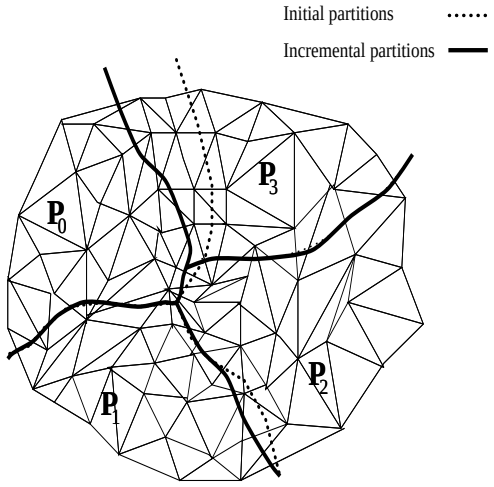


Figure 6: The new partition of the graph in Figure 2 (b) after the **Load Balancing** step.

The above set of constraints may not have a feasible solution. One approach is to relax the constraint in (11) and not have $l_{ij} \leq \alpha_{ij}$ as a constraint. Clearly, this would achieve load balance but may lead to major modifications in the mapping. Another approach is to replace the constraint in (12) by

$$\sum_{0 \leq i < P} (l_{ij} - l_{ji}) = \frac{|B'(j)| - \mu}{\Delta} \quad 0 \leq j < P. \tag{13}$$

Assuming $C > \Delta > 1$, this would not achieve load balancing in one step, but several such steps can be applied to do so. If a feasible solution cannot be found with a reasonable value of Δ (within an upper bound C), it would be better to start partitioning from scratch or solve the problem by adding only a fraction of the nodes at a given time, i.e., solve the problem in multiple stages. Typically, such cases arise when all the new nodes correspond to a few partitions and the amount of incremental change is greater than the size of one partition.

3.4 Refinement of partitions

The formulation in the previous section achieves load balance but does not try explicitly to reduce the number of cross-edges. The minimization term in (10) and the constraint in (11) indirectly keep the cross-edges to a minimum under the assumption that the initial partition is good. In this section we describe a linear programming-based strategy to reduce the number of cross-edges, while still maintaining the load balance. This is achieved by finding all the vertices of partitions i on the boundary of partition i and j such that the cost of edges to the vertices in j are larger than the cost of edges to local vertices (Figure 7), i.e., the total cost of cross-edges will decrease by moving the vertex from partition i to j, which will affect the load

[1] We have used a dense version of simplex algorithm. The total time can potentially be reduced by using sparse representation.

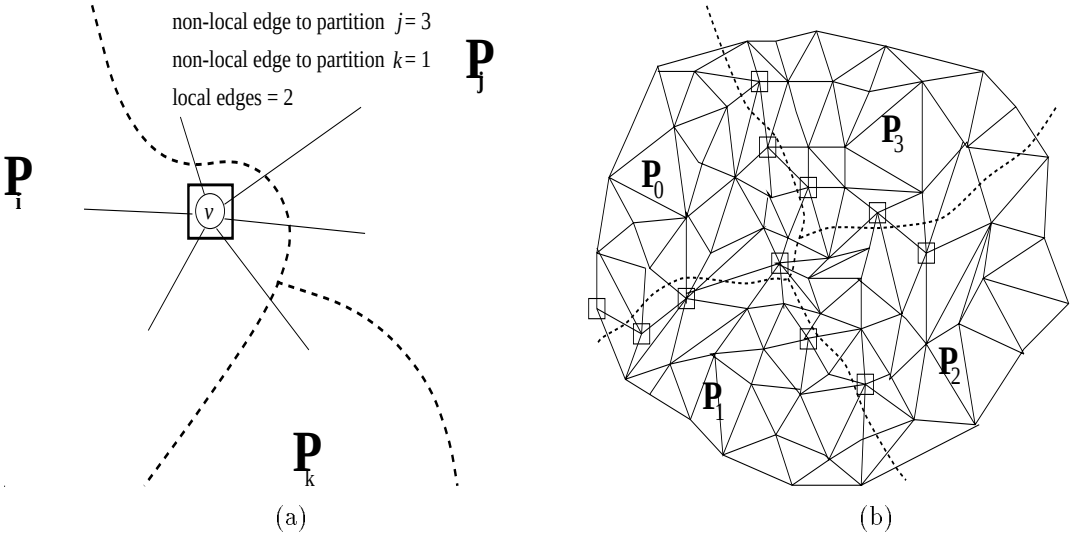


Figure 7: Choosing vertices for refinement. (a) Microscopic view of a vertex which can be moved from partition P_i to P_j, reducing the number of cross edges; (b) the set of vertices with the above property in the partition of Figure 6.

balance. In the following a linear programming formulation is given that moves the vertices while keeping the load balance.

Let $M''(k) : V' \longrightarrow P$ represent the mapping of each vertex after the load-balancing step. Let $out(k, j)$ represent the number of edges of vertex k in partition $M''(k)$ connected to partition $j (j \neq M''(k))$, and let $in(k)$ represent the number of vertices a vertex k is connected to in partition $M''(k)$. Let b_{ij} represent the number of vertices in partition i which have more outgoing edges to partition j than local edges.

$$b_{ij} = |\{V \in B_i'' | out(V, j) - in(V) \geq 0\}|.$$

We would like to maximize the number of vertices moved so that moving a vertex will not increase the cost of cross-edges. The inequality in the above definition can be changed to a strict inequality. We leave the equality, however, since by including such vertices the number of points that can be moved can be larger (because these vertices can be moved to satisfy load balance constraints without affecting the number of cross-edges).

