UCLA
COMPUTATIONAL AND APPLIED MATHEMATICS

Recursive Partitioning Methods and Greedy Partitioning Methods: A Comparison on Finite Element Graphs

P. Ciarlet, Jr.

F. Lamour

April 1994

(Revised May 1994)

CAM Report 94-9

Department of Mathematics
University of California, Los Angeles
Los Angeles, CA. 90024-1555

Recursive partitioning methods and greedy partitioning methods: a comparison on finite element graphs.*

P. Ciarlet, Jr † F. Lamour ‡

May 16, 1994

Abstract

The partitioning problem of a graph $G = (V, E)$ consists of dividing the node set V into p subsets in order to minimize the number of intersubset edges in the case of an edge-partitioning or the number of interface nodes in the case of a vertex-partitioning. In either cases the partitioning problem is NP-complete. Consequently, many studies have been devoted to design heuristic methods to approximate the problem for general graphs or even for particular graphs. In this report, we survey different heuristics which address the edge-partitioning problem for finite element graphs. The existing heuristics for these particular graphs can be organized into two main classes: the recursive methods and the greedy methods. We also present a comparison study between various software packages performed on well-known finite element graphs.

1 Introduction

Many problems can be represented by graphs where nodes stand for a distinct amount of work and edges between nodes schematize the information exchanges. The graph partitioning problem is invoked every time one needs to decompose the original problem into smaller subproblems in order to solve them simultaneously or even sequentially, but in both cases to solve them faster than the original larger problem could be solved.

There are different formulations and many variants of the graph partitioning problem, depending on whether one looks at it from the vertex point of view or from the edge point of view. According to the edge point of view, on which we focus in this report, the common definition is formally the one that follows. Let $G = (V, E)$ be a graph where V is the set of nodes and E is the set of edges. The edge-partition of G is a partition of V into two disjoint sets V_1 and V_2 such that:

*this report is submitted to the International Journal of High Speed Computing.

†Department of Mathematics, Univ. of Calif. at Los Angeles, CA 90024 and Commissariat à l'Energie Atomique, CEL-V, D.MA, MCN, 94195 Villeneuve-St-Georges Cedex, France. E-mail: ciarlet@math.ucla.edu, ciarlet@etca.fr. The work of this author was partially supported by the DGA/DRET under contract 93-1192.

‡Department of Mathematics, Univ. of Calif. at Los Angeles, CA 90024. E-mail: lamour@math.ucla.edu. The work of this author was partially supported by the National Science Foundation under contract ASC 92-01266, the Army Research Office under contract DAAL03-91-G-0150, and ONR under contract ONR-N00014-92-J-1890.

- $V = V_1 \bigcup V_2$,
- $\max(\mid V_1 \mid, \mid V_2 \mid) \leq \alpha \mid V \mid$,
- $\mid E_c \mid = \mid \{(v_1, v_2) \in E \text{ and } v_1 \in V_1,\ v_2 \in V_2\} \mid$, called the number of edge cuts or intersubset edges, is as small as possible.

Here, the value of the constant α, $(\frac{1}{2} \leq \alpha < 1)$, characterizes the balancing of the partition. Instead of partitioning the set of nodes into two subsets according to the edge-partitioning problem, one can partition the set of nodes into an arbitrary number of subsets. This is known as the p edge-partitioning problem, where p is the required number of subsets of the partition. One property that has not been mentioned yet concerns the connectivity of the subsets. Besides the requirements of the p edge-partitioning problem it could be necessary to impose that each resulting subset has to be connected.

For all the formulations and the variants of the graph partitioning, the general problem is a NP-hard problem. Many heuristics have been developed while few optimal results have been proved. As it is a difficult problem, researchers have focused on relaxed versions of the problem or on particular graphs of well known structure.

Our concern is to approximate the p edge-partitioning problem for graphs which come from meshes and finite element graphs. For this type of graphs, the heuristic methods proposed in the literature can be roughly arranged in two categories: the greedy methods and the recursive methods.

In this report, we present some representative algorithms for the two types of methods. We also propose an empirical study of these algorithms applied to a catalogue of finite element graphs.

Section 2 recalls some graph definitions and introduces the notations. Section 3 describes the principles and theoretical results of eigenvalue-based methods as well as some resulting algorithms. The next Section presents the greedy methods. Local optimization methods for both partitioning heuristic methods are explained in Section 5. We present the experimental environment, the different parameters which characterize the quality of a given partition, as well as the selected finite element graph examples in Section 6. Section 7 lists the results of an exhaustive comparison between both types of methods. Finally we summarize the main results that ensue from the comparison and briefly outline some directions of our work in this field.

2 Notations and problem definition

First, let us recall some definitions about graphs and fix some notations that will be used through this report. Let $G = (V, E)$ be a graph where V is the set of nodes and E is the set of edges. Let $N = \mid V \mid$ and $M = \mid E \mid$ be the number of nodes and edges respectively. The *degree* of a node v, denoted $d(v)$, is the number of edges that have one endpoint in v, i.e. the number of neighbors of v. Moreover, let $\Gamma(v)$ denote the set of the neighbors of the node v. Finally, we use d to denote the average degree of a node in G.

The Laplacian matrix of a graph G, denoted $L(G) = (l_{ij})_{i,j=1\ldots N}$, is defined by:

$$l_{ij} = \begin{cases} -1 & if\ (v_i, v_j) \in E \\ d(v_i) & if\ i = j \\ 0 & otherwise \end{cases}$$

As we restrict our study to the graphs that come from physical meshes, usually two- or three-dimensional, we shall talk about the *boundary* of the graph.

Among these graphs, there is an important class of graphs which are the *finite element graphs*. There are two ways of looking at finite element graphs. One way is to consider that the node set is the set of vertices of each element and that there exists an edge between two nodes if and only if they belong to the same element ($P1$ or $Q1$ finite elements). The other way is to define the node set by assigning a node to each element and to associate an edge between two nodes if and only if their corresponding elements are neighbors, by sharing an edge in 2 dimensions or a face in 3 dimensions. This last characterization is known as the *dual* graph. In this paper all algorithms are used on the first characterization.

The partitioning problem considered here is the p edge-partitioning problem where p is the number of subsets of the partition. Besides, we want the partition to be balanced, i.e. all subsets must have approximately the same number of nodes. So, formally the problem is stated as follows. Given a graph $G = (V, E)$, let us determine p disjoint subsets V_1, V_2, ..., V_p such that according to the edge-partitioning problem:

- $\bigcup_{i=1}^{p} V_i = V$,
- $\mid V_i \mid \approx \mid V_j \mid$ for $1 \leq i, j \leq p$ and $i \neq j$,
- $\mid \{(v_i, v_j) \in E \; and \; v_i \in V_i, \; v_j \in V_j \; with \; i \neq j\} \mid$ is as small as possible.

The connectivity of the different subsets is not ensured in all algorithms presented in the next sections, but we will specify those which insure this property.

3 Recursive methods

3.1 Spectral bisection

Spectral partitioning methods are based on a particular eigenvalue, and on its associated eigenvector, of the Laplacian matrix of the graph to be partitioned.

In order to understand the intuitive justifications of these methods we must summarize some known properties of the eigenvalues of the Laplacian matrix of a graph and display their links with the graph bisection, or 2 edge-partitioning, problem.

Let $L(G)$ be the Laplacian matrix of G and $N \geq 2$. Let $\lambda_1 \leq \lambda_2 \leq \lambda_3 \leq ... \leq \lambda_N$ be the eigenvalues of $L(G)$.

To begin with, λ_1 is always equal to zero. λ_2, the second smallest eigenvalue, is also known as the algebraic connectivity of G, denoted $a(G)$. Let us recall that $a(G) = 0$ if and only if G is a disconnected graph. By another way, the number of eigenvalues equal to zero gives exactly the number of connected components of the graph. Let us assume, without loss of generality, that λ_2 is the first non zero eigenvalue.

