Heuristic algorithms for automatic graph partitioning *

T. Goehring †and Y. Saad ‡

April 24, 1994

Abstract

Practical implementations of the Finite Element method on distributed memory multi-computer systems necessitate the use of partitioning tools to subdivide the mesh into sub-meshes of roughly equal size. Graph partitioning algorithms are mandatory when implementing distributed sparse matrix methods or domain decomposition techniques for irregularly structured problems, on parallel computers. We propose a class of algorithms which are based on level set expansions from a number of center nodes. A critical component of these methods is the location of these centers. We present a number of different strategies for finding centers which lead to good-quality partitionings.

*Work supported in part by ARPA under grant NIST 60NANB2D1272, in part by NSF grant CCR-9214116, and by the Minnesota Supercomputer Institute.

†Department of Computer Science, University of Minnesota, Minneapolis 55455

‡Department of Computer Science, and Minnesota Supercomputer Institute, University of Minnesota, Minneapolis 55455

1 Introduction

Domain decomposition has emerged as a quite general and efficient means of solving partial differential equations on parallel computers. Typically, a domain is partitioned into several subdomains and some technique allows us to recover the global solution by a succession of solutions of independent subproblems associated with all the subdomains. Each processor handles one or more subdomains in the partition.

In order to implement a domain decomposition approach we need a number of numerical and non-numerical tools for performing the preprocessing tasks required to decompose a domain and map it into processors, as well as to set up the various data structures. It is important to be able to perform such tasks automatically. One of the basic tasks to be performed is to find a decomposition of the graph into s subgraphs whose union is equal to the original graph. Ideally, the subdomains should have roughly equal size, although criteria of load balancing other than subdomain size can also be used. Other similar tools that must be developed in any software library for implementing domain decomposition algorithms are the following: (a) building a coarse mesh given the original fine mesh (see justification later); (b) finding a good subdomain to processor mapping (architecture dependent), once the partition array is found; and (c) coloring the subdomains such that two neighboring subdomains have different colors (d) Preprocessing to set-up the data structures for subsequent Domain Decomposition iterations.

One of the the first steps in implementing domain decomposition type methods is to partition an initial mesh into a number of submeshes and to map these submeshes into the processors. This problem has recently become the focus of much attention [10, 5, 11, 9, 7, 6, 2, 12, 1] mainly because a rapid progress in the practical implementations of finite element methods on parallel computers is currently being achieved and such tools are becoming indispensable. The criterion used in determining an appropriate mesh partitioning is often that of load balancing, i.e., the work that would result from a given partitioning should lead to more or less the same amount of work at each iteration of the domain decomposition approach. Another important criterion that can be used is one which focuses on communication costs. We can attempt to find partitioners that will minimize communication costs during a typical iterative process.

There have been three popular approaches used for partitioning a graph. The algorithm proposed by Pothen, Simon and Liu [11] is a 'spectral analysis' algorithm which is based on the concept of eigenvectors of a graph. Leet, Peyton and Sincovec [9] later extended the algorithm and added to it the capability of handling arbitrary number of partitions and by proposing a parallel implementation. The approach taken by Chrisochoides et al. [5, 4, 3] is a geometrical one and exploits the coordinates of the mesh. The algorithm presented by Liu [10] uses graph theory concepts and exploits node separators. It concentrates on heuristics for finding good node separators. The algorithms we propose are also in this third class, i.e., they are based on the adjacency graph of the sparse matrix, and exploit level sets or breadth-first search traversals in graph theory terminology. We will consider several different approaches and compare

them.

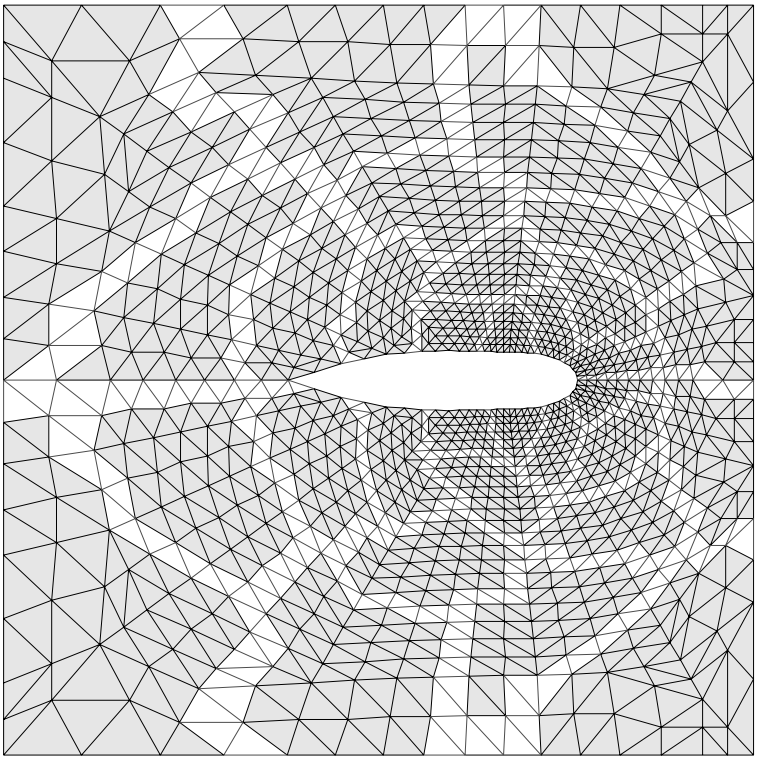

Figure 1: Recursive Spectral Bisection Algorithm

The main idea of the class of techniques we will present is to perform a level set expansion from a number of different nodes called centers. The level set expansion is a multinode breadth-first traversal. It starts from the centers and traverses their immediate neighbors, the set of which constitutes the first level. Then the set of all the neighbors of all the nodes of the first level are traversed. These nodes will form the second level. The algorithm continues until all nodes are traversed. Each time a new node is traversed, it is labeled by the label of the parent node from which it was reached. At the end of the process each node will have the label of the subgraph to which it belongs. It is clear that the performance of this algorithm will depend essentially on the location of the centers relative to each other. Centers that are clustered together will yield a poor graph partition. A heuristic criterion we can use to define a good distribution of centers is a global distance between the centers. Then we can find heuristic algorithms to maximize this distance.

