1994

Parallel Incremental Graph Partitioning Using Linear Programming

Chao Wei Ou
Syracuse University

Sanjay Ranka
Syracuse University

Parallel Incremental Graph Partitioning Using Linear Programming*

Chao-Wei Ou and Sanjay Ranka
School of Computer and Information Science
Syracuse University
Syracuse, NY 13244-4100

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 from scratch. 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 the 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.

*This research was supported in part by DARPA under contract #DABT63-91-C-0028.

Several suboptimal methods have been suggested for finding good solutions to the graph-partitioning problem. Important heuristics include recursive coordinate bisection, recursive graph bisection, recursive spectral bisection, mincut-based methods, clustering techniques, geometry-based mapping, block-based spatial decomposition, and scattered decomposition [3, 4, 1, 6, 5, 8, 9, 10, 12].

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. In this paper we concentrate on methods for which such information is not available, and which therefore have wider applicability. Our incremental graph-partitioning algorithm uses linear programming. 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 the linear programming-based incremental graph partitioning. Experimental results of our methods on sample meshes are described in Section 4. Conclusions are given in Section 5.

1.1 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

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

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

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

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, i.e., the mapping function M is available such that

$$B(1) \approx B(2) \approx B(3) \approx \cdots \approx B(q-1) \qquad (3)$$

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

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

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

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

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.

2 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.

2.1 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,

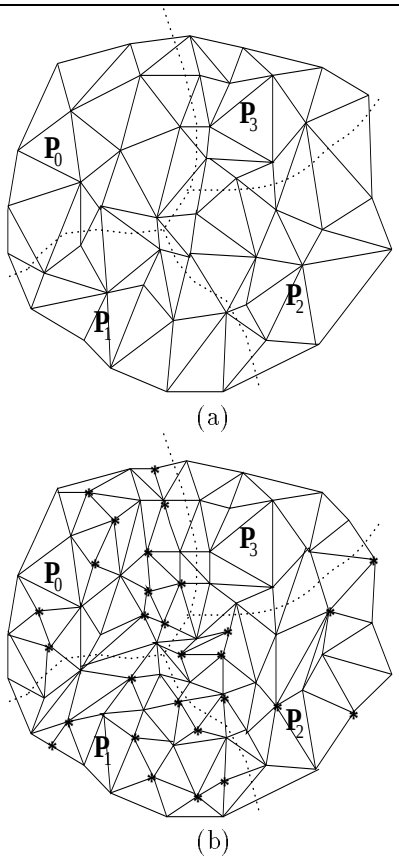

Figure 2: (a) Initial Graph (b) Incremental Graph (New vertices are shown by "*").

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.

2.2 Layering each partition

{ $map[v[j]]$ 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 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

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).

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

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.

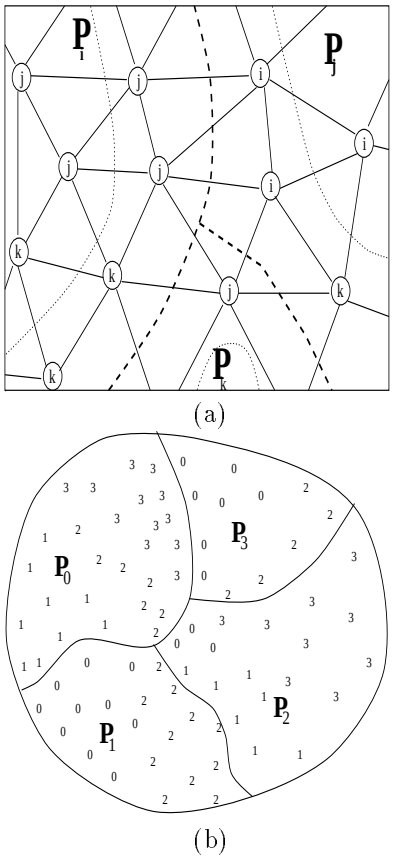

Figure 4: Labeling the nodes of a graph to the closest outside partition. (a) A microscopic view of the layering for a graph near the boundary of three partitions. (b) Layering of the graph in Figure 2 (b); no edges are shown.

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}.

2.3 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 methods for load balancing. However, since one of our goals is to minimize the 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.

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.

The above set of constraints may not have a feasible solution. One approach is to relax the constraint in (11) and not have $l_{ij} \le \alpha_{ij}$ as a constraint. Clearly,

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

Constraints in (11):

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

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).