The 2 edge-partitioning problem can be reformulated in the following terms. Let us consider a vector $q \in Q = \{(q_i) \in R^N, \; q_i = \pm 1, \; \sum q_i = 0\}$ and the induced balanced partition for which a node i is assigned to subset V_1 if $q_i = +1$ or to subset V_2 if $q_i = -1$.

According to q the number of intersubset edges (or edge cuts) is equal to

$$\frac{1}{4} \sum_{(i,j) \in E} (q_i - q_j)^2 = \frac{1}{4}(q, L(G)q).$$

So the 2 edge-partitioning problem amounts to finding a vector $q \in Q$ which minimizes this quantity.

By relaxing this discrete minimization problem to a continuous problem, that is to the search of a vector x such that $\| x \|_2^2 = N$ and $\sum x_i = 0$, one finds that the minimum of $(x, L(G)x)$ is obtained for $x = x_2$, where x_2 is the eigenvector associated with λ_2, named the *Fiedler* vector.

Knowing this, the idea is to compute a vector $q \in Q$ by using x_2 in the following way. Let x_l be the median value of the components of x_2. Then q_i is defined as:

$$q_i = \begin{cases} +1 & if\ (x_2)_i > x_l \\ -1 & if\ (x_2)_i < x_l \\ \pm 1 & if\ (x_2)_i = x_l\ (to\ make\ sure\ the\ partition\ is\ balanced) \end{cases}$$

The partition induced by such a vector q is known as the *median cut partition*. Of course this is not an optimal partition. Although we cannot say how close to the optimal the median cut is, there are still some interesting properties about it.

First, there is a noticeable result about the connectivity of a median cut partition which has been stated by Fiedler [7]. Briefly, it also comes from that paper that if there are exactly $\frac{N}{2}$ strictly positive components and $\frac{N}{2}$ strictly negative components then both balanced subsets of the partition are connected. If it is not the case, the connectivity of one of the subsets is however guaranteed.

Secondly, it have been proved by Chan et al. in [2] that for all $p \in Q$, $\| x_2 - p \| \geq \| x_2 - q \|$. This result, which does not allow us to conclude about the optimality of q, can nevertheless reassures the promoters of the median cut method by assuming that it is a close enough choice.

3.2 Recursive spectral algorithms

The median cut partitioning method is the source of many partitioning algorithms. First, it has been used as a step in a divide and conquer process which allows one to partition a given graph into any number of subsets that is a power of two, like in the *Recursive Spectral Bisection* algorithm (*RSB*) due to Simon [16]. There the median cut algorithm is recursively applied to each subgraph induced by the bisection previously computed until the required number of subsets is obtained. Moreover, as the computation of the Fiedler vector is time consuming, many studies have been devoted to speeding up the calculation of this particular vector.

Barnard and Simon have accelerated the RSB algorithm by approximating the Fiedler vector (see [1]). For this, they contract some edges in the graph in order to obtain a smaller graph and repeat this operation a certain number of times until the contracted graph is small enough. Then, they compute the Fiedler vector of the smallest graph. From the smallest graph, an approximation of the Fiedler vector of the previous graph is deduced by an interpolation technique. This approximated vector is then used as a starting point in an iterative method that computes the Fiedler vector. Once the refined Fiedler vector is obtained the process goes back to the larger graph and recomputes a Fiedler vector for this graph using the same strategy (interpolation and refinement), and so on. The algorithm is given below and it will be referred to as "SP 1".

Algorithm SP 1

1. Compute the Fiedler vector for the graph by:
   (a) Constructing a series of smaller graphs $(G^l)_{l=1,\cdots,L}$ obtained by some contraction operations applied to the original graph, (contraction step).
   (b) Computing the Fielder vector for the smallest graph G^L.
   (c) Constructing a series of Fiedler vectors corresponding to the series of graphs by:
      i. interpolating the previously found Fiedler vector to the next larger graph in a way that provides a good approximation to next Fielder vector (interpolation step),
      ii. computing from the given approximated Fielder vector, a more accurate vector (refinement step).
2. Sort vertices according to size of entries in Fielder vector.
3. Assign half of the vertices to each subdomain.
4. Repeat recursively (divide and conquer).

Analysis and complexity of SP 1

The algorithm follows the same stages as the RSB algorithm. However, the computation of the Fiedler vector is decomposed into 3 steps. The first one, called the contraction step, computes a series of smaller graphs deduced from one another by a contraction operation. Here, the contraction technique is based on a *maximal independent set* of the graph. An independent set $V' \subset V$ is a subset of nodes such that for all $v \in V'$, no neighbor of v is in V'. A contracted graph $G' = (V', E')$ is computed by traveling through G, from each node in V', in a breadth search manner (accumulating in a first pass neighbors, then in a second pass neighbors of neighbors, etc.) and by adding an edge to E' each time the paths intersect (redundant edges are avoided). This contraction process is repeated until the size of the resulting graph is reasonably small.

The second step computes the Fiedler vector x_2^L of the smallest graph G^L using an iterative method.

The third step constructs substep by substep the Fiedler vector of the graph G. First, an approximation of the Fiedler vector, $\tilde{x}_2^{L-1}$, of the larger graph G^{L-1} is deduced by expanding the components of x_2^L in the following way. The components corresponding to the same nodes in both graphs are exactly the same. The other components are computed by averaging the components of the neighboring nodes. Note that, since V' is a maximal independent set with respect to V, the components of the neighboring nodes are already defined. Secondly, the Fiedler vector x_2^{L-1} of the graph G^{L-1} is calculated by an iterative method (for example the *Rayleigh Quotient Iteration*, RQI) using $\tilde{x}_2^{L-1}$ as an initial guess. Finally, a series of Fiedler vectors for the graphs G^{L-2}, ..., G are computed by repeating the same process.

The complexity of computing a maximal independent set of a graph G is $O(M)$ (we recall that M is the total number of edges). By constructing a contracted graph based on a maximal independent set, the number of edges of this new graph will be necessarily lower than M. Moreover as we can bound *a priori* the number of contraction steps, the complexity of the whole contraction step can be estimated to be in the order of $O(M)$. The complexity of the determination of the Fiedler vector for G can be estimated to be bounded by a constant number of iterations of the RQI method (times the cost of one

iteration), i.e. M. So for partitioning G into p subsets the complexity of the whole process can be estimated to $O(M \log_2 p)$.

Hendrickson and Leland [10] have modified the RSB algorithm by replacing the bisection steps with either quadrisection or octosection steps, i.e. splitting into four or eight parts during each step. They call the resulting algorithms *Recursive Spectral Quadrisection* (*RSQ*) and *Recursive Spectral Octasection* (*RSO*). Moreover, they have defined a new cut function[1] to be minimized, called the *hop weight*, which takes into account a target architecture, namely a hypercube. Its dimension depends on the algorithm: it is a square for quadrisection (hypercube of dimension 2) and a cube for the octasection. When the partition is done, then each subset is assigned to a processor. Thus, the cost of a communication between two subsets is equal to 1 if they are neighbors (on the hypercube) and more generally l, where l is the length of the shortest path on the hypercube.

In the following, a thorough description of one step of the *RSQ* algorithm is given. The generalization to the octasection is briefly addressed afterwards. However, we consider here only unweighted graphs, whereas Hendrickson and Leland's original paper includes weighted graphs.