2 Graph Partitioning Concepts

Many scientific problems involve the solution of sparse, linear or nonlinear equations. The dependency between the unknowns in the problem is often conveniently rep-

resented by a graph. The nodes of the graph may actually represent vertices of a physical mesh for a discretized Partial Differential Equation. However the concept is more general and many techniques that have been developed for PDE's can be extended to more general sparse matrices.

The very first task that a programmer faces when solving a problem on a parallel computer, whether it is a dense or a sparse system of equations, is to decide how to map the data into the processors. For shared memory and SIMD computers, directives are often provided to help select among a few possible layouts of the data in memory. Distributed memory computers are far more general and allow one to map the data arbitrarily.

The most general way of defining a node-to-processor mapping is to set up a list for each processor, containing all the nodes that are mapped to that processor. This can be efficiently implemented with a pointer-list data structure of the following form.

PTR =	1 5 7 9 13
LIST =	1 2 5 6 3 4 9 10 7 8 11 12

The LIST array lists the vertices mapped to processor followed by those mapped to processor 2, etc... The PTR array is a pointer array that points to the beginning of the sublist for each processor in the array list, in the natural order. Thus, in the above example, the vertices 1,2,5,6, will be mapped to Processor 1, vertices 3,4, will be mapped to Processor 2, vertices 7,8,11,12 will be mapped to Processor 3 will be mapped and vertices 9, 10, will be mapped to Processor 4. This is illustrated in Figure 3.

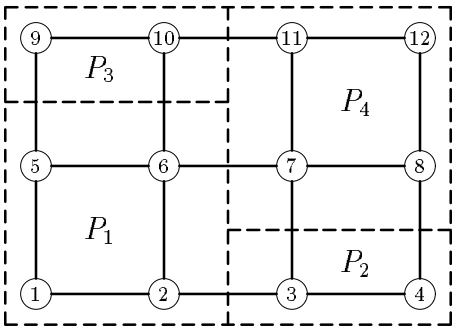

Figure 2.1 Mapping of a 4×3 mesh to 4 processors.

There are two issues related to the definition of a mapping as defined in this section. First, we would like to be able to find a good partitioning of the original graph into subgraphs. In our approach this will require only graph theory tools. The

goal here is to subdivide the graph into smaller subgraphs in such a way as to achieve a good load balancing of the work among the processors and ensure that the ratio of communication over computation with external processors is small for the given task.

A second problem which is architecture dependent, is to find a good *mapping* of the subdomains or subgraphs to the processors. In this paper, we leave this as a separate problem and will uncouple it from the graph partitioning problem. Our motivation is that it is far easier to first find an architecture-independent decomposition, and then, in a second phase, find the proper mapping to the given architecture. Currently, many parallel computers are built with the goal of attempting to make the system look as a fully connected computer in which communication would not depend too much on the distance between processors.

Although the graph partitioning problems which we consider do not directly invoke the underlying architecture, the partitioning algorithm can clearly take advantage of a measure of cost which can take into account load computation as well as communication costs. We can exploit different weight functions for the vertices, for vertex-based partitionings. We can also search for a good mapping which will minimize communication costs given some knowledge on the architecture. In this paper we will not consider these refinements.

3 One-way and two-way graph partitioning

One of the simplest form of graph partitioning is the 'one-way dissection' orderings as defined by George and Liu [8]. This technique is remarkably simple. It consists of performing a breadth-first search traversal of the graph from a given node and assigning a certain number of consecutive levels to each subdomain. We can also start the Breadth-First Search from a set of nodes instead of a single node.

For the sake of completeness, we first describe the Breadth First Search algorithm.

ALGORITHM **3.1 Breadth-First-Search**

1. Input: initial node i_1*; Output: level sets.*
2 **Start:** *Define the 1-st level set* $levset = \{i_1\}$*;*
3. *Set* $next = 2$*;* $marker(i_1) = 1$*;* $nlev = 1$*.*
4. **Loop:** *While* $(next < n)$ *Do*
5. $Next\_levset = \phi$
6. *For each* j *in* $levset$ *Do*
7. *For each neighbor* k *of* j *s.t.* $marker(k) = 0$ *Do*
8. *Append* k *to the set* $Next\_levset$
9. $marker(k) := 1$*;*
10. $next = next + 1$ *;*
11. *EndDo*
12. *EndDo*
13. $levset := Next\_levset$

14. *nlev := nlev+1;*
15. *EndWhile*

The above algorithm is at the basis of many of the techniques to be described in this paper. We note that the algorithm does not need to start with a single node. We can, for example, start with a known level, i.e., a set of nodes forming a string of connected nodes. The one-way partitioning algorithm simply traverse the nodes in the Breadth-First-Search order and assigns nodes to partitions until a given number of assigned nodes is reached. Note that a variation of this algorithm gives rise to the well-known Cuthill McKee algorithm. This variation consists of specifying the order in which to traverse the nodes of a previous level in line 7. In the Cuthill-Mc Kee variation, this order is by increasing degree.