The refinement problem can now be posed as the following linear programming problem:

Maximize

$$\sum_{0 \leq i \neq j \subset P} l_{ij} \tag{14}$$

such that

$$0 \leq l_{ij} \leq b_{ij} \quad 0 \leq i \neq j < P \tag{15}$$

$$\sum_{0 \leq i < j} (l_{ij} - l_{ji}) = 0 \quad 0 \leq j < P. \tag{16}$$

This refinement step can be applied iteratively until the effective gain by the movement of vertices is small. After a few steps, the inequalities ($l_{ij} \leq b_{ij}$) need to be replaced by strict inequalities ($l_{ij} < b_{ij}$);

Constraint (15)

$$l_{01} \leq 1\ l_{02} \leq 1\ l_{03} \leq 1\ l_{10} \leq 2\ l_{12} \leq 1$$
$$l_{20} \leq 0\ l_{21} \leq 1\ l_{23} \leq 1\ l_{30} \leq 2\ l_{32} \leq 1$$

Load Balancing Constraint (16)

$$l_{01} + l_{02} + l_{03} - l_{10} - l_{20} - l_{30} = 0$$
$$l_{10} + l_{12} - l_{01} - l_{21} = 0$$
$$-l_{20} - l_{21} - l_{23} + l_{02} + l_{12} + l_{32} = 0$$
$$-l_{30} - l_{32} + l_{03} + l_{23} = 0$$

Solution using Simplex Method

$$l_{01} = 0,\ l_{02} = 1,\ l_{03} = 1,\ l_{10} = 1,\ l_{12} = 1$$
$$l_{20} = 0,\ l_{21} = 1,\ l_{23} = 1,\ l_{30} = 1,\ l_{32} = 1$$

Figure 8: Formulation of the refinement step using linear programming and its solution.

otherwise, vertices having an equal number of local and nonlocal vertices may move between boundaries without reducing the total cost. The simplex formulation of the example in Figure 6 is given in Figure 8, and the new partitioning after refinement is given in Figure 9.

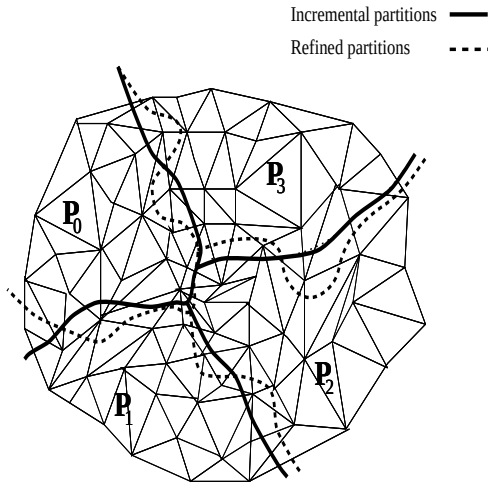


Figure 9: The new partition of the graph in Figure 6 after the **Refinement** step.

3.5 Time complexity

Let the number of vertices and the number of edges in a graph be given by V and E, respectively. The time for layering is $O(V+E)$. Let the number of partitions be P and the number of edges in the partition graph[2]

[2]Each node of this graph represents a partition. An edge in the super graph is present whenever there are any cross edges from a node of one partition to a node of another partition.

be R. The number of constraints and variables generated for linear programming are $O(P+R)$ and $O(2R)$, respectively.

Thus the time required for the linear programming is $O((P+R)R)$. Assuming R is $O(P)$, this reduces to $O(P^2)$. The number of iterations required for linear programming is problem dependent. We will use $f(P)$ to denote the number of iterations. Thus the time required for the linear programming is $O(P^2 f(P))$. This gives the total time for repartitioning as $O(E + P^2 l(P))$.

The parallel time is considerably more difficult to analyze. We will analyze the complexity of neglecting the setup overhead of coarse-grained machines. The parallel time complexity of the layering step depends on the maximum number of edges assigned to any processor. This could be approximated by $O(E/P)$ for each level, assuming the changes to the graph are incremental and that the graph is much larger than the number of processors. The parallelization of the linear programming requires a broadcast of length proportional to $O(P)$. Assuming that a broadcast of size P requires $b(P)$ amount of time on a parallel machine with P processors, the time complexity can be approximated by $O(\frac{E}{P} + l(P)(P + b(P)))$.

4 A multilevel approach

For small graphs a large fraction of the total time spent in the algorithm described in the previous section will be on the linear programming formulation and its solution. Since the time required for one iteration of the linear programming formulation is proportional to the square of the number of partitions, it can be substantially reduced by using a multilevel approach. Consider the partitioning of an incremental graph for 16 partitions. This can be completed in two stages: partitioning the graph into 4 super partitions and partitioning each of the 4 super partitions into 4 partitions each. Clearly, more than two stages can be used. The advantage of this algorithm is that the time required for applying linear programming to each stage would be much less than the time required for linear programming using only one stage. This is due to a substantial reduction in the number of variables as well as in the constraints, which are directly dependent on the number of of partitions. However, the mapping initialization and the layering needs to be performed from scratch for each level. Thus the decrease in cost of linear programming leads to a potential increase in the time spent in layering.

The multilevel algorithm requires combining the partitions of the original graph into super partitions. For our implementations, recursive spectral bisection was used as an *ab initio* partitioning algorithm. Due to its recursive property it creates a natural hierarchy of partitions. Figure 10 shows a two-level hierarchy of partitions. After the linear programming-based algorithm has been applied for repartitioning a graph that has been adapted several times, it is possible that some of the partitions corresponding to a lower level subtree have a small number of boundary edges between them. Since the multilevel approach results in repartitioning with a small number of partitions at the lower levels, the linear programming formulations may produce infeasible solutions at the lower levels. This problem can be partially addressed by reconfiguring the partitioning hierarchy.

A simple algorithm can be used to achieve reconfiguration. It tries to group proximate partitions to form a multilevel hierarchy. At each level it tries to combine two partitions into one larger partition. Thus the number of partitions is reduced by a factor of two at every level by using a procedure FIND_UNIQUE_NEIGHBOR(P) (Figure 11), which finds a unique neighbor for each partition such that the number of cross-edges between them is as large as possible. This is achieved by applying a simple heuristic (Figure 12) that uses a list of all the partitions in a random order (each processor has a different order). If more than one processor is successful in generating a feasible solution, ties are broken based on the weight and the processor number. The result of the merging is broadcast to all the processors. In case none of the

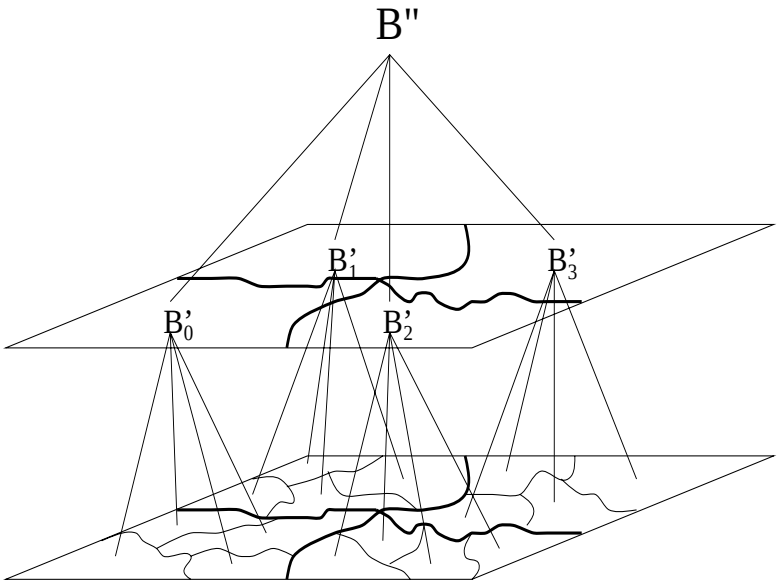


Figure 10: A two-level hierarchy of 16 partitions

FIND_UNIQUE_NEIGHBOR(P)