Let us define precisely what the hop weight is, according to the quadrisection. If (v, w) belongs to E, then the cost of assigning them to different subsets is either 1 or 2, depending on the length of the shortest path on the square. In order to compute the shortest path automatically, let us write the subset numbers in bits. Here, -1 corresponds to 0 and $+1$ to 1. Subset #0 is $(-1,-1)$, subset #1 is $(-1,+1)$, subset #2 is $(+1,-1)$ and subset #3 is $(+1,+1)$. In short, a subset I is characterized by (q_I^1, q_I^2), $q_I^k \in \{+1,-1\}$. Then the shortest path between subset (q_I^1, q_I^2) and subset (q_J^1, q_J^2) is equal to $\frac{1}{4}\{(q_I^1 - q_J^1)^2 + (q_I^2 - q_J^2)^2\}$. For a partition into four subsets, we can define two vectors q^1 and q^2 belonging to $\{+1,-1\}^N$ such that a node i is assigned to subset I if and only if $q_i^1 = q_I^1$ and $q_i^2 = q_I^2$. Finally, the hop weight to be minimized is equal to

$$\frac{1}{4} \sum_{(i,j)\in E} \{(q_i^1 - q_j^1)^2 + (q_i^2 - q_j^2)^2\} = \frac{1}{4}\{(q^1, L(G)q^1) + (q^2, L(G)q^2)\}.$$

There remains to address the constraints the balancing imposes. As proved in [10], there are three of them:

$$\sum q_i^1 = 0, \ \sum q_i^2 = 0 \text{ and } \sum q_i^1 q_i^2 = 0,$$

which are not equivalent to $(q^1, q^2) \in Q^2$, but to $(q^1, q^2) \in Q^{(2)} = Q^2 \cap \{\sum q_i^1 q_i^2 = 0\}$.

Relaxing the discrete minimization to a continuous problem leads to the following problem: find a pair of vectors (x^1, x^2) minimizing $(x^1, L(G)x^1) + (x^2, L(G)x^2)$ subject to $\|x^k\|_2^2 = N$, $k = 1, 2$, $\sum x_i^1 = 0$, $\sum x_i^2 = 0$ and $(x^1, x^2) = 0$. Clearly, the solutions to this problem correspond exactly to the pairs of orthogonal vectors of norm $\sqrt{N}$ spanning $\{x_2, x_3\}$, where x_2 and x_3 are respectively the second and third eigenvectors of $L(G)$. The next step is to choose one solution (x_S^1, x_S^2) among all the candidate pairs, which minimizes $\sum_k \sum [1 - (x_i^k)^2]^2$.

Then the last step is to find a pair (q^1, q^2) close enough to (x_S^1, x_S^2). The solution provided by Hendrickson and Leland is to find a pair which belongs to $Q^{(2)}$ and minimizes $\sum_k \|x_S^k - q^k\|_2^2$.

The algorithm is given below and it will be referred to as "SP 2".

[1] it is a generalization of the usual edge cut function that is minimized when a bisection algorithm is used.

Algorithm SP 2

1. Compute the second and third eigenvectors (x_2, x_3) of $L(G)$ by:
   (a) Constructing a series of smaller graphs $(G^l)_{l=1,\cdots,L}$ obtained by some contraction operations applied to the original graph, (contraction step).
   (b) Computing (x_2^L, x_3^L) for the smallest graph G^L.
   (c) Constructing a series of pairs of eigenvectors corresponding to the series of graphs by:
      i. interpolating the previously found pair of eigenvectors to the next larger graph in a way that provides a good approximation to next pair (interpolation step),
      ii. computing from the given approximated pair, a more accurate pair of eigenvectors (refinement step).
2. Find a pair of vectors (x_S^1, x_S^2) of norm $\sqrt{N}$ spanning $\{x_2, x_3\}$ which minimizes $\sum_k \sum [1-(x_i^k)^2]^2$.
3. Find a pair (q^1, q^2) of $Q^{(2)}$ which minimizes $\sum_k ||x_S^k - q^k||_2^2$.
4. Assign the vertices to the subsets according to the components of (q^1, q^2).
5. Repeat recursively (divide and conquer).

Analysis and complexity of SP 2

This algorithm is a generalization of the RSB algorithm. Moreover, they compute the eigenvectors x_2 and x_3 by using an idea very similar to that of Barnard and Simon. The only difference is when they contract the graph. Instead of using a contraction based on the maximal independent set, they prefer a *maximal matching* technique. First, they find a maximal set of edges such that no two of them share an endpoint. Then two vertices joined by such an edge are merged. The process is repeated until a small enough graph is obtained. Now, if $x^1 = \cos\theta x_2 + \sin\theta x_3$ and $x^2 = -\sin\theta x_2 + \cos\theta x_3$, then step 2 amounts to solving a quartic equation and sines and cosines of θ. The next step amounts to solving an assignment problem. Assigning vertices to subsets according to the values of (q^1, q^2) is straightforward. Finally, the algorithm is applied recursively to each of the subsets previously constructed.

Before estimating the overall complexity of the algorithm, let us consider only steps 1 through 4. As for SP 1, the cost of computing the eigenvectors x_2 and x_3 with the Lanczos algorithm is proportional to $O(M)$, if we bound *a priori* the number of contraction steps. The cost of step 2 is $O(N)$. Next, the complexity of step 3 is proportional to $O(N \log_2 N)$. And finally, step 4 requires $O(N)$ operations. Therefore, the complexity of steps 1 through 4 is in the order of $\max[O(N \log_2 N), O(M)]$. Now, the cost of these four steps applied to the first four subsets is $\max[\frac{1}{4}O(N \log_2 N), O(M)]$. Then, as $\log_4 p$ recursions are performed, the overall cost is proportional to $\max[O(N \log_2 N), O(M \log_4 p)]$.

Let us discuss briefly the octasection algorithm RSO. It can be derived easily from the RSQ algorithm. First, three vectors (q^k) are now required to construct the assignment. The hop weight is equal to $\frac{1}{4}\sum_{k=1}^{3}(q^k, L(G)q^k)$. The balancing constraints are $\sum q_i^k = 0$, $\sum q_i^k q_i^l = 0$, $k \neq l$ and $\sum q_i^1 q_i^2 q_i^3 = 0$, defining a subset $Q^{(3)}$ of Q^3. Then by relaxing the discreteness, the problem to be solved is now to minimize $\sum_{k=1}^{3}(x^k, L(G)x^k)$ subject to $||x^k||_2^2 = N$, $k = 1, 2, 3$, $\sum x_i^k = 0$ and $(x^k, x^l) = 0$, $k \neq l$. For practical reasons, Hendrickson and Leland chose to remove the cubic constraint. Again, the solutions to this problem are orthogonal triplets (x^1, x^2, x^3) of norm $\sqrt{N}$ spanning $\{x_2, x_3, x_4\}$. To determine

(x_S^1, x_S^2, x_S^3), $\sum_k \sum [1 - (x_i^k)^2]^2$ is minimized over the set of solutions, subject to the cubic constraint. Finally, a triplet (q^1, q^2, q^3) of $Q^{(3)}$ is obtained by minimizing $\sum_k ||x_S^k - q^k||_2^2$.

The structure of RSO is very similar to RSQ and is omitted here.

Concerning the overall complexity, as the cost of computing the first three eigenvectors of $L(G)$ is still $O(M)$, then it remains proportional to $\max[O(N \log_2 N), O(M \log_8 p)]$.

3.3 A multilevel technique

On the other hand, Hendrickson and Leland in [13] used a multilevel technique to partition a graph. They first reduce the size of the graph, and derive a series of smaller graphs by contracting the edges of the original graph until the size of the last graph is small enough. They partition the smallest graph using bisection according to the second eigenvector. Then, they reflect the partition when uncontracting the series of graphs. Moreover, to improve the quality of the partition (in term of balancing and of number of intersubset edges) they perform a local optimization method which exchanges nodes between the subsets of the partition. Finally they repeat the previous sequence of steps on each subset until the total number of subsets is obtained. The algorithm below is called "ML".

Algorithm ML

1. Construct a series of smaller graphs obtained by contraction operations applied to the original graph.
2. Partition the smallest graph based on the second eigenvector.
3. Propagate the partition by:
   (a) Uncontracting the smallest graph.
   (b) Reflecting back the partition to the uncontracted graph.
   (c) Refining locally the partition using a local optimization method.
   (d) Repeating steps (a), (b), (c), until the original graph.
4. Repeat recursively (divide and conquer).

Remark: We can notice that the foundations of this method are in the use of a multilevel technique to speed up the computation of the partition and not in the type of the partitioning algorithm used in the bottom level.