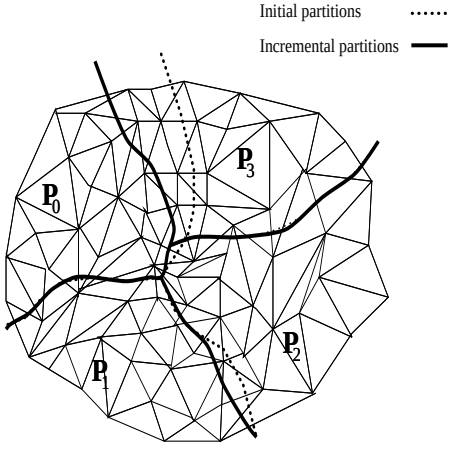

Figure 6: The new partition of the graph in Figure 2 (b) after the **Load Balancing** step.

this would achieve load balance but may lead to major modifications in the mapping. Another approach is to replace the constraint in (12) by:

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

Assuming $C > \Delta > 1$, this would not achieve load balancing in one step, but several such steps can be applied to achieve load balancing. 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.

2.4 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 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 $in\ (k)$ 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.

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

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

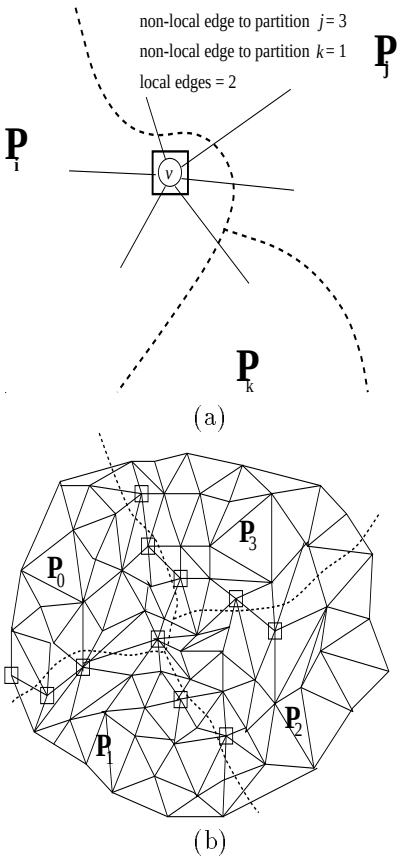

Figure 7: Choosing vertices for refinement. (a) Microscopic view of a vertex which can be moved from partition P_i to P_j, reduceing the number of cross edges. (b) The set of vertices with the above property in the partition of Figure 6.

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

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

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

Constraint (15)

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

Load Balancing Constraint (16)

$$\begin{array}{c} l_{01} + l_{02} + l_{03} - l_{10} - l_{20} - l_{30} = 0 \\ l_{10} + l_{12} - l_{01} - l_{21} = 0 \\ -l_{20} - l_{21} - l_{23} + l_{02} + l_{12} + l_{32} = 0 \\ -l_{30} - l_{32} + l_{03} + l_{23} = 0 \end{array}$$

Solution using Simplex Method

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

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

This refining 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 ($l_{ij} < b_{ij}$); 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.

3 Experimental results

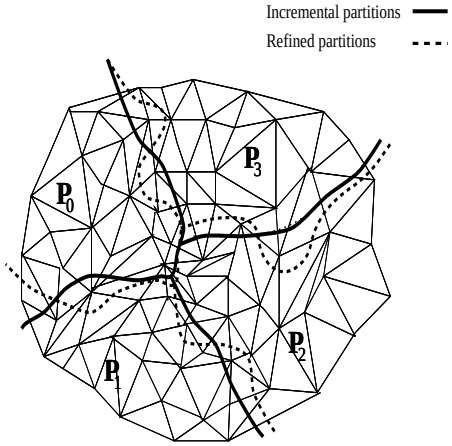

Figure 9: The new partition of the graph in Figure 6 after the **Refinement** step.

In this section, we present experimental results of the linear programming-based incremental partitioning presented in the previous section (we will use the term Incremental Graph Partitioner (IGP) to refer to this algorithm). The timings are given for 32 partitions on a 1-node and 32-node CM-5.