{ $A_0, A_1, A_2, \cdots A_{P-1}$ represent the P partitions }

{ $Edge_{ij}$ represents the number of edges from partition i to partition j. }

$global\_success$:= FALSE

$trial$:= 0

While (**not** $global\_success$) **and** ($trial < T$) **do**

For each processor i **do**

$Mark[0..P-1]$:= -1

$Random\_list[0..P-1]$:= list of all partitions in a random order

$Weight$:= 0

FIND_PAIR$(success, Mark, Weight, Edge)$

$global\_success$:= **GLOBAL_OR**$(success)$

if (**not** $global\_success$) **then**

FIX_PAIR$(success, Mark, Weight, Edge)$

$global\_success$:= **GLOBAL_OR**$(success)$

if ($global\_success$) **then**

$winner$:= **FIND_WINNER**$(success, Weight)$

{ Return the processor number of maximum $Weight$ }

BROADCAST$(winner, Mark)$

{ Processor $winner$ broadcast $Mark$ to all the processors }

return$(global\_success)$

else

$trial$:= $trial$+1

Figure 11: Reconstruction Algorithm

FIND_PAIR$(success, Mark, Weight, Edge)$

$success$:= TRUE

for $j \longleftarrow 0$ **to** $P-1$ **do**

  if $(Mark[j] < 0)$ **then**

    Find a neighbor k of j where $(Mark[k] < 0)$

    if k exists **then**

      $Mark[k] := j$ $Mark[j] := k$

      $Weight := Weight + Edge(i,j)$

    else

      $success$:= FALSE

FIX_PAIR$(success, Mark, Weight, Edge)$

$success$:= TRUE $j = 0$

While $(j < P)$ **and** $(success)$ **do**

  if $(Mark[j] < 0)$ **then**

    if a x exists such that $(Mark[x] < 0)$, (x is a neighbor of l), $(Mark[l] = k)$, and (k is a neighbor of j)

      $Mark[x] := l$ $Mark[l] := x$

      $Mark[j] := k$ $Mark[k] := j$

      $Weight := Weight + Edge(j,k) + Edge(l,x)$ - $Edge(k,l)$

      $j := j + 1$

    else

      $success$:= FALSE

  else

    $j := j + 1$

Figure 12: A high level description of the procedures used in **FIND_UNIQUE_NEIGHBOR**.

processors are successful, another heuristic (Figure 12) is applied that tries to modify the partial assignments made by heuristic 1 to find a neighbor for each partition. If none of the processors are able to find a feasible solution, each processor starts with another random solution and the above step is iterated a constant number (L) times.[3] Figure 11 shows the partition reconfiguration for a simple example.

If the reconfiguration algorithm fails, the multilevel algorithm can be applied with a lower number of levels (or only one level).