Analysis and complexity of ML

The contraction step (step 1) is done as in SP 2. The contraction step is repeated from the contracted graph until the size of the graph is small enough.

From the partition of the smallest graph, the partition of the larger graph is deduced by assigning to the same subset the corresponding nodes at the origin of a merged node in the smaller graph (step 3.(b)). The induced partition is not necessarily as good as the one of the smaller graph. So step 3.(c) of the algorithm tries to rectify the induced partition by moving some nodes from one set to another. The local optimization method used to perform these moves is described in more details in Section 5. Step 3.(d) simply iterates the reflection of the partition until the original graph is reached.

As for step 1 in SP 2, the complexity of the contraction step can be estimated to be in the order of $O(M)$. The partitioning of the smallest graph (step 2) is expected to be at most $O(M)$. The complexity of step 3 is dominated by the optimization process (step (c)) which is estimated by Hendrickson and Leland [13] to be $O(M)$. Therefore, the overall complexity can be estimated as $O(M \log_2 p)$.

4 Greedy algorithms

4.1 Principles

Greedy algorithms are a natural and naive way to look at some graph problems. According to the p edge-partitioning problem a greedy algorithm can be described as an algorithm that computes each subset V_i by simply accumulating nodes when traveling through the graph. The problematical questions are only: how to start and how to stop?

The way of accumulating nodes in each subset is obvious from the graph structure of the problem. A starting node v_s is chosen and marked. The accretion process is done by selecting and marking the unmarked neighbors of v_s, then the unmarked neighbors of the neighbors of v_s and so on as long as the expected total number of nodes is not reached. This can be viewed as successively building fronts.

The way of choosing a starting node v_s will clearly affect the shape of the final partition. It will also influence the communication scheme, i.e. the number of existing edges between different subsets of the partition.

In the same way, the manner that one chooses the prescribed number of nodes among all the candidate nodes of the last front contributes to the quality of the final partition.

Thus a greedy heuristic for solving the p edge-partitioning problem can be defined roughly by iterating the following 3 steps:

1. Choose a "good" starting node v_s,
2. Accumulate enough descendants of v_s,
3. Stop according to some tie-break strategy in case of multiple choices and mark all the chosen nodes.

At present, there are no theoretical results on the "goodness" of one starting node. Neither are there results on how good a tie-break strategy is. For those two points only intuitive guesses help to design p partitioning problem heuristics. However, an obvious justification of using greedy heuristics for solving the p partitioning problem is that they are inexpensive. We have shown in [3], that for the general case the overall complexity of such algorithm is $O(N \max(p, d, \log_2(\frac{N}{p})))$.

4.2 An algorithm

For finite element graphs, the use of an efficient greedy partitioning heuristic was revived by C. Farhat (see [4]). In his article he addresses the p vertex-partitioning problem: V is partitioned into p subsets V_i and an interface I. The aim is here to obtain well balanced subsets V_i while minimizing $|I|$. For results about a comparison between some vertex-partitioning methods we refer the reader to [5].

Algorithm GP

The next algorithm, presented in detail in [3], implements the principles of a greedy method as well as some other original features. First, this algorithm builds connected subsets. On the other hand, it does not always provide well balanced subsets. In the second place, the subsets are constructed in a concentric way around the boundary of the graph. Finally, for each subset, in case of multiple choices between the nodes of the last front, the tie-break strategy chooses those which are linked to as few unmarked nodes as possible.

More precisely, in the partitioning process, the starting node of each iteration i is chosen in order to belong simultaneously to the *boundary* of G, to be an unmarked neighbor of a node of V_{i-1} and to have a minimal positive *current degree*. Here, the current degree of a node is the number of nodes connected to it which have not yet been selected, i.e. marked, during the accumulation step. The current degree of a node decreases to zero as the algorithm goes along.

As for the tie-break strategy, it is achieved by keeping the nodes which have a minimal current degree. In fact, the algorithm does not simply build the subsets as mentioned but also check the connectivity of each of them. Whenever a subset is found to be multiconnected, the algorithm corrects the feature by reassigning small components to other subsets and by keeping the largest component.

The algorithm "GP" is described next.

1. If $i < p$
   (a) Compute $n_i = \frac{N - \sum_{j=1}^{i-1} n_j}{p-(i-1)}$.
   (b) Choose an unmarked node v_s such that:
      i. v_s belongs to the current boundary,
      ii. if the current boundary is not new, v_s is a neighbor of a node belonging to V_{i-1} (if possible[2]),
      iii. v_s has a minimal current degree.

      Mark v_s and initialise V_i with v_s.
   (c) If there are unmarked neighbors of nodes of V_i, let k be their number.
      i. If $\mid V_i \mid + k < n_i$ then mark those nodes, add them to V_i and update the current degree of their neighbors, and then return to 1.(c).
      ii. Mark $(n_i - \mid V_i \mid)$ minimal current degree nodes and add them to V_i.
      Update the current and virtual boundaries.
      Do $i = i + 1$ and return to 1.
   (d) If there are no more unmarked neighbors of nodes of V_i and if $\mid V_i \mid < n_i$ then unmark the nodes in V_i and assign them to neighboring subsets. Return to 1.
2. Mark all the remaining nodes and add them into V_p. If V_p is multiconnected then keep the largest component and unmark the nodes of the other components and assign these nodes to neighboring subsets.

[2]It may not be possible to find a node neighboring the previously built subset if the boundary is multiconnected.

Analysis and complexity of GP

The two conditions (b) i, ii required for the starting node force it to belong either to the boundary of the graph or to its *current boundary*. The current boundary is defined, at each iteration, as the set of nodes that belong to the boundary of the graph and that have not yet been marked. The definition of the current boundary can be extended even when there remains no unmarked node. In that case, the new current boundary is the set of unmarked nodes that are the neighbors of marked nodes. This new current boundary is computed by the intermediate of what we call a *virtual boundary*, which is simply the set of nodes that are neighbors of marked nodes excluding current boundary nodes. Hence, the subsets have a tendency to be more easily connected. Anyway the subsets are compelled to be connected owing to step 1(d) and 2 for the last subset. Step 1(a) is a consequence of the reassignment steps which requires us to update the number of prescribed nodes by subset (n_i). Lastly, step 1(b)iii and 1(c)ii try to minimize the number of intersubset edges. The complexity of GP is $O(M)$.

5 Local optimization methods

Whatever partitioning methods may be used, one can couple them with a local optimization method in order to improve the load balancing or the intersubset structure. One of the most famous optimization methods for improving a given partition is due to Kernighan and Lin [15].

Fundamentally the method is based on a given 2 edge-partition (V_1, V_2) of the node set V of a graph and tries to improve it by exchanging a subset of V_1 with one of V_2. The selection criterion of the subsets is determined from a gain function which is defined as follows. First, for $v \in V_1$, $g_v = d_{V_2}(v) - d_{V_1}(v)$ ($d_A(v)$ is the number of neighbors of v which belong to A) and for $w \in V_2$, $g_w = d_{V_1}(w) - d_{V_2}(w)$. Then the gain obtained by exchanging a node $v \in V_1$ with a node $w \in V_2$ is equal to:

$$g_{v,w} = g_v + g_w - 2\delta(v, w)$$

where $\delta(v, w)$ is defined by:

$$\delta(v, w) = \begin{cases} 1 & if\ (v, w) \in E \\ 0 & otherwise \end{cases} .$$

So, after computing for each node $v \in V_1$ and $w \in V_2$ the values of g_v and g_w, the algorithm chooses a pair of nodes (v_1, w_1) which maximizes the gain g_{v_1,w_1}. Then, for all neighbors u of v_1 or w_1 the values of g_u are updated. The optimizing process is iterated until $g_{v_{n-1},w_{n-1}}$ is computed. Finally, the algorithm chooses among all pairs of subsets $\{v_1, v_2, ..., v_k\}$, $\{w_1, w_2, ..., w_k\}$ for $k \in \{1, ..., n-1\}$ the one which maximizes the sum of the gains.