ALGORITHM **3.2** *One-way-partitioning using level-sets*

1. Input: nlev, levels, ip (number of nodes per partition)
2. Output: ndom, node-to-processor list.
3. **Start:** *ndom = 1; siz = 0;*
4. **Loop:** *For lev =1, nlev Do*
5. *For each j in levels(lev) Do*
6. *add j to dom(ndom); siz = siz+1;*
7. *If (siz .ge. ip) then*
8. *ndom = ndom+1; siz = 0;*
9. *EndIf*
10. *EndDo*

In an actual implementation, some care must be exercised to obtain equal sized domains, and to avoid having a last subdomain consisting of the points that are left-over in the very last step. For example, if $n = 101$ and $ip = 10$, we will have 10 subdomains of size 10 each and one of size one. This can be prevented by adjusting the ip parameter as we proceed.

Two-way partitioning consists of two applications of one-way partitionings. In a first pass we determine a one-way partition. Then in each of the subdomains created, we will again perform a one-way partitioning.

ALGORITHM **3.3** *Two-way-partitioning using level-sets*

1. Input: Graph;
2. Output: ndom, node-to-processor list.
3. Call BFS to get a one way partitioning of global graph
4. **Loop:** *For idom=1,ndom Do*
5. *Select an initial point from set lev(i)*

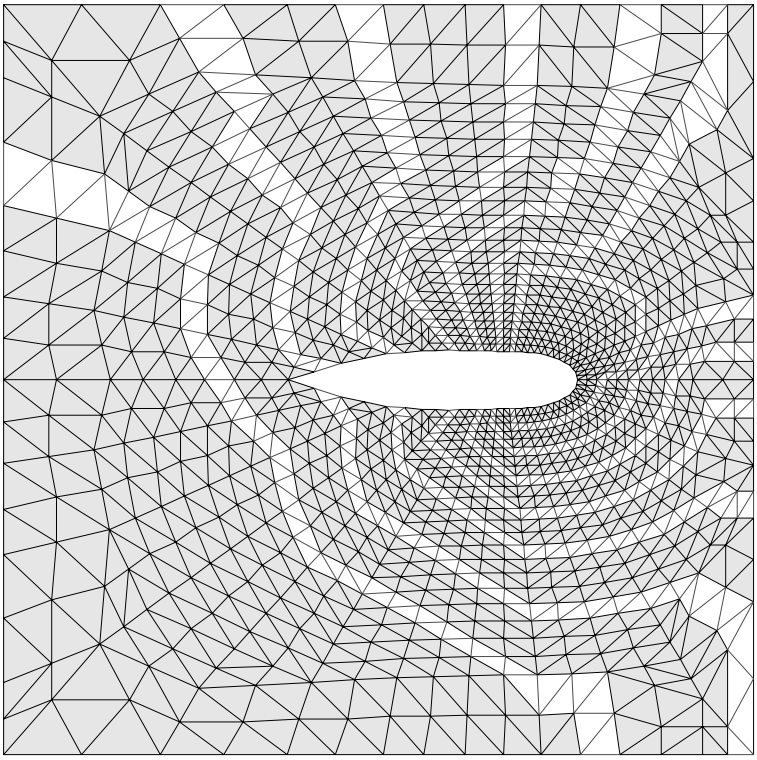

Figure 2: One-way-partitioning

5. *Do a BFS from this node on the subgraph whose nodes are the nodes in dom(idom)*
6. *Call One-way partitioner for this subdomain*
7. Enddo

A critical ingredient in the efficiency of the procedure is the starting node. A well-known procedure for this is a heuristic for finding a so-called pseudo-peripheral node [8]. This procedure starts with an arbitrary node x and performs a BFS from this node. It then records the node y that is farthest away from x and the corresponding distance $dist(x, y)$. We then assign $x := y$ and perform a new traversal from x. We repeat the process and stop when $dist(x, y)$ does not vary between two successive traversals.

4 Level-set expansion algorithms

The level set expansion algorithms are divided up in two phases. In the first phase we must find 'center' nodes for each partition from which to expand each subdomain. We need to find vertices in the initial graph, that are far apart from one another, in a graph theory sense. In the second phase these points are used to generate the subdomains.