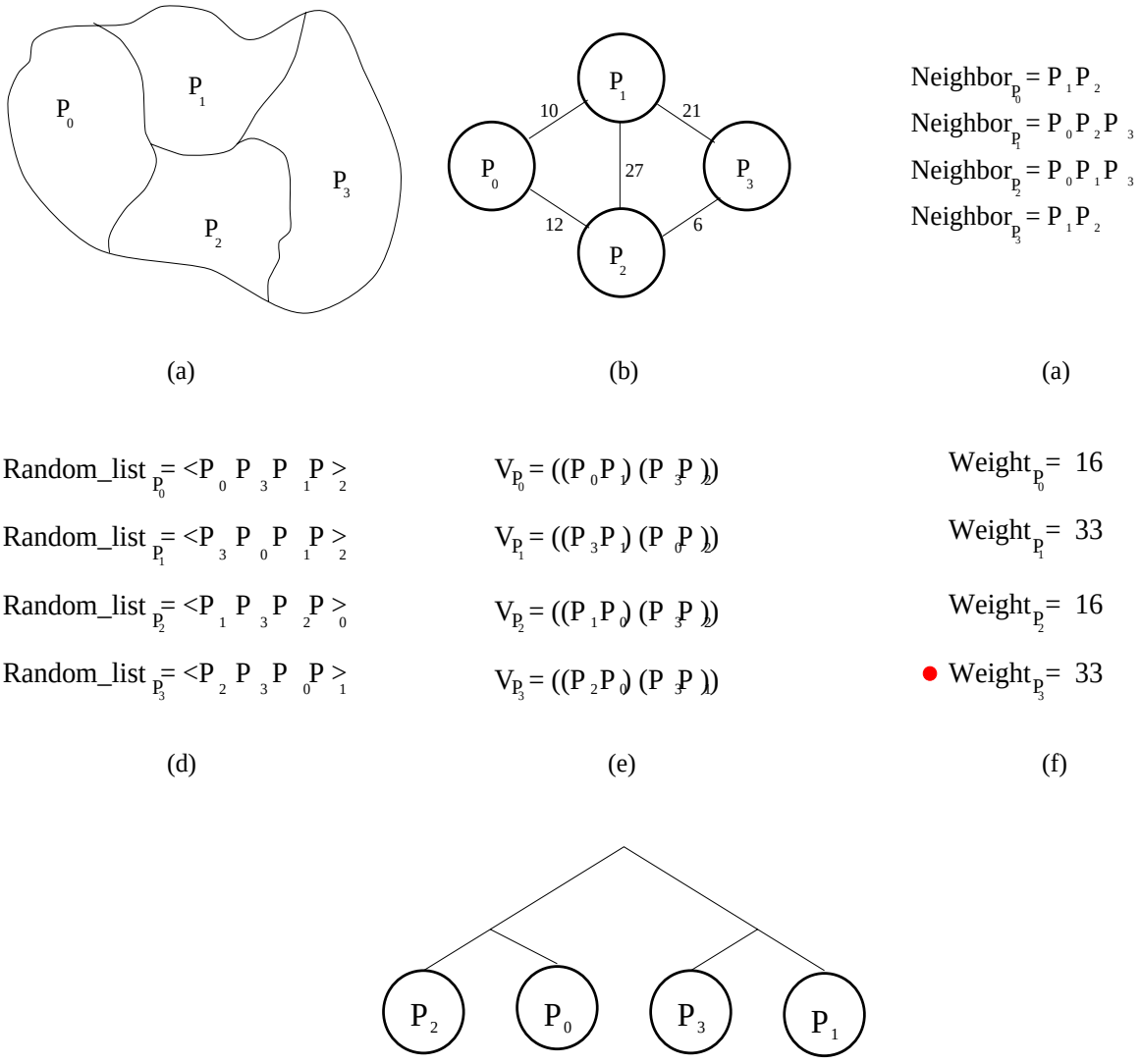


Figure 13: A working example of the reconstruction algorithm. (a) Graph with 4 partitions; (b) partition graph; (c) adjacency lists; (d) random order lists; (e) partition rearrangement; (f) processor 3 broadcasts the result $((P_1P_0)(P_3P_1))$ to the other processors; (g) hierarchy after reconfiguration.

[3] In practice, we found that the algorithm never requires more than one iteration.

4.1 Time complexity

In the following we provide an analysis assuming that reconfiguration is not required. The complexity of reconfiguration will be discussed later. For the multilevel approach we assume that at each level the number of partitions done is equal and given by k. Thus the number of levels generated is $log_k P$. The time required for layering increases to $O(Elog_k P)$. The number of linear programming formulations can be given by $O(\frac{P}{k})$. Thus the total time for linear programming can be given by $O(\frac{P}{k}k^2 f(k))$. The total time required for repartitioning is given by $O(Elog_k P + Pkf(k))$. An appropriate value of k would minimize the sum of the cost of layering and the cost of the linear programming formulation. The choice of k also depends on the quality of partitioning achieved; increasing the number of layers would, in general, have a deteriorating effect on the quality of partitioning. Thus values of k have to be chosen based on the above tradeoffs. However, the analysis suggests that for reasonably sized graphs the layering time would dominate the total time. Since the layering time is bounded by $O(ElogP)$, this time is considerably lower than applying spectral bisection-based methods from scratch.

Parallel time is considerably more difficult to analyze. The parallel time complexity of the layering step depends on the maximum number of edges any processor has for each level. This can be approximated by $O(\frac{E}{P})$ for each level, assuming the changes to the graph are incremental and that the graph is much larger than the number of processors. As discussed earlier, the parallelization of linear programming requires a broadcast of length proportional to $O(k)$. For small values of k, each linear programming formulation has to be executed on only one processor, else the communication will dominate the total time. Thus the parallel time is proportional to $O(\frac{E}{P} + k^2 f(k) log_k P)$.

The above analysis did not take reconfiguration into account. The cost of reconfiguration requires $O(kd^2)$ time in parallel for every iteration, where d is the average number of partitions to which every partition is connected. The total time is $O(kd^2 \log P)$ for the reconfiguration. This time should not dominate the total time required by the linear programming algorithm.

5 Experimental results

In this section we present experimental results of the linear programming-based incremental partitioning methods presented in the previous section. We will use the term "incremental graph partitioner" (IGP) to refer to the linear programming based algorithm. All our experiments were conducted on the 32-node CM-5 available at NPAC at Syracuse University.

Meshes

We used two sets of adaptive meshes for our experiments. These meshes were generated using the DIME environment [15]. The initial mesh of Set A is given in Figure 14 (a). The other incremental meshes are generated by making refinements in a localized area of the initial mesh. These meshes represent a sequence of refinements in a localized area. The number of nodes in the meshes are 1071, 1096, 1121, 1152, and 1192, respectively.

The partitioning of the initial mesh (1071 nodes) was determined using recursive spectral bisection. This was the partitioning used by algorithm IGP to determine the partitioning of the incremental mesh (1096 nodes). The repartitioning of the next set of refinement (1121, 1152, and 1192 nodes, respectively) was achieved using the partitioning obtained by using the IGP for the previous mesh in the sequence. These meshes are used to test whether IGP is suitable for repartitioning a mesh after several refinements.