This process can be repeated several times until no better improvement, regarding to the number of intersubset edges, is obtained. One iteration of the algorithm, called "KL", is described next.

Algorithm KL

1. Compute g_v, g_w for each $v \in V_1$ and $w \in V_2$.
2. $Q_{V_1} = \emptyset$, $Q_{V_2} = \emptyset$.

3. For $i = 1$ *to* $n-1$ do
   (a) Choose $v_i \in V_1 - Q_{V_1}$ and $w_i \in V_2 - Q_{V_2}$ such that g_{v_i,w_i} is maximal over all choices of v_i and w_i.
   (b) Set $Q_{V_1} = Q_{V_1} \bigcup \{v_i\}$, $Q_{V_2} = Q_{V_2} \bigcup \{w_i\}$.
   (c) For each v in $V_1 - Q_{V_1}$ and v in $\Gamma(v_i) \bigcup \Gamma(w_i)$ do $g_v = g_v + 2\delta(v, v_i) - 2\delta(v, w_i)$.
   (d) For each w in $V_2 - Q_{V_2}$ and w in $\Gamma(v_i) \bigcup \Gamma(w_i)$ do $g_w = g_w + 2\delta(w, w_i) - 2\delta(w, v_i)$.
4. Choose $k \in \{1, ..., n-1\}$ to maximize $\sum_{i=1}^{k} g_{v_i,w_i}$.
5. Interchange the subsets $\{v_1, v_2, ..., v_k\}$ and $\{w_1, w_2, ..., w_k\}$.

Analysis and complexity of KL

Step 1 calculates the corresponding gain of each node. The sets of candidates for moving from one subset to another are first initialized to empty set in step 2. Step 3 (a) chooses a good pair of nodes (v_i, w_i) whose gain g_{v_i,w_i} is maximal. Steps 3.(b), (c), (d) respectively update the sets of candidates Q_{V_1} and Q_{V_2}, then the gains of the nodes which have not been already elected as candidates for moving from V_1 to V_2 and from V_2 to V_1. The subsets which maximize the sum of the gains are determined in step 4 and the interchange is proceeded in the last step. The complexity of the algorithm is dominated by the complexity of step 3 (a) which can be in the order of $O(N^2 \log N)$.

The effectiveness of KL is, of course, dependent on a good starting partition. It has also been displayed that the effectiveness of KL is strongly related to the structure of the graph. Jones in [14] has listed the conclusions of several authors which have experimented with the KL algorithm. It turns out that KL gives better results on graphs which have a high average degree. Thereby, the KL algorithm is usually run conjointly with another heuristic called *compaction* which aggregates the nodes of a given graph in order to increase the average degree of the compacted graph. It has been shown that using both heuristics together greatly improves the resulting partition.

5.1 KL-like methods

The KL algorithm has inspired many local optimization methods. One of them differs by a slight modification in the way of choosing good candidates for the subset exchange and is due to Fiduccia and Mattheyses [6]. Another one generalizes the algorithm described in [6] in the case of a partition into an arbitrary number of subsets (see [10], [12]).

Fiduccia and Mattheyses's method

Fiduccia and Mattheyses in [6] reduce the complexity of the KL algorithm by moving one node after another instead of directly exchanging a pair of nodes. So, after computing the gains g_v and g_w for each $v \in V_1$ and $w \in V_2$ and initializing Q_{V_1} and Q_{V_2} where Q_{V_1} (resp. Q_{V_2}) is the set of nodes of V_1 (resp. V_1) that have been selected to move to V_2 (resp. V_2), (step 1 and step 2), they first choose a node $v_i \in V_1 - Q_{V_1}$ which has the maximum gain value, then they update the gain values of all the neighbors of v_i. In the next step, they perform the same operations with a node from V_2. These two

steps replace the step 3 of the KL algorithm while steps 4 and 5 are identical. The overall complexity of the FM algorithm is thus reduced to $O(M)$, by using a special sorting technique which accelerates the selection step of a node of highest gain.

Hendrickson and Leland's method

Hendrickson and Leland, [12] and [13], generalize the previous method to an arbitrary number of subsets. Instead of a single gain function there are $(p-1)$ gain functions associated with each node. Each of these functions computes the gain obtained by moving the given node to a specific subset. Moreover their method integrates others features. First, their gain function takes into consideration an intersubset cost metric which can be either the classical cut function or the hop weight function. Second as their starting partition is not necessary balanced, they give the precedence to the moves from large subsets to small ones. The overall complexity is estimated to $O((p-1)M)$ using the same sorting technique than the one used in [6].

5.2 A retrofitting method

This method, called CL, has been suggested by the experiments conducted on GP. It consists of three steps. Steps 1 and 3 are identical and are reshaping steps while step 2 is a balancing step.

The reshaping step tries to redesign the outlines of subsets by deleting the excrescences. Generally these excrescences occur during the accumulation step of the partitioning process when some of the chosen nodes encounter prematurely another subset. Then some nodes are attached to the subset to which they belong by a single edge. By reassigning those nodes to neighboring subsets and by iterating the process until no more excrescences remain, the shape of the subsets is improved. Note that these boundary nodes are transfered to the subset which holds the highest number of its neighbors. It is not always possible to eliminate all the excrescences, so the reshaping step has to stop when the overall shape of the partition, i.e. the number of excrescences, does not seem to be improved over the iterations (more precisely five iterations). Here, one iteration consists of spanning the whole set of nodes V and reassigning the excrescences.

Because of the reassignment of nodes to neighboring subsets in order to preserve the connectivity of the subsets, the resulting partition provided by GP is not balanced in most of the cases. Thus the second step of the retrofitting method looks for rebalancing the subsets by moving nodes from large subsets to small ones. One iteration of this consists of three parts. First, the largest and smallest subsets are determined. Then a node of the largest subset is reassigned to its smallest neighbor. Last, a node is reassigned to the smallest subset, coming from its largest neighbor.

If we call V_i the subset from which a node v is taken and V_j the subset to which it is reassigned, then v is chosen among the nodes of V_i bordering V_j. It is the one which leads to the best partition in terms of edge cuts. In other words, it is a node v of V_i such that $d_{V_j}(v) \neq 0$ which maximizes $\{d_{V_j}(v) - d_{V_i}(v)\}$. Moreover, before moving a node v from subset V_i one has to make sure that $V_i - \{v\}$ remains connected. To verify it , we choose a neighbor of v in V_i and then construct fronts (in V_i) from this starting node. If all the nodes of V_i can be reached, then $V_i - \{v\}$ is connected. A way of accelerating this verification is to use the following sufficient condition: if we call $\Gamma_{V_i}(v)$ the neighbors of v belonging to V_i, then $V_i - \{v\}$ is connected if all the nodes of $\Gamma_{V_i}(v)$ can be reached. If $V_i - \{v\}$ does not remain connected then another boundary node of V_i is chosen. If no node of V_i bordering V_j satisfies this connexity requirement, then another subset has to be chosen.

The process is iterated until the standard deviation (i.e. $\sigma = \sqrt{\frac{1}{p}\sum_{i=1}^{p}(n_i - n)^2}$) approches the optimal standard deviation σ_{opt} which is equal to $\sqrt{\frac{1}{p}[((q+1)p - N)(q-x)^2 + (N - qp)((q+1)-x)^2]}$ where $x = \frac{N}{p}$ and $q = \lfloor \frac{N}{p} \rfloor$[3], or does not decrease anymore (after five iterations).

The complexity of one iteration of the reshaping step is $O(M)$. The complexity of one iteration of the balancing step is $O(p)$ for the first part and $O(nd)$ for the two other parts. The cost of checking the connectivity is also $O(nd)$.

Note that when the retrofitting method is performed, neither the number of iterations for the reshaping steps nor for the balancing step are bounded *a priori*.