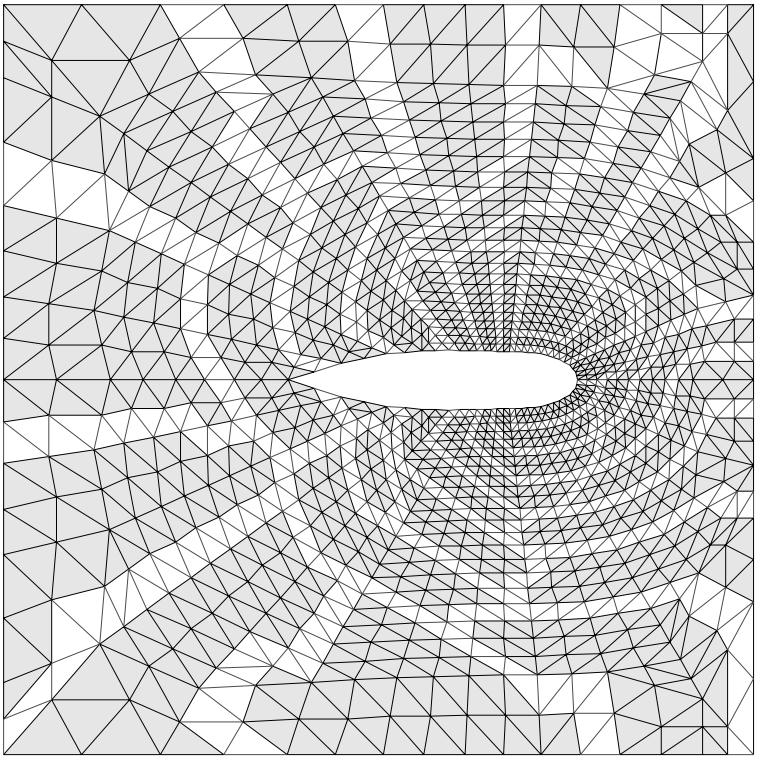

Figure 3: Two-way-partitioning

In this paper we consider a number of techniques based on level-set expansions for generating subgraphs from centers. As was noted in Section 2, we can generalize the Breadth First Search traversal slightly by starting from several independent vertices at once instead of one node only. This will be useful if the initial set of points are spread apart. In addition, the mark that is put on each node that has been visited is now a label and the vertices will inherit the labels of their parents in the traversal process.

We present a version of the algorithm that is sequential with respect to the domains but it is clear that the process is inherently parallel.

ALGORITHM **4.1** *Level-set-expansion*

1. **Start:**
2. *Find an initial set of 'coarse mesh' vertices 3.* $v_1, \ldots, v_{ndom}$
4. *For* $i = 1, 2, \ldots, ndom$ *Do* $label(v_i) := i$.
5. *Define* $levset := \{v_1, \ldots, v_{ndom}\}$ *and* $nodes = ndom$
6. **2. Loop:** *While (nodes<n) Do*
7. $Next\_levset = \phi$
8. *For each* v_j *in levset Do*
9. *for each neighbor* v_k *of* v_j *s.t.* $label(v_k) = 0$ *Do*

10. $Next\_levset := Next\_levset \bigcup \{v_k\}$
11. $label(v_k) := label(v_j)$
12. $nodes = nodes + 1$
13. *EndDo*
14. *EndDo*
15. $levset := Next\_levset$
16. *EndWhile*

The algorithm starts with one node in each processor then expands by adding level-sets until all points are labeled. At the end, all nodes having the same label will constitute a subdomain. We must assume here that the initial graph is connected or that there is at least one starting node in each connected component. The parallel version of this algorithm consists of assigning each starting node to a different processor, then expanding the level sets independently. At some point there will be conflicts, i.e., two processors will attempt to 'acquire' the same node which belongs to two level sets originating from two different starting nodes. In such cases, the host program must arbitrate. Our current implementation uses a first-come first served rule, but there are several possible improvements which are not considered here.

A modification of this algorithm, which will be used later, consists of recording the distances for the centers of the nodes as they are assigned a label.

ALGORITHM **4.2** *Level-set traversal with distance updates*

1. Start:
Find an initial set of vertices $v_1, \ldots, v_{ndom}$
For $i = 1, 2, \ldots, ndom$ *Do*
$label(v_i) := i.$
$dist(v_i) := 0.$
End For
Define $levset := \{v_1, \ldots, v_{ndom}\}$ *and* $nodes = ndom$
2. Loop: *While (nodes<n) Do*
$Next\_levset = \phi$
For each v_j *in levset Do*
for each neighbor v_k *of* v_j *s.t.* $label(v_k) = 0$ *Do*
$Next\_levset := Next\_levset \bigcup \{v_k\}$
$label(v_k) := label(v_j)$
$dist(v_k) := dist(v_j) + 1$
$nodes = nodes + 1$
EndDo
EndDo
$levset := Next\_levset$
EndWhile

5 Finding center points

It was shown in the previous section that once we find a number of nodes that are uniformly spread-out within the whole graph, it is quite easy to partition the graph using these nodes are centers, in a level-set expansion algorithm. The next question which we may ask now is how to find such center points. There are at least three possible options. First, if a coarse mesh is already available from the discretization then the nodes of this mesh can be taken as the centers. If a coarse mesh is not available we can alternatively use points provided from a two-way partitioning algorithm. Second, if the coordinates of the nodes are available then one can easily select the centers from simple geometrical considerations. For example, for a rectangular 2-D mesh, we can choose points that are uniformly distributed in each direction. Other alternatives are required for the cases where only the graph is known and coordinate information is not available.

5.1 Using nodes from a 2-way partitioning

In general, the two-way partitioning algorithm does not provide as good a splitting of the graph as some of the well-known alternatives such as the Recursive Spectral Bisection technique [11]. It is, however, rather inexpensive to obtain. As a result, we can use this partitioning only to get the centers for the Level-Set Expansion. For example we can simply take the middle node in the subdomain as a center. The resulting partitioning will be far better than the original two-way partitioning in general. An illustration of the process is given in Figure 4. This partitioning results from subdividing each subdomain obtained in Figure 4 with one-partitioning, into four subdomains.

5.2 Modified pseudo-extents

Here we modify the breadth-first search algorithm in an attempt to find a set of *ndom* centers as far apart from each other as possible in a graph theoretical sense. One simple way to find a vertex at a maximum distance from a set of vertices is to use the set of vertices as a starting point in a breadth first search and choose the last vertex visited as the furthest distance vertex from the set. This simple idea is exploited in the modified pseudo-extents algorithm. Given a set of candidate centers, each center node is checked to see if it is at a maximum distance from the remaining center nodes. If it is not, it is moved to the maximum distance node. When all centers are at a maximum distance from the other centers, the algorithm has completed.