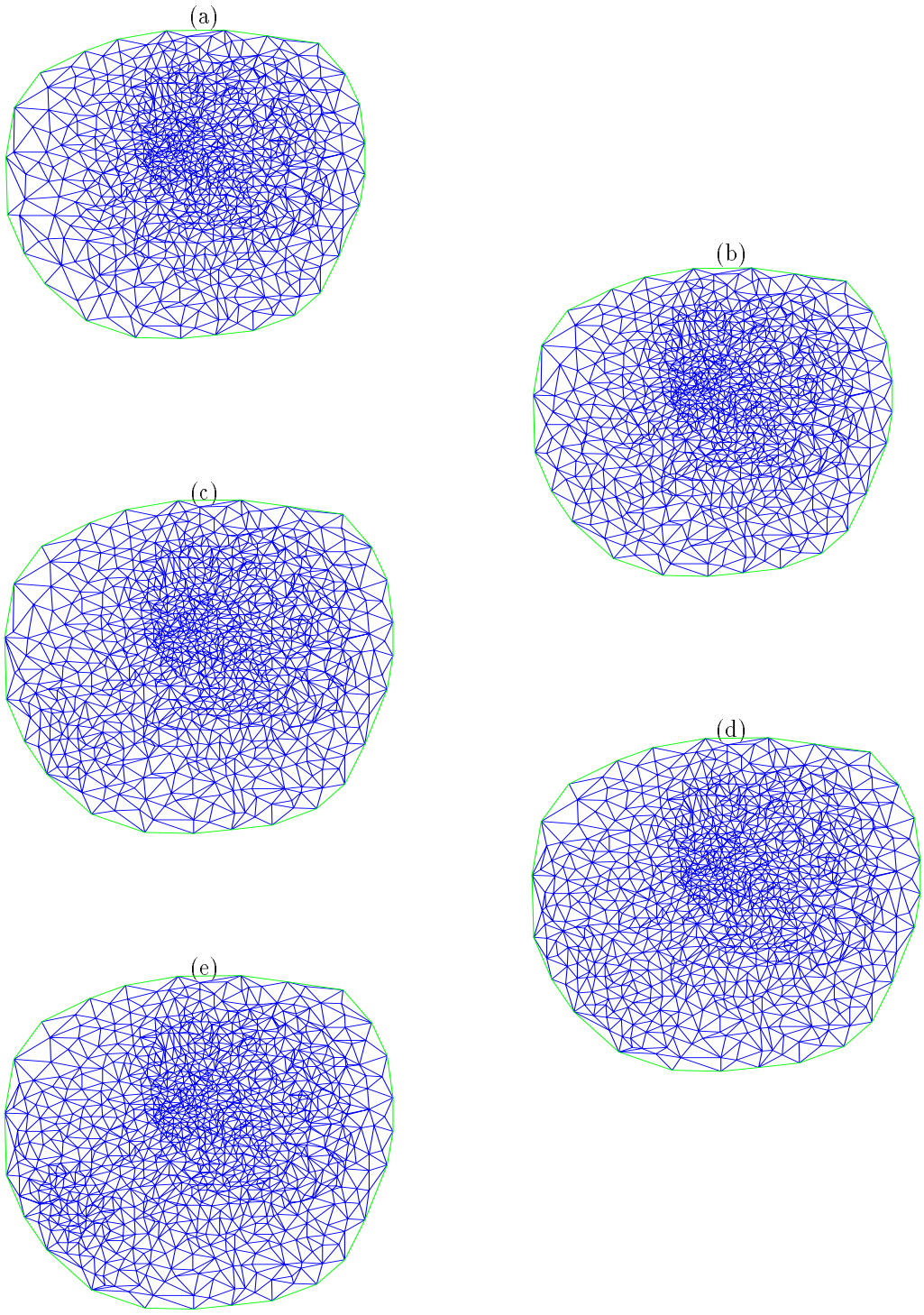


Figure 14: Test graphs set A (a) an irregular graph with 1071 nodes and 3185 edges; (b) graph in (a) with 25 additional nodes; (c) graph in (b) with 25 additional nodes; (d) graph in (c) with 31 additional nodes; (e) graph in (d) with 40 additional nodes.

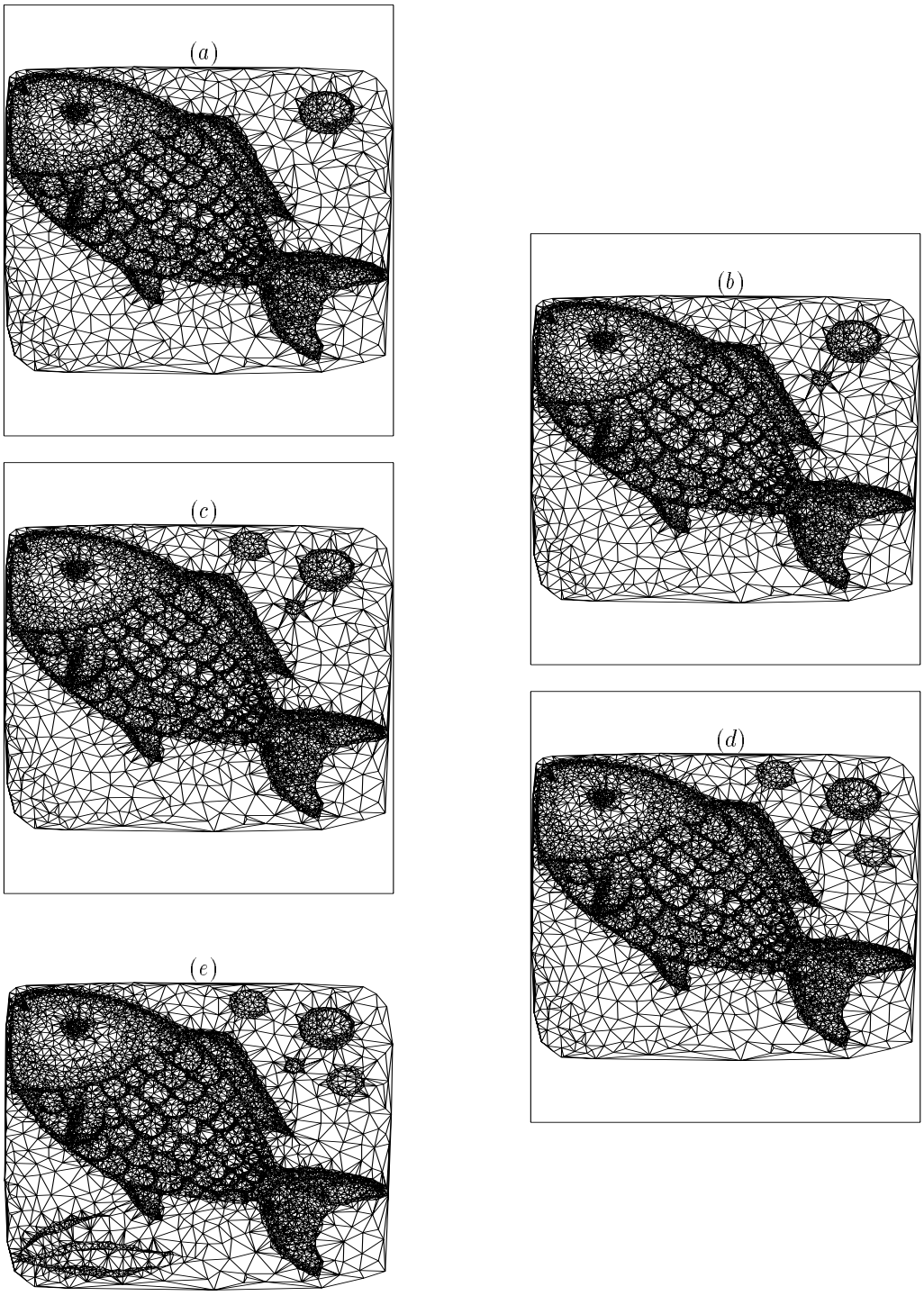


Figure 15: Test graphs Set B (a) a mesh with 10166 nodes and 30471 edges; (b) mesh a with 48 additional nodes; (c) mesh a with 139 additional nodes; (d) mesh a with 229 additional nodes; (e) mesh a with 672 additional nodes.