6 The experimental environment

All the partitioning methods described in the previous sections are tested through a common tool: the *Portable Extensible Tools for Scientific Computation* (PETSc) developed by Gropp and Smith [8], [9]. PETSc is a software library for parallel and serial scientific computations. It provides a variety of packages that go from interfacing message passing systems to the definition of data structures and code for the manipulation of large sparse matrices, as well as data-structure-neutral linear and non-linear solvers. In addition to these existing packages, PETSc makes it easy to include any software written either in C or Fortran. Because of this flexibility, we have used PETSc as a common interface between the partitioning softwares implementing the different heuristics.

6.1 The partitioning softwares

The softwares described in the following have been given by their authors and have been used in PETCs as is. We have written interfaces for transfering inputs and parameters as well as for interpreting the results. The SP 1 code, written by Barnard and Simon, is a software and can be obtained by a request to their authors[4]. The SP 2 and ML code is copyrighted but can be obtained via a request to Hendrickson or Leland[5]. The GP code has been written by the authors of this report.

6.1.1 The SP 1 code

The corresponding code for the SP 1 algorithm represents 4,500 Fortran lines. Actually, the user's inputs are simply the graph which must be in a sparspak format and the number of subsets which is no longer required to be a power of two. Another input that can be specified by the user is the size (number of nodes) of the smallest graph of the contraction step. The value of this number must be a compromise between the swiftness of the execution of the partitioning process and its quality. For our experiments we have fixed this parameter to 200. The SP 1 algorithm can be coupled with a local optimization method. Here the KL algorithm can be invoked, in order to improve the partition, between the two sets obtained after each bisection. The authors choose to perform this optimization step only when the size of the concerned subsets does not make this step too expensive in regards to the whole

[3] $\lfloor . \rfloor$ stands for the lower integer part of a number.

[4] send e-mail to simon@nas.nasa.gov or to barnard@nas.nasa.gov.

[5] send e-mail to bahendr@cs.sandia.gov or rwlelan@cs.sandia.gov.

process. So, the maximal number of nodes of the subsets on which the KL algorithm is performed is a parameter of the method. For our experiments we choose to fix its value to 300.

6.1.2 The SP 2 and ML code

The SP 2 and ML algorithms have been both coded in C and are put together in a tool called *Chaco (version 1.0)* which represents about 15,000 lines (see [11]). In addition to these two algorithms Chaco offers some other partitioning methods like the *Random* algorithm or the *Inertial* algorithm which are not addressed in this paper. Some comparison results on recursive spectral methods and the inertial algorithm can be found in [10], [12] and [13]. The number of input parameters is relatively important and the flexibility that they provide is to the detriment of the facility of the interfacing. Among the inevitable parameters, the number of subsets is required to be, necessarily, a power of two. Moreover, the authors have chosen not to perform automatically the optimization step in either SP 2 and ML but to invoke it from time to time during the uncontraction process. So the parameters inherent in these multilevel methods are three in number. They correspond respectively to the maximal number of nodes of the smallest graph, the respective frequencies of using the optimization technique in SP 2 or in ML. For running the examples described latter we have fixed these parameters respectively to 200, 2, 3. For SP 2, this means that for every other uncoarsening step 1(c), only the interpolation (i) is carried out. The optimization technique used in ML (step 3(c)) is the HL algorithm.

The HL algorithm can also be performed in SP 2, following the user's desire, as an optimization step after each quadrisection. As it greatly improves the behavior of SP 2, it has been automatically included when running this algorithm. This optimization is subject to various parameters, but we will not discuss them here in order to remain concise. Nevertheless, we have set these parameters to the default values advocated by the authors, except for the HL metric parameter. The HL metric parameter defines the intersubset cost metric which guides the moves of nodes from one set to another. The value of the HL metric parameter can be either 1 if the classical cut function is used or 2 if the hop weight function is required. For our experiments the value of this parameter is 1 although the quadrisection is designed to be more efficient in minimizing the number of hops than the number of cuts.

6.1.3 The GP code

The code of the greedy algorithm is written in Fortran and represents about 1,800 lines. Moreover, all computations are performed in integers and therefore all computations terminate (no round-off errors), which certainly helps to guarantee the robustness of the algorithm. Like for SP 1 code, few inputs and parameters need to be specified. In addition to the graph and the number of subsets (which can be arbitrary), the user has the choice to run the code with or without the optimization step which invokes the retrofitting algorithm CL described in paragraph 5.2.

6.2 Comparison parameters

We perform the comparison study according to several paramaters. The more obvious ones are, of course, those that come directly from the problem definition as the average number of nodes by subset or the percentage of intersubset edges. In the same way, the time parameter reflects directly the complexity of the different algorithms. However, we have chosen to add some other parameters in order to emphasize the intrinsic qualities (or weaknesses) of each partitioning heuristic. As the greedy

method does not necessarily insure the balancing of the subsets, we introduce the *average standard deviation* parameter of the subsets which gives a realistic idea of the difference of the load between large subsets and small ones according to the average size of the subsets: σ/n (%). Moreover as some algorithms do not provide connected subsets when other do, we define a connectivity parameter which determines the number of multiconnected subsets.

The notations that summarize the different parameters attached to a given partition are listed below.

p is the number of subsets,

n is the average number of nodes by subset,

σ/n (%) is the average standard deviation of the number of nodes by subset,

m/M is the percentage of intersubset edges,

d is the average degree of the graph,

n_{cc} is the number of multiconnected subsets,

t is the elapsed time in second.

6.3 Graph tests

We have chosen to run the codes on six representative types of finite element graphs: Annulus, a mesh between two concentric circles build by B. Smith (Figure 1), Airfoil, a mesh around an airfoil provided with the latest version of the Matlab package (Figure 2), Eppstein built by D. Eppstein (Figure 3), Cube and Square, respectively a cubic and a square mesh built by the authors, Spheres a mesh between two concentric spheres designed using Modulef by L. Crouzet. Half of them are meshes made of triangles or tetrahedra for a P1 finite element discretization (Airfoil, Eppstein, Spheres) and the other half are meshes made of rectangles or parallelepipeds for a Q1 finite element discretization (Annulus, Cube and Square). Therewith, four of these graphs are two-dimension meshes (Annulus, Airfoil, Eppstein, Square) and the other two are three-dimension meshes (Cube and Spheres). A third category can be drawn according to whether a mesh is regular or irregular. Annulus and Spheres are irregular graphs because they have one hole (the boundary is multiconnected) while Eppstein is irregular because it is a non uniform mesh. Airfoil is irregular because it has three holes and the mesh is non uniform. Cube and Square are regular meshes.

In order to make comparisons on large graphs, we have run the heuristics on *refined* versions of some of these graphs. Here, one level of refinement is carried out (in PETSc) by dividing 2D elements into four parts and 3D elements into eight parts. Clearly, several levels of refinement can be performed to obtain larger and larger graphs.

7 Results and comments

The different codes have been run with the values of parameters described in the corresponding sections of each code in paragraph 6. Let us recall how the different heuristics are coupled with an optimization method. SP 1 invokes automatically KL when the size of the subsets is less than 300 nodes. SP 2 is always coupled with the HL optimization step, which is then automatically invoked

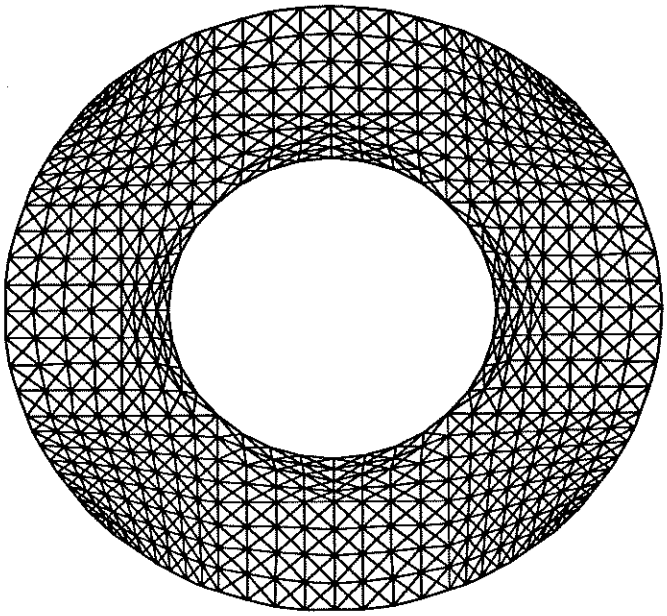

Figure 1: Unrefined Annulus

after each quadrisection. The ML algorithm is run, by definition, jointly with HL. As for the GP algorithm, it is always coupled with the retrofitting method CL. We compare all four methods on a "Viking" SuperSparc processor and the results are summarized for the above-mentioned (possibly refined) graphs in 24 tables and 3 figures. In the Appendix, we provide examples of the resulting partitions on the Eppstein grid.