First we recall a few standard definitions. The distance between two individual vertices as the minimum number of edges in a path connecting the two vertices. If there is no path connecting the two vertices the distance between them is infinite. The distance from a node to itself is defined to be zero. The distance between a vertex and a set of vertices is then the minimum distance between the vertex and any

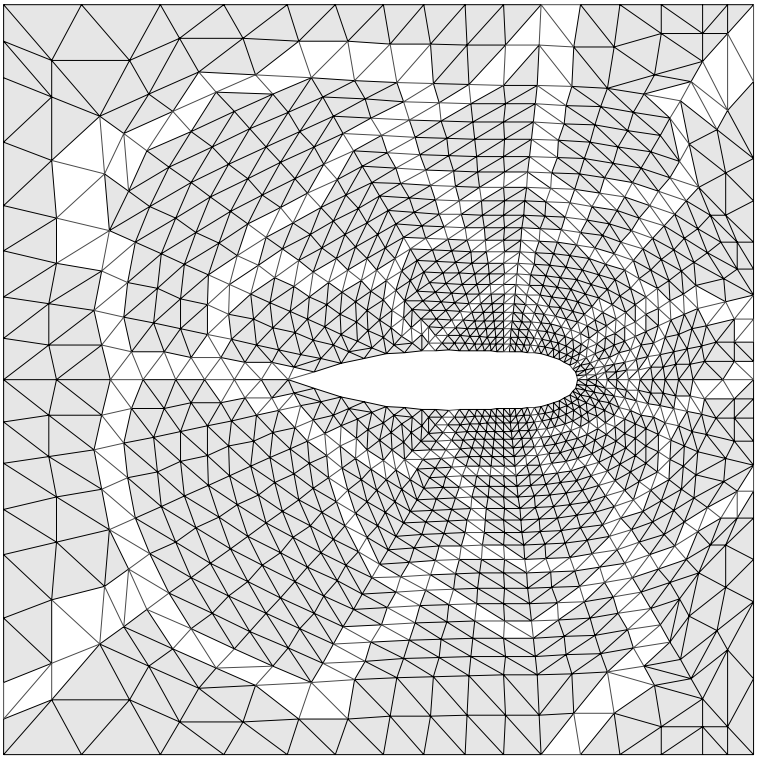

Figure 4: Level-set expansion using nodes from the two-way partition

vertex in the set,

$$dist(v, S) = \min_{s \in S} dist(v, s)$$

and we will denote by $argdist(v, S)$ the set of elements of S that achieve this distance:

$$argdist(v, S) = \{s_j \in S_j \mid d(v, s_j) = dist(v, S)\} \quad (1)$$

Additionally, we define the extent of a subset S of V with respect to V, to be the maximum of the distances between elements of v and S,

$$ext(S)_V = \max_{v \in V} dist(v, S)$$

and similarly to the distance, we define,

$$argext(S)_V = \{v \in V \mid dist(v, S) = ext(S)_V\}$$

Now consider an algorithm for finding a set of $ndom$ centers C, where C is a subset of all the vertices V. To begin, take C to be any set of $ndom$ nodes from V. Take one node $c_i \in C$ and use $\{C - c_i\}$ as an initial set of nodes in a breadth-first search. Now take the last node visited as a candidate center c_i^{new}. If

$$dist(c_i^{new}, \{C - c_i^{old}\}) < dist(c_i^{old}, \{C - c_i^{old}\})$$

then we replace c_i^{old} by c_i^{new}. We continue this way until no center can be replaced with a preferable node.

ALGORITHM **5.1** *Modified Pseudo-Extents*

1. **1. Start:**
2. *Define:* $C = \{v_1, \ldots, v_{ndom}\}$; $not\_done = TRUE$
3. **2. Loop:**
4. *DO WHILE* not_done
5. $not\_done = FALSE$
6. *For each* c_i *in* C *do:*
7. $C := C - \{c_i\}$
8. *For each* $v_j \in V \backslash C$ *compute* $d_j = dist(v_j, C)$,
9. *If* $d_i = ext(C)_V$ *Then*
10. $C = C \cup \{c_i\}$
11. *Else*
12. $C = C \ \cup \ argext(C)_V$
13. $not\_done = TRUE$
14. *End If*
15. *End For*
16. *End While*

In line 12, $argext(C)_V$ may be a set containing more that one element. In this case, we need to take one element that achieves the extent instead of the whole set of elements.

To prove that the algorithm converges, we start by defining the notation

$$dist_*(s, C) = dist(s, C - s)$$

Consider one inner step of the algorithm which starts in line 6 of the algorithm. We observe that each time one instance of this loop is executed, the distance from the new c_i to the rest of C increases or is unchanged. More precisely, we have by definition of the new center c_i,

$$dist(c_i^{new}, C - c_i^{old}) = \max_{v \in V} dist(v, C - c_i^{old}) \geq dist(c_i^{old}, C - c_i^{old}) \qquad (2)$$

However, since the new set C^{new} is defined by

$$C^{new} = (C - c_i^{old}) \cup \{c_i^{new}\}$$

we observe that the left-hand-side of (2) is exactly equal to $dist_*(c_i^{new}, C^{new})$. Therefore, we always have,