Results

Initial Graph — Figure 14 (a)					
	$\|V\|$	$\|E\|$	Total Cutset	Max Cutset	Min Cutset
	1071	3185	734	56	35
$\|V\| = 1096$ $\|E\| = 3260$ — Figure 14 (b)					
Partitioner	Time-s	Time-p	Total Cutset	Max Cutset	Min Cutset
Spectral Bisection	31.71	—	733	56	33
IGP	14.75	0.68	747	55	34
IGP with Refinement	16.87	0.88	730	54	34
$\|V\| = 1121$ $\|E\| = 3335$ — Figure 14 (c)					
Partitioner	Time-s	Time-p	Total Cutset	Max Cutset	Min Cutset
Spectral Bisection	34.05	—	732	56	34
IGP	13.63	0.73	752	54	33
IGP with Refinement	16.42	1.05	727	54	33
$\|V\| = 1152$ $\|E\| = 3428$ — Figure 14 (d)					
Partitioner	Time-s	Time-p	Total Cutset	Max Cutset	Min Cutset
Spectral Bisection	34.96	—	716	57	34
IGP	15.89	0.92	757	56	33
IGP with Refinement	18.32	1.28	741	56	33
$\|V\| = 1192$ $\|E\| = 3548$ — Figure 14 (e)					
Partitioner	Time-s	Time-p	Total Cutset	Max Cutset	Min Cutset
Spectral Bisection	38.20	—	774	63	34
IGP	15.69	0.94	815	63	34
IGP with Refinement	18.43	1.26	779	59	34

Time unit in seconds. p - parallel timing on a 32-node CM-5. s - timing on 1-node CM-5.

Figure 16: Incremental graph partitioning using linear programming and its comparison with spectral bisection from scratch for meshes in Figure 14 (Set A).

The next data set (Set B) corresponds to highly irregular meshes with 10166 nodes and 30471 edges. This data set was generated to study the effect of different amounts of new data added to the original mesh. Figures 17 (b), 17 (c), 17 (d), and 17 (e) correspond to meshes with 68, 139, 229, and 672 additional nodes over the mesh in Figure 15.

The results of the one-level IGP for Set A meshes are presented in Figure 16. The results show that, even after multiple refinements, the quality of partitioning achieved is comparable to that achieved by recursive spectral bisection from scratch, thus this method can be used for repartitioning several stages. The time required by repartitioning is about half the time required for partitioning using RSB. The algorithm provides speedup of around 15 to 20 on a 32-node CM-5. Most of the time spent by our algorithm is in the solution of the linear programming formulation using the simplex method. The number of variables and constraints generated by the one-level linear programming algorithm for the load-balancing step for meshes in Figure 16 with $|V| = 1096$ and $|E| = 3260$ for 32 partitions are 188 and 126, respectively.

For the multilevel approach, the linear programming formulation for each subproblem at a given level

Initial Graph — Figure 15 (a)					
	$\|V\|$	$\|E\|$	Total Cutset	Max Cutset	Min Cutset
	10166	30471	2118	171	82

(b) Initial assignment by IGP using the partition of Figure 15 (a')					
$\|V\|$ = 10214 $\|E\|$ = 30615 Total Cutset = 2118 Max load = 361 Min load = 317					
Partitioner	Time-s	Time-p	Total Cutset	Max Cutset	Min Cutset
Spectral Bisection	800.05	—	2137	178	90
IGP before Refinement	13.90	1.01	2139	186	84
IGP after Refinement	24.07	1.83	2040	172	82

(c) Initial assignment by IGP using the partition of Figure 15 (a')					
$\|V\|$ = 10305 $\|E\|$ = 30888 Total Cutset = 2128 Max load = 409 Min load = 317					
Partitioner	Time-s	Time-p	Total Cutset	Max Cutset	Min Cutset
Spectral Bisection	814.36	—	2099	166	87
IGP before Refinement	18.89	1.08	2295	219	93
IGP after Refinement	29.33	2.01	2162	206	85

(d) Initial assignment by IGP using the partition of Figure 15 (a')					
$\|V\|$ = 10395 $\|E\|$ = 31158 Total Cutset = 2162 Max load = 409 Min load = 317					
Partitioner	Time-s	Time-p	Total Cutset	Max Cutset	Min Cutset
Spectral Bisection	853.35	—	2057	169	94
IGP before Refinement (2)	35.98	2.08	2418	256	92
IGP after Refinement	43.86	2.76	2139	190	85

(e) Initial assignment by IGP using the partition of Figure 15 (a')					
$\|V\|$ = 10838 $\|E\|$ = 32487 Total Cutset = 2536 Max load = 523 Min load = 317					
Partitioner	Time-s	Time-p	Total Cutset	Max Cutset	Min Cutset
Spectral Bisection	904.81	—	2158	158	94
IGP before Refinement (3)	76.78	3.66	2572	301	102
IGP after Refinement	89.48	4.39	2270	237	96

Time unit in seconds. p - parallel timing on a 32-node CM-5. s - timing on 1-node CM-5.

Figure 17: Incremental graph partitioning using linear programming and its comparison with spectral bisection from scratch for meshes in Figure 15 (Set B).

was solved by assigning a subset of processors. Table 19 gives the time required for different algorithms and the quality of partitioning achieved for different numbers of levels. A $4 \times 4 \times 2$-based repartitioning implies that the repartitioning was performed in three stages with decomposition into 4, 4, 2 partitions, respectively. The results are presented in Figure 19. The solution qualities of multilevel algorithms show an insignificant deterioration in number of cross edges and a considerable reduction in total time.

The partitioning achieved by algorithm IGP for Set B meshes in Figure 18 using the partition of mesh in Figure 15 (a) is given in Figure 17. The number of stages required (by choosing an appropriate value of Δ, as described in section 2.3) were 1, 1, 2, and 3, respectively.[4] It is worth noting that, although the load imbalance created by the additional nodes was severe, the quality of partitioning achieved for each case was close to that of applying recursive spectral bisection from scratch. Further, the sequential time is at least an order of magnitude better than that of recursive spectral bisection. The CM-5 implementation improved the time required by a factor of 15 to 20. The time required for repartitioning Figure 17 (b) and Figure 17 (c) is close to that required for meshes in Figure 14. The timings for meshes in Figure 17 (d) and 17 (e) are larger because they use multiple stages. The time can be reduced by using a multilevel approach (Figure 20). However, the time reduction is relatively small (from 24.07 seconds to 6.70 seconds for a two-level approach). Increasing the number of levels increases the total time as the cost of layering increases. The time reduction for the last mesh (10838 nodes) is largely due to the reduction of the number of stages used in the multilevel algorithm (Section 3.3). For almost all cases a speedup of 15 to 25 was achieved on a 32-node CM-5.