Annulus (refined twice)

$N = 8448$

$M = 33024$

$d = 7.82$

Tables 1.1-1.4 show the characteristics of the partition into 4, 16, 64 and 256 subsets of Annulus (an unrefined Annulus is presented Figure 1) for all four methods. GP shows a slight unbalance while the recursive methods provide, by definition, well balanced subsets. The number of intersubset edges is lower for SP 2, but all three recursive heuristics give close percentage of intersubset edges. GP gives the worst result in terms of intersubset edges but has the smallest execution time.

p	SP 1	SP 2	ML	GP
4	0	0	0	0.3
16	0	0	0	0
64	0	0	0	4.4
256	0	0	0	2.9

Table 1.1: σ/n (%).

p	SP 1	SP 2	ML	GP
4	1.54	1.17	1.17	2.63
16	4.82	4.61	4.65	5.83
64	11.29	10.87	11.38	11.72
256	24.12	23.47	24.51	24.75

Table 1.2: m/M (%).

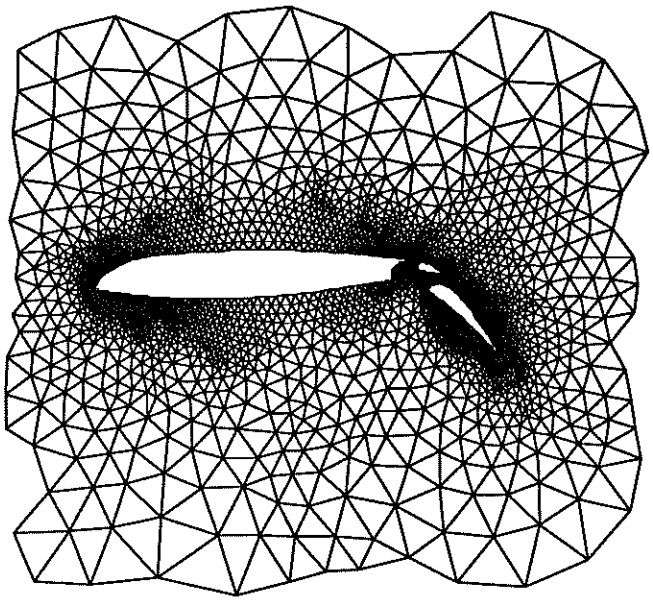

Figure 2: Unrefined Airfoil

p	SP 1	SP 2	ML	GP
4	0	2	0	0
16	0	0	0	0
64	0	0	0	0
256	0	0	0	0

Table 1.3: n_{cc}.

p	SP 1	SP 2	ML	GP
4	2.8	11	2.3	0.5
16	5.1	23	6	0.3
64	5.6	30	11	0.7
256	10	38	19	0.5

Table 1.4: t (s).

For a partition into 64 or 256 subsets the margin between different number of intersubset edges is less important than for the 4-partition. The partitioning time, which is not very dependent on the number of partitions for the GP code, increases for recursive codes. Finally, let us notice that the SP 2 method gives two multiconnected subsets for a partition in 4 subsets.

Airfoil (refined three times)

$N = 258990$

$M = 773168$

$d = 5.97$

Figure 2 shows the unrefined Airfoil. Tables 2.1-2.4 illustrate respectively the characteristics of the partitions into 4, 16, 64, 256 and 1024 subsets of a refined Airfoil. The larger the number of subsets is, the smaller the margin between the values of σ/n and the values of m/M is for all four codes. Nevertheless, ML provides lower bounds in terms of number of intersubset edges for partitions into 4, 16 and 64 while SP 2 gives best results for 256 and 1024 subsets. For the partitioning time the results are very in favor of GP when the number of subsets is greater than 64. We notice also that SP 2 always produce multiconnected subsets and that ML produces multiconnected subsets for large number of subsets.

p	SP 1	SP 2	ML	GP
4	0	0	0	18.5
16	0	0	0	0
64	0	0	0	6.7
256	0	0	0	4.3
1024	0	0	0	3.0

Table 2.1: σ/n (%).

p	SP 1	SP 2	ML	GP
4	0.25	0.41	0.17	0.40
16	0.66	0.77	0.59	1.00
64	1.76	1.87	1.66	1.88
256	3.94	3.90	3.93	4.08
1024	8.29	8.21	8.46	8.35

Table 2.2: m/M (%).

p	SP 1	SP 2	ML	GP
4	0	4	0	0
16	0	11	0	0
64	0	18	0	0
256	0	17	6	0
1024	0	30	22	0

Table 2.3: n_{cc}.

p	SP 1	SP 2	ML	GP
4	48	417	69	58
16	107	810	125	300
64	178	1126	178	46
256	251	1821	329	18
1024	375	1801	751	16

Table 2.4: t (s).

Square

$N = 4096$

$M = 16002$

$d = 7.81$

GP is the incontestable winner for the next example (Tables 3.1-3.4). Actually, even if SP 2 and ML reach the same value of m/M as GP for a 4-partition and if SP 2 reaches also the same value of m/M as GP for a 16-partition or 256-partition, the latter is much faster.

p	SP 1	SP 2	ML	GP
4	0	0	0	0
16	0	0	0	0
64	0	0	0	0
256	0	0	0	0

Table 3.1: σ/n (%).

p	SP 1	SP 2	ML	GP
4	2.82	2.36	2.36	2.36
16	7.46	7.01	7.06	7.01
64	17.15	16.32	16.14	16.01
256	34.79	32.81	33.00	32.81

Table 3.2: m/M (%).

p	SP 1	SP 2	ML	GP
4	0	0	0	0
16	0	0	0	0
64	0	2	0	0
256	0	0	0	0

Table 3.3: n_{cc}.

p	SP 1	SP 2	ML	GP
4	1.1	4.2	1.5	0.1
16	2.1	9.6	3.6	0.1
64	2.8	10.6	5.2	0.1
256	4.8	16	8.8	0.1

Table 3.4: t (s).

Spheres

$N = 9020$

$M = 59418$

$d = 13.17$

p	SP 1	SP 2	ML	GP
4	0	0	0	0
16	0	0	0	0
64	0.2	0.2	0.2	0.8
256	1.2	1.2	1.2	1.8

Table 4.1: σ/n (%).

p	SP 1	SP 2	ML	GP
4	7.21	6.13	6.05	8.03
16	15.83	14.45	14.39	18.30
64	26.53	25.42	25.74	29.81
256	43.08	41.83	42.38	47.51

Table 4.2: m/M (%).

p	SP 1	SP 2	ML	GP
4	0	0	0	0
16	0	0	0	0
64	0	0	0	0
256	0	0	1	0

Table 4.3: n_{cc}.

p	SP 1	SP 2	ML	GP
4	2.5	24	4.1	0.3
16	17	39	8.0	0.3
64	25	54	13	0.8
256	30	56	20	0.4

Table 4.4: t (s).