$$dist_*(c_i^{new}, C^{new}) \geq dist_*(c_i^{old}, C^{old})$$

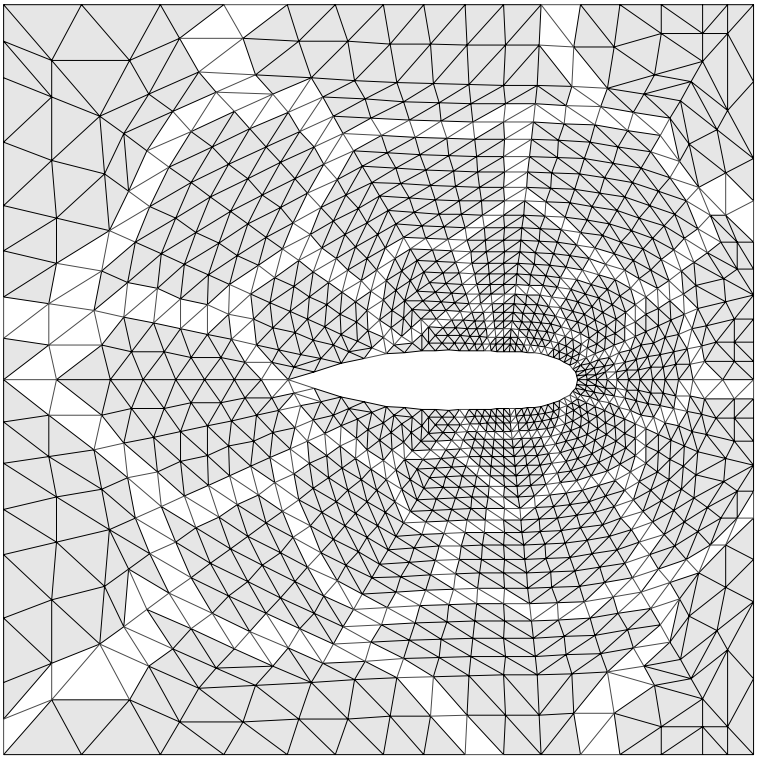

Figure 5: Modified Pseudo Extents Algorithm

after each inner step. Since the sequence of all distances generated by the algorithm are bounded from above and monotonically non-decreasing, they have a limit. In fact, since we have a finite number of possible values, the sequence will become constant after a certain step and the algorithm will stop.

The solution provided by the algorithm is not unique. It depends on the initial set C as well as on the choices we make when encountering an $argext$ set which contains more than one point in line 12 of the algorithm. We should also mention that the algorithm can be regarded as an extension of a standard algorithm used in sparse matrix computations for finding pseudo-peripheral nodes in graphs [8].

5.3 Inverse power algorithm

The inverse power algorithm described below is identical to the modified pseudo-extents algorithm with an alteration in the method of measuring distance between a node and a set of nodes that more fully incorporates the global spread between candidate centers.

This algorithm finds a set of $ndom$ centers C from the set of all vertices V such

that no single center v_i could be replaced by another vertex v_j so that

$$\sum_{\substack{k=1 \\ k \neq j}}^{ndom} (d_{jk})^{-pow} < \sum_{\substack{k=1 \\ k \neq i}}^{ndom} (d_{ik})^{-pow}$$

where d_{ij} is the distance from v_i to v_j.

As mentioned above, this measure allows for a more globally spread out set of centers at a slight increase in cost. The reason for this is that this measure incorporates not just the distance to the nearest other center, but also the distance to all other centers, with the nearer centers contributing a larger share of the total. Proof of convergence is similar that of the modified pseudo extents algorithm.

ALGORITHM **5.2** *Inverse Power*

1. Start:
Define: $C = \{v_1, \ldots, v_{ndom}\}$
For Each c_i *In* C
Let d_j *be the distance from* v_j *to* c_i
For Each v_j *In* $(V - c_i)$
$p_j = p_j + (d_j)^{-pow}$
End For
End For
2. Loop:
$done = TRUE$
For Each c_i *In* C
Let d_j *be the distance from* v_j *to* c_i
$Q = P$
For Each v_i *In* $(V - c_i)$
$q_i = q_i - (d_i)^{-pow}$
End For
If $p_i > \min\{Q - \{q_j | v_j \in C\}\}$ *Then*
$P = Q$
$c_i = \{v_j | q_j = \min\{Q - \{q_k | v_k \in C\}\}$
$done = FALSE$
End If
End For
UNTIL done

As a measure of internode distance, we used the minimum number of edges forming a path connecting the two nodes. Other measures of distance between two nodes are possible. As a significant improvement, we could modify the BFS algorithm to begin with one node and to count all of the edges examined before reaching the second

node instead of just the edges on the path connecting the two nodes. As a necessary condition for convergence, one must only count the edges in levels up to but not including the current level. This definition of internode distance is used in the Edge Inverse Power (EIP) Algorithm.