Figure 21 and Figure 22 show the detailed timing for different steps for the mesh in Figure 14 (d) and mesh in Figure 15 (b) of the sequential and parallel versions of the repartitioning algorithm, respectively. Clearly, the time spent in reconfiguration is negligible compared to the total execution time. Also, the time spent for linear programming in a multilevel algorithm is much less than that in a single-level algorithm. The results also show that the time for the linear programming remains approximately the same for both meshes, while the time for layering is proportionally larger. For the multilevel parallel algorithm, the time for layering is comparable with the time spent on linear programming for the smaller mesh, while it dominates the time for the larger mesh. Since the layering term is $O(levels\frac{E}{P})$, these results support the complexity analysis in the previous section. The time spent on reconfiguration is negligible compared to the total time.

6 Conclusions

In this paper we have presented novel linear programming-based formulations for solving incremental graph-partitioning problems. The quality of partitioning produced by our methods is close to that achieved by applying the best partitioning methods from scratch. Further, the time needed is a small fraction of the latter and our algorithms are inherently parallel. We believe the methods described in this paper are of critical importance in the parallelization of adaptive and incremental problems.

[4] The number of stages chosen were by trial and error, but can be determined by the load imbalance.

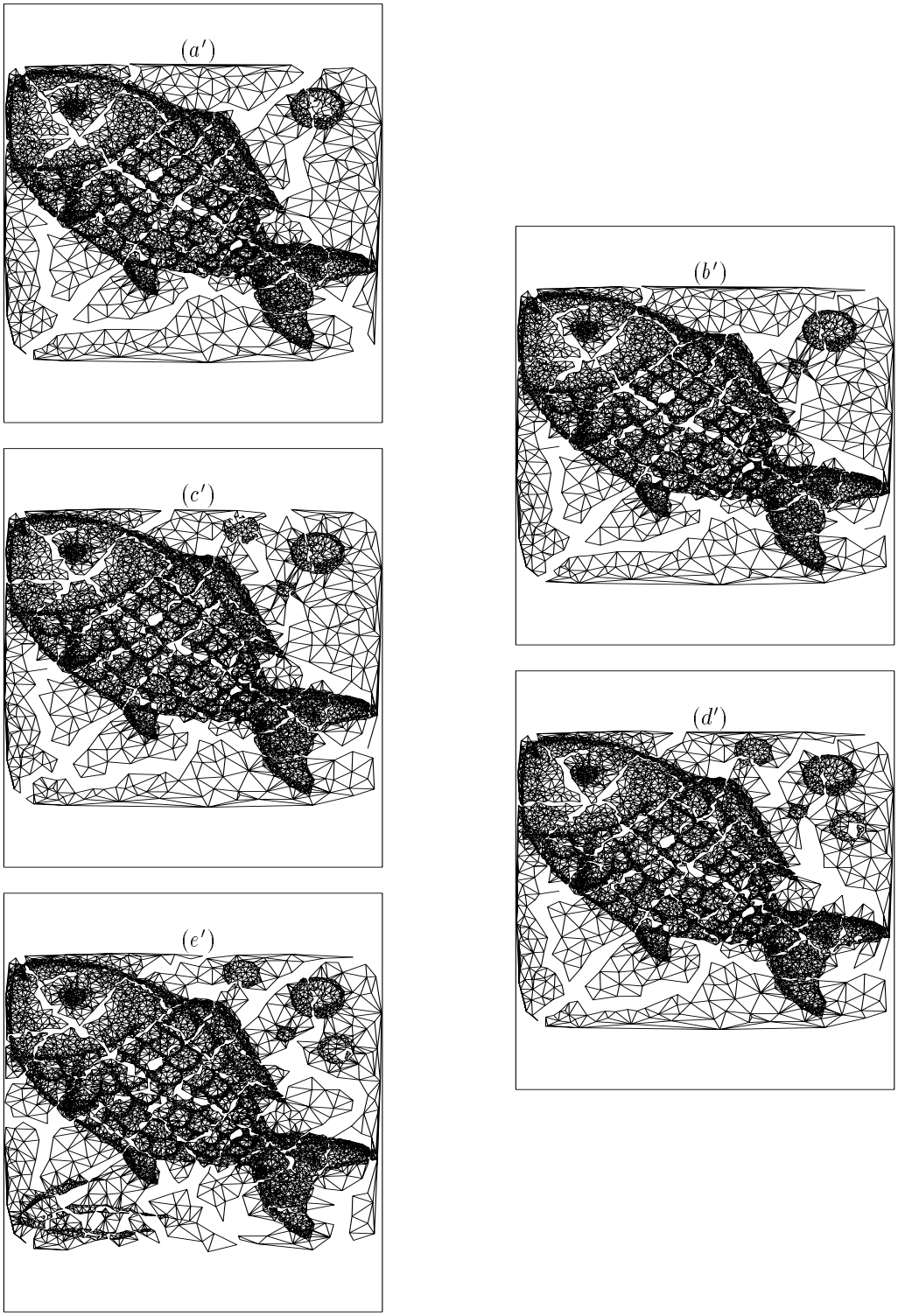


Figure 18: (a') Partitions using RSB; (b') partitions using IGP starting from a'; (c') partitions using IGP starting from a'; (d') partitions using IGP starting from a'; (e') partitions using IGP starting from a'.

Graph	Level	Description	Time-s	Time-p	Total Cutset
	1	32	16.87	0.88	730
$\|V\|$=1096	2	8x4	1.16	0.58	740
$\|E\|$=3260	3	4x4x2	1.27	0.23	745
	1	32	16.42	1.02	727
$\|V\|$=1121	2	8x4	1.60	0.64	752
$\|E\|$=3335	3	4x4x2	1.27	0.22	766
	1	32	18.32	1.28	741
$\|V\|$=1152	2	8x4	1.58	0.76	758
$\|E\|$=3428	3	4x4x2	1.28	0.21	741
	1	32	18.43	1.26	779
$\|V\|$=1192	2	8x4	1.44	0.75	816
$\|E\|$=3548	3	4x4x2	1.33	0.20	811

Time unit in seconds on CM-5.

Figure 19: Incremental multilevel graph partitioning using linear programming and its comparison with single-level graph partitioning for the sequence of graphs in Figure 14.

Graph	Level	Description	Time-s	Time-p	Total Cutset
	1	32	24.07	1.83	2040
$\|V\|$=10214	2	8x4	6.70	0.73	2099
$\|E\|$=30615	3	4x4x2	8.48	0.40	2067
	1	32	29.33	2.01	2162
$\|V\|$=10305	2	8x4	6.89	0.73	2170
$\|E\|$=30888	3	4x4x2	8.40	0.40	2176
	1	32	43.86	2.76	2139
$\|V\|$=10395	2	8x4	6.80	0.81	2220
$\|E\|$=31158	3	4x4x2	8.61	0.44	2214
	1	32	89.48	4.39	2270
$\|V\|$=10838	2	8x4	7.43	0.99	2408
$\|E\|$=32487	3	4x4x2	9.07	0.48	2337

Time unit in seconds on CM-5.