For Spheres partitioned into 4, 16, 64 or 256 subsets (Tables 4.1-4.4) the best result in terms of intersubset edges is achieved by ML for the first 2 partitions and by SP 2 for the remainder. Both of them even makes a gap with SP 1. One of the reasons why SP 2 and ML really perform well in that case is because of the high degree of the graph. Actually, as it has been pointed out by C. Jones ([14]), the performance of the KL heuristic (which is the HL heuristic in that case and which is automatically included after each quadrisection in SP 2 and between the uncoarsening steps in ML) is improved when it is performed on graphs with high degree. SP 1 does not take advantage of this interesting feature in KL because the method is only invoked when the size of the subsets is small, that is almost at the end of the recursive process. GP is very fast because disconnected subsets occur less often when the average degree increases. Therefore, subsets are better balanced and the retrofitting step CL is not very costly.

Cube

$N = 15625$

$M = 186696$

$d = 23.90$

For this example (Tables 5.1-5.4), the results are similar to the ones obtained on Spheres. Note that, for the partitions into 16, 64 and 256, SP 1 produces multiconnected subsets. Here, GP gives relatively well balanced subsets because of the high average degree of the nodes.

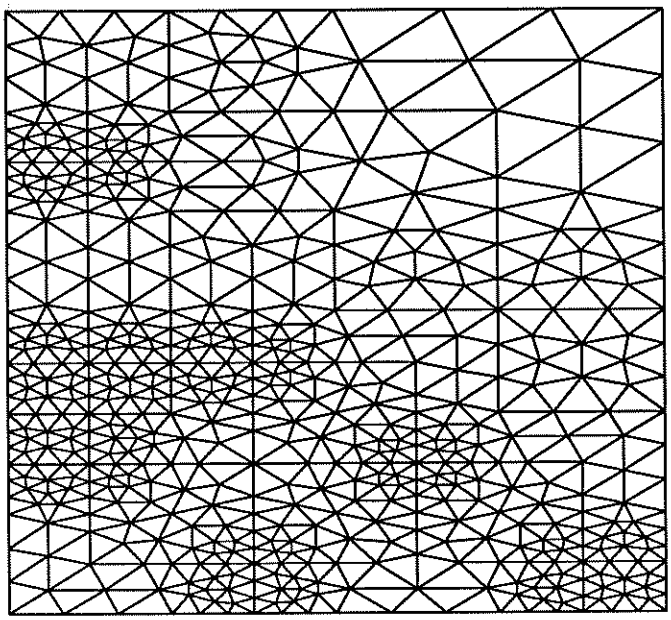

Figure 3: Unrefined Eppstein

p	SP 1	SP 2	ML	GP
4	0	0	0	0
16	0.1	0.1	0.1	0.1
64	0.1	0.1	0.1	0.2
256	0.3	0.3	0.3	0.5

Table 5.1: σ/n (%).

p	SP 1	SP 2	ML	GP
4	6.95	5.71	5.70	7.43
16	15.32	13.60	13.69	16.35
64	26.40	24.59	24.28	27.22
256	42.65	41.41	42.20	46.06

Table 5.2: m/M (%).

p	SP 1	SP 2	ML	GP
4	0	0	0	0
16	2	0	0	0
64	2	0	0	0
256	3	0	0	0

Table 5.3: n_{cc}.

p	SP 1	SP 2	ML	GP
4	7.4	58	11	0.7
16	24	100	19	2.8
64	72	132	29	0.8
256	90	164	45	1.0

Table 5.4: t (s).

Eppstein (refined four times)

$N = 131137$

$M = 392256$

$d = 5.98$

Tables 6.1-6.4 summarize some results for the Eppstein grid refined four times (see Figure 3 for an unrefined version). The partitioning time for GP decreases as the number of subsets increases, because for larger subsets checking the connectivity every time a node is reassigned can be very expensive. The behavior of the partitioning times is exactly the opposite for recursive methods. Note that for this example SP 2 always produces multiconnected subsets.

p	SP 1	SP 2	ML	GP
4	0	0	0	0
16	0	0	0	0
64	0	0	0	3.5
256	0.1	0.1	0.1	1.1
1024	0.2	0.2	0.2	1.4

Table 6.1: σ/n (%).

p	SP 1	SP 2	ML	GP
4	0.43	0.43	0.32	0.56
16	1.19	1.24	1.12	1.35
64	2.79	2.75	2.63	2.96
256	5.92	5.71	5.68	6.01
1024	11.97	11.63	11.70	12.03

Table 6.2: m/M (%).

p	SP 1	SP 2	ML	GP
4	0	1	0	0
16	0	3	0	0
64	0	3	0	0
256	0	5	0	0
1024	0	9	0	0

Table 6.3: n_{cc}.

p	SP 1	SP 2	ML	GP
4	22	193	33	2.5
16	50	361	70	7.7
64	88	478	90	6.2
256	126	740	132	3.9
1024	164	752	220	4.9

Table 6.4: t (s).

Based on these experiments and many others which have been omitted here, we can derive many comments according to the different parameters: the load balancing, the number of intersubset edges, the connectivity of the subsets and finally the partitioning time.

First, concerning the load balancing, all spectral methods realize this criteria by definition while the greedy method does not always give an optimal answer. However, the average (over the six examples) unbalancing of the subsets built by GP, is for the six previous graph tests less than 3% (Figure 4). Moreover this unbalancing tends to decrease as the number of subsets increases.

As we can see in Figure 5, ML produces in average less intersubset edges than the other methods.

While GP is designed to build connected subsets, spectral methods do not guarantee the connectivity. Nevertheless we have verified that SP 1 very rarely produces multiconnected subsets on 2D graphs but happens to produce some multiconnected subsets on 3D graphs. On the contrary, according to that criteria ML performs very well on 3D graphs but gives sometimes multiconnected subsets on 2D graphs, especially when the number of subsets increases. As for SP 2, it has a tendency to often construct multiconnected subsets (12 times for 26 experiments).

With regard to the overall execution times of the different codes (Figure 6) for the six examples, GP is by far the fastest. The overall partitioning times of SP 1 and ML are similar, while SP 2 is beyond any doubt the slowest, although it is structurally very close to SP 1.

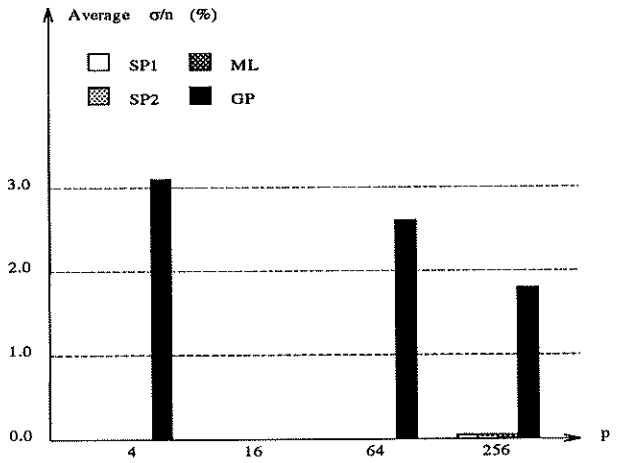

Figure 4: Average unbalancing

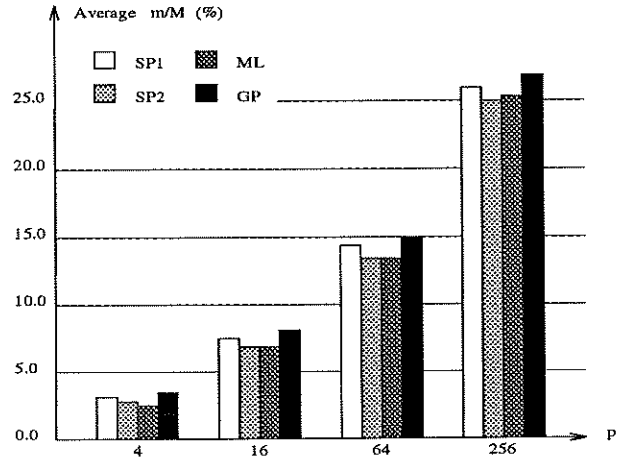

Figure 5: Average cuts