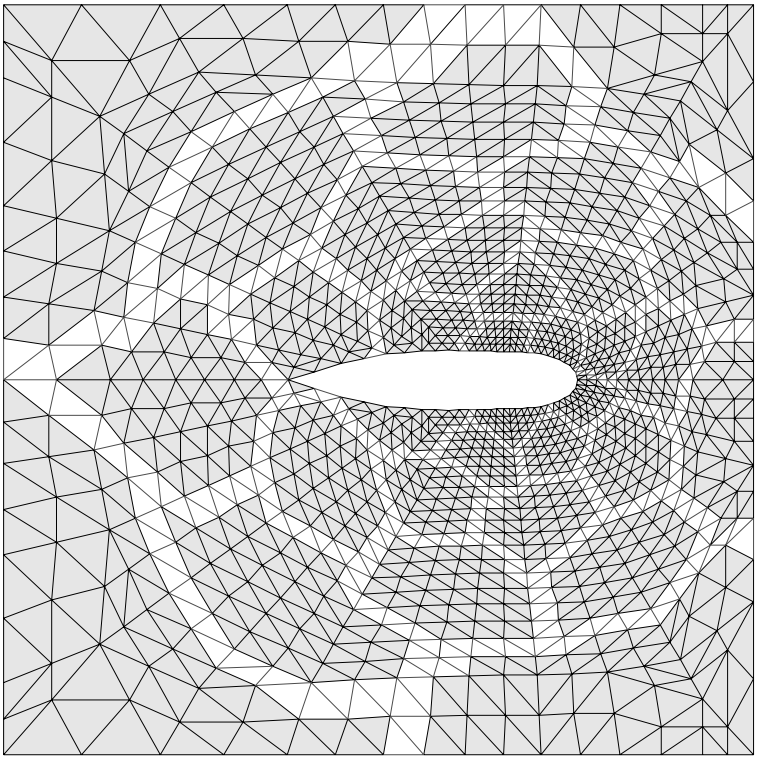

Figure 6: Inverse Edge Power Algorithm

5.4 Recursive formulations

The MPE and EIP algorithms both have run-times of $O(k * e)$ with k being the number of partitions and e being the number of edges and with MPE having a smaller multiplier. These algorithm's run-times may be reduced to $O(\log(k) * e)$ with possible reduction in partition effectiveness using recursive calls. First the graph is partitioned into (approximately) $\sqrt{k}$ partitions, and then partitioning each of these $\sqrt{k}$ partitions into smaller partitions. We will call these algorithms Recursive Modified Pseudo-Extents (RMPE) and Recursive Edge Inverse Power (REIP) respectively.

6 Analysis

As shown earlier, many measures of the quality of a partition exist. Typically, a partition should divide a graph into an approximately equal number of nodes. Nor-

mally, the amount of work required of a single processor node in a parallel algorithm is proportional to the number of nodes at that processor. If the total amount of work is divided evenly among processors, each processor will finish at the same time, eliminating idle processors, and ensuring that the parallel program finishes as soon as possible. Furthermore, since nodes in a graph usually have, more or less, a constant number of edges connected to them, the memory requirements at each node are lessened. A partition should also try to minimize the number of edges connecting separate domains. These edges represent information that needs to be communicated between subdomains, and as such, correspond to overhead in the parallel program that does not exist in the serial formulation. In addition, since we are attempting to balance the load at each processor node, the number of cut edges surrounding each subdomain should be nearly equal. Another consideration that affects the quality of a partition, is the number of subdomains bordering each subdomain. Two subdomains border each other if there exists at least one edge connecting nodes in each subdomain. Bordering subdomains therefore, need to exchange information. Since there is some non-trivial overhead associated with each collection of information sent from one subdomain to another, a partition should attempt to minimize the number of bordering subdomains and, again, the number of neighbors should be balanced among all of the subdomains. Finally, in addition to all of these conflicting considerations, the mapping of the subdomains onto the actual architecture can affect the relative importance of each one to the other.

Given all of these parameters and the fact that reducing one tends to increase another, some objective measure is needed for the quality of a partition. Since the partitions are to be used to improve the speed of execution on a parallel machine, we will use an estimate of a related algorithm on a hypothetical machine. We use the sparse matrix vector product as the sample algorithm, and attempt to define a hypothetical machine representative of current technology.

For sparse matrix-vector product, we assume that the total number of operations required is proportional to the number of edges plus the number of nodes in the graph. For our hypothetical machine we assume that a floating point operation takes time T_c. Sending a message of size m between processor nodes on our machine takes

$$T_s + m * T_w$$

time. We are assuming that communication between any pair of processor nodes takes equal time. This is necessary since the mapping of subdomains to processors is independent of the partition. Also, as mentioned in section 2 earlier, many parallel computers attempt to equalize this interprocessor communication time. For our hypothetical machine we take

$$T_w = 8 * T_c$$

and

$$T_s = 100 * T_c$$

which we feel are representative of current technology.

7 Results and conclusion

Figure 7 shows the estimated run-times scaled by the scalar run-time for a large (66,049 nodes, 394,240 edges) regular, two-dimensional graph. This is also the inverse of the estimated speedup. Although the graph used is regular, none of the algorithms make use of this fact. Figure 7 shows the same run-times scaled to the best algorithm at each point to more clearly show their relative effectiveness.

As can be seen, all of the algorithms are within 10-20 percent of each other over a wide range, with the Edge Inverse Power algorithm giving the lowest estimate for most of the partition sizes.

The main advantages of level-expansion algorithms are their low cost and their simplicity. They are also quite flexible in that they can incorporate arbitrary weights in the graphs in order to take into account the unequal computational demands in different parts of the physical domain, or different demands from the numerical method being used. In these situations, additional costs can just be retrofitted to the various weight functions used for partitioning.

References

[1] Stephen T. Barnard and Horst D. Simon. A fast multilevel implementation of recursive spectral bisection for partitioning unstructured problems. Technical Report RNR-92-033, NASA Ames, Moffett field, CA, 1993.

[2] Marsha J. Berger and Shahid H. Bokhari. A partitioning strategy for nonuniform problems on multiprocessor s. *IEEE Transactions on Computers*, C-36:570–580, 1987.

[3] N. Chrisochoides, Geoffrey Fox, and Joe Thompson. MENUS-PGG mapping environment for numerical unstructured and structured parallel grid generation. In *Proceedings of the seventh international conference on domain decomposition methods in scientific and engineering computing*, 1993.

[4] N. Chrisochoides, C. E. Houstis, E. N. Houstis, P. N. Papachiou, S. K. Kortesis, and J. Rice. DOMAIN DECOMPOSER: a software tool for mapping PDE computations tp parallel architectures. In R. Glowinski et. al., editor, *Domain Decomposition Methods for Partial Differential Equations*, pages 341–357. SIAM publications, 1991.

[5] N. Chrisochoides, E. Houstis, and J. Rice. Mapping algorithms and software environment for data parallel pde iterative solvers. *Journal of Parallel and Distributed Computing.* toappear.

[6] Yeh-Ching Chung and Sanjay Ranka. Mapping finite element graphs on hypercubes. *Journal of Supercomputing*, 6:257–282, 1992.