We have used two sets of adaptive meshes for our experiments. These meshes were generated using the DIME environment [11]. The initial mesh of the first set is given in Figure 10. 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 (size 1071 nodes) was determined using Recursive Spectral bisection. This was the partitioning used by algorithm IGP to determine the partition of the incremental mesh (of size 1096). The repartitioning of the next set of refinement (with 1121, 1152, and 1192 nodes, respectively) was achieved using the partitioning obtained by using the IGP for the previous mesh in the sequence. 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 for several stages. The time required by repartitioning is about half of 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 so-

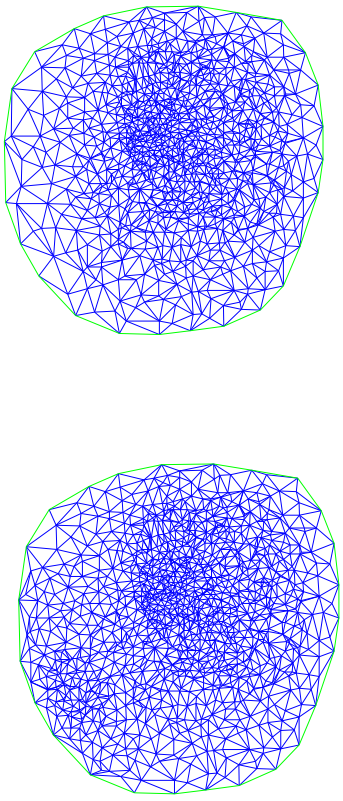

Figure 10: Test graph A an irregular graph with 1071 nodes and 3185 edges. The refinement graph with 1192 nodes and 3548 edges.

Initial Graph — Figure 10			Cutset		
Partitioner	$\|V\|$	$\|E\|$	Total	Max	Min
SB	1071	3185	734	56	35
$\|V\|$ = 1096 $\|E\|$ = 3260			Cutset		
Partitioner	Time-s	Time-p	Total	Max	Min
SB	31.71	—	733	56	33
IGP	14.75	0.68	747	55	34
IGPR	16.87	0.88	730	54	34
$\|V\|$ = 1121 $\|E\|$ = 3335			Cutset		
Partitioner	Time-s	Time-p	Total	Max	Min
SB	34.05	—	732	56	34
IGP	13.63	0.73	752	54	33
IGPR	16.42	1.05	727	54	33
$\|V\|$ = 1152 $\|E\|$ = 3428			Cutset		
Partitioner	Time-s	Time-p	Total	Max	Min
SB	34.96	—	716	57	34
IGP	15.89	0.92	757	56	33
IGPR	18.32	1.28	741	56	33
$\|V\|$ = 1192 $\|E\|$ = 3548			Cutset		
Partitioner	Time-s	Time-p	Total	Max	Min
SB	38.20	—	774	63	34
IGP	15.69	0.94	815	63	34
IGPR	18.43	1.26	779	59	34

Time unit in seconds.
p - parallel timing on a 32-node CM-5.
s - timing on a one-node CM-5.
SB - Spectral Bisection.
IGP - Incremental Graph Partitioner.
IGPR - Incremental Graph Partitioner with Refinement.

Figure 11: Incremental graph partitioning using linear programming and its comparison with spectral bisection from scratch for meshes in Figure 10.

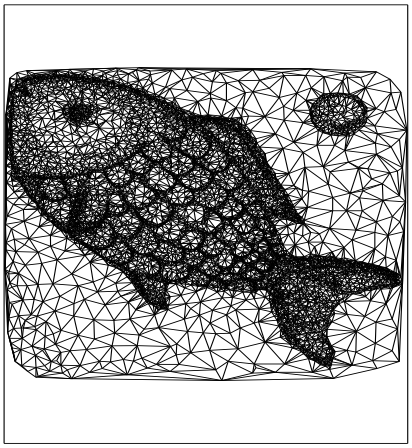

Figure 12: A mesh with 10166 nodes and 30471 edges.

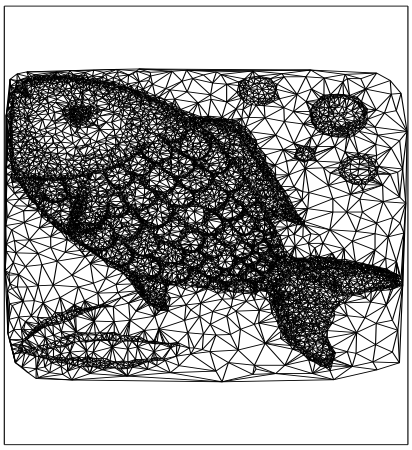

Figure 13: A refinement of mesh in Figure 12 with 672 extra nodes.