Figure 20: Incremental multilevel graph partitioning using linear programming and its comparison with single-level graph partitioning for the sequence of meshes in Figure 15.

$\|V\|$ = 1152, $\|E\|$ = 3428, in Figure 14 (d)										
Level	Reconfiguration	Layering			Linear programming			Total		
		B	R	T	B	R	T	B	R	T
1	—	0.29	0.21	0.50	15.89	1.90	17.79	16.19	2.12	18.32
2	0.01	0.47	0.34	0.82	0.13	0.62	0.78	0.61	1.00	1.65
3	0.01	0.65	0.47	1.13	0.045	0.09	0.17	0.72	0.60	1.31
$\|V\|$ = 10214, $\|E\|$ = 30615, in Figure 15 (b)										
Level	Reconfiguration	Layering			Linear programming			Total		
		B	R	T	B	R	T	B	R	T
1	—	1.45	1.57	3.02	12.53	8.52	21.05	13.98	10.09	24.07
2	0.01	2.80	2.84	5.64	0.24	0.76	1.10	3.50	4.22	6.70
3	0.01	4.26	3.92	8.18	0.05	0.14	0.19	4.33	4.12	8.48

Time in seconds

B – Balancing. R – Refinement. T – Total.

Figure 21: Time required for different steps in the sequential repartitioning algorithm.

$\|V\|$ = 1152, $\|E\|$ = 3428, in Figure 14 (d)													
Level	Reconfiguration	Layering			Linear programming			Data movement			Total		
		B	R	T	B	R	T	B	R	T	B	R	T
1	—	0.01	0.02	0.03	0.82	0.41	1.23	0.01	0.01	0.02	0.84	0.43	1.27
2	0.01	0.02	0.01	0.03	0.11	0.61	0.72	0.01	0.02	0.03	0.14	0.65	0.80
3	0.01	0.02	0.02	0.04	0.03	0.07	0.10	0.02	0.03	0.05	0.07	0.12	0.20
$\|V\|$ = 10214, $\|E\|$ = 30615, in Figure 15 (b)													
Level	Reconfiguration	Layering			Linear programming			Data movement			Total		
		B	R	T	B	R	T	B	R	T	B	R	T
1	—	0.06	0.03	0.09	0.84	0.87	1.71	0.01	0.02	0.03	0.91	0.92	1.83
2	0.01	0.10	0.11	0.20	0.08	0.39	0.48	0.02	0.05	0.07	0.21	0.51	0.73
3	0.01	0.15	0.13	0.28	0.02	0.07	0.09	0.02	0.06	0.08	0.18	0.22	0.40

Time in seconds

B – Balancing. R – Refinement. T – Total.

Figure 22: Time required for different steps in the parallel repartitioning algorithm (on a 32-node CM-5).

References

[1] I. Angus, G. Fox, J. Kim, and D. Walker. *Solving Problems on Concurrent Processors*, volume 2. Prentice Hall, Englewood Cliffs, NJ, 1990.

[2] Alok Choudhary, Geoffrey C. Fox, Seema Hiranandani, Ken Kennedy, Charles Koelbel, Sanjay Ranka, and Joel Saltz. Software Support for Irregular and Loosely Synchronous Problems. In *Proceedings of the Conference on High Performance Computing for Flight Vehicles*, 1992. To appear.

[3] F. Ercal. *Heuristic Approaches to Task Allocation for Parallel Computing.* Ph.D. thesis, Ohio State University, 1988.

[4] G. C. Fox and W. Furmanski. Load Balancing Loosely Synchronous Problems with a Neural Network. 1988.

[5] G. C. Fox, M. Johnson, G. Lyzenga, S. Otto, J. Salmon, and D. Walker. *Solving Problems on Concurrent Processors*, volume 1. Prentice Hall, Englewood Cliffs, NJ, 1988.

[6] Geoffrey C. Fox. *Graphical Approach to Load Balancing and Sparse Matrix Vector Multiplication on the Hypercube.* 1988. Ed. M. Schultz, Springer-Verlag, Berlin.

[7] Bruce Hendrickson and Robert Leland. An Improved Spectral Graph Partitioning Algorithm for Mapping Parallel Computations. Technical Report SAND92-1460, Sandia National Laboratories, Albuquerque, NM 87185, 1992.

[8] Bruce Hendrickson and Robert Leland. Multidimensional Spectral Load Balancing. Technical Report SAND93-0074, Sandia National Laboratories, Albuquerque, NM 87185, 1993.

[9] Harpal Maini, Kishan Mehrotra, Chilukuri Mohan, and Sanjay Ranka. Genetic Algorithms for Graph Partitioning and Incremental Graph Partitioning. In *Proceedings of Supercomputing '94*, November 1994.

[10] Nashat Mansour. *Physical Optimization Algorithms for Mapping Data to Distributed-Memory Multiprocessors.* PhD thesis, Syracuse University, NY, 1993.

[11] B. Nour-Omid, A. Raefsky, and G. Lyzenga. Solving Finite Element Equations on Current Computers. *Parallel Computations and Their Impact on Mechanics*, 1986.

[12] Chao-Wei Ou, Sanjay Ranka, and Geoffrey Fox. Fast Mapping And Remapping Algorithm For Irregular and Adaptive Problems. In *Proceedings of the 1993 International Conference on Parallel and Distributed Systems*, Taipei, Taiwan, December 1993.

[13] A. Pothen, H. Simon, and K-P Liou. Partitioning Sparse Matrices with Eigenvectors of Graphs. *SIAM Journal of Matrix Analysis and Application*, 11(3), July 1990.

[14] H. Simon. Partitioning of Unstructured Mesh Problems for Parallel Processing. In *Proceedings of the Conference on Parallel Methods on Large Scale Structural Analysis and Physics Applications.* Permagon Press, 1991.

[15] R.D. Williams. *DIME: Distributed Irregular Mesh Enviroment.* California Institute of Technology, February 1990.

[16] R.D. Williams. Performance of Dynamic Load-Balancing Algorithm for Unstructured Mesh Calculations. *Concurrency*, 3:457–481, 1991.