[7] Charbel Farhat and Michel Lesoinne. Automatic partitioning of unstructured meshes for the parallel solution of problems in computational mechanics. *International Journal for Numerical Methods in Engineering*, 36:745–764, 1993.

[8] J. A. George and J. W. Liu. *Computer Solution of Large Sparse Positive Definite Systems.* Prentice-Hall, Englewood Cliffs, N. J., 1981.

[9] C. A. Leet, B. W. Peyton, and R. F. Sincovec. Toward a parallel recursive spectral bisection mapping tool. Technical report, Oak Ridge National Lab., Knoxville, TN, 1993.

[10] J. W. H. Liu. A graph partitioning algorithm by node separators. *ACM Trans. Math. Software*, 15:198–219, 1989.

[11] A. Pothen, H. D. Simon, and K. P. Liou. Partitioning sparse matrices with eigenvectors of graphs. *SIAM J. Matrix. Anal. Appl.*, 11:430–452, 1990.

[12] Ponnuswamy Sadayappan and Fikret Ercal. Nearest-neighbor mapping of finite element graphs onto processor meshes. *IEEE Transactions on Computers*, C-36:1408–1424, 1987.

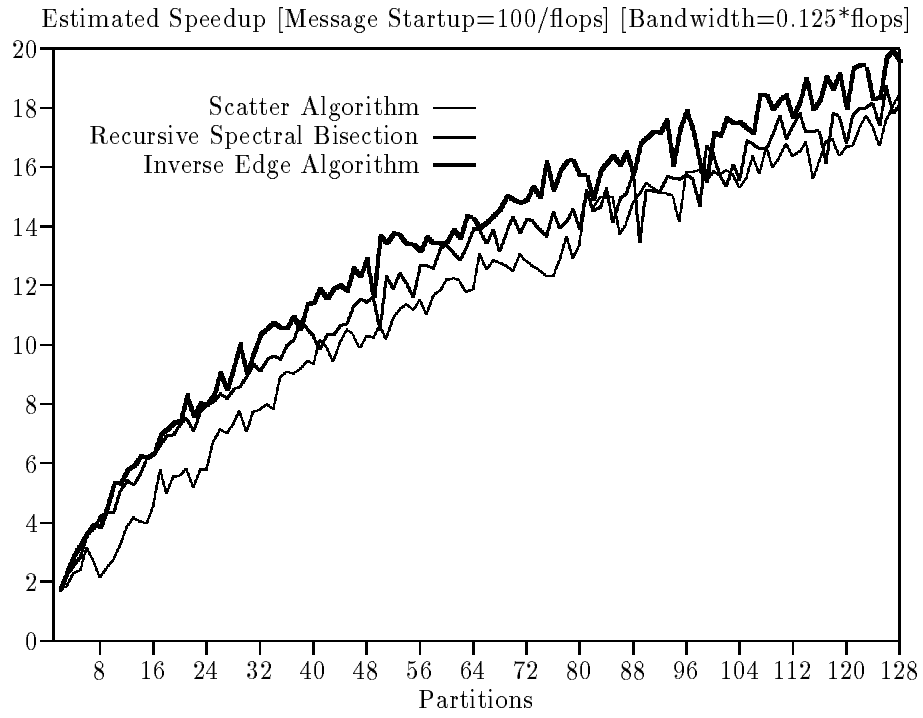

Figure 7: Estimated speed-up

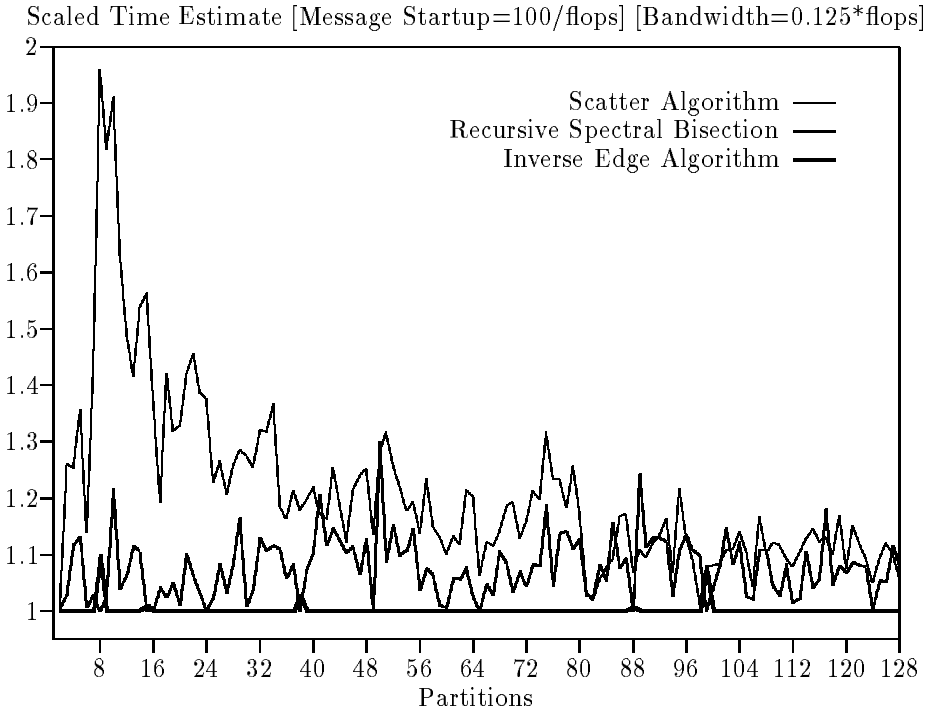

Figure 8: Scaled time estimates