lution of the linear programming formulation using the simplex method. The cost of the simplex method depends on the number of variables (v) and the number of constraints (c). Each iteration in the dense matrix formulation requires time proportional to the $O(vc)$. The value of v and c depend largely on the number of partitions and the number of edges between the partitions (corresponding to e_{ij} and l_{ij} as described in section 2.3 and section 2.4, respectively). The values of v and c for the formulation corresponding to performing the load balancing step for mesh in Figure 11 with $|V| = 1096$ and $|E| = 3260$ for 32 partitions are 188 and 126, respectively These costs are independent of the number of vertices in the mesh and depend on the number of partitions. Thus, for large meshes the performance should be much better. Our software currently implements the simplex method using a dense matrix formulation. Since the matrix is highly sparse, this cost can be substantially reduced by using a sparse representation. Clearly, the latter would be more difficult to parallelize. Another option is to use a multilevel approach and apply incremental partitioning recursively. We are currently exploring this approach. Since most of the time (even for large meshes) is spent on the solution of the linear programming using the simplex method, any improvements in the time required will have a major impact on the total time required for partitioning.

The next data set corresponds to highly irregular mesh 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 14 (b), 14 (c), 14 (d), and 14 (e) correspond to meshs

(a) Initial Graph — Figure 12			Cutset		
Partitioner	$\|V\|$	$\|E\|$	Total	Max	Min
	10166	30471	2118	171	82
(b) $\|V\| = 10214$ $\|E\| = 30615$			Cutset		
Partitioner	Time-s	Time-p	Total	Max	Min
SB	800.05	—	2137	178	90
IGP	13.90	1.01	2139	186	84
IGPR	24.07	1.83	2040	172	82
(c) $\|V\| = 10305$ $\|E\| = 30888$			Cutset		
Partitioner	Time-s	Time-p	Total	Max	Min
SB	814.36	—	2099	166	87
IGP	18.89	1.08	2295	219	93
IGPR	29.33	2.01	2162	206	85
(d) $\|V\| = 10395$ $\|E\| = 31158$			Cutset		
Partitioner	Time-s	Time-p	Total	Max	Min
SB	853.35	—	2057	169	94
IGP(2)	35.98	2.08	2418	256	92
IGPR	43.86	2.76	2139	190	85
(e) $\|V\| = 10838$ $\|E\| = 32487$			Cutset		
Partitioner	Time-s	Time-p	Total	Max	Min
SB	904.81	—	2158	158	94
IGP(3)	76.78	3.66	2572	301	102
IGPR	89.48	4.39	2270	237	96
Time unit in seconds.					
p - parallel timing on a 32-node CM-5.					
s - timing on a one-node CM-5.					
SB - Spectral Bisection.					
IGP - Incremental Graph Partitioner.					
IGPR - Incremental Graph Partitioner with Refinement.					

Figure 14: Incremental graph partitioning using linear programming and its comparison with spectral bisection from scratch for meshes in Figure 12 and Figure 13.

with 68, 139, 229, and 672 additional nodes over the mesh in Figure 12. The partitioning achieved by algorithm IGP for mesh in Figure 13 using the partition of mesh in Figure 12 for mesh is given in Figure 14. 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. [2] It is worth noting that although the load imbalance created by the additional nodes was severe, the quality of partitioning achieved for each of the cases 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 14 (b) and Figure 14 (c) is close to that required for meshes in Figure 10. The timings for meshes in Figure 14 (d) and 14 (e) are larger because they use multiple stages.

The above results show that the IGP at a fraction of the cost, can be effectively used for repartitioning to achieve solutions similar in quality to those obtained by applying recursive spectral bisection from scratch. Further, the algorithm can be parallelized effectively.

4 Conclusions

In this paper we have presented a novel linear programming-based formulation 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 to the parallelization of the adaptive and incremental problems described earlier.

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. M. Schultz, Ed., Springer-Verlag, Berlin.

[7] Harpal Maini, Kishan Mehrotra, Chilukuri Mohan, and Sanjay Ranka. *Genetic Algorithms for Graph Partitioning and Incremental Graph Partitioning. Supercomputing '94*

[8] S. Nolting. Nonlinear Adaptive Finite Element Systems on Distributed Memory Computers. In *Proceedings of European Distributed Memory Computing Conference*, April 1991.

[9] 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.

[10] 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.

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

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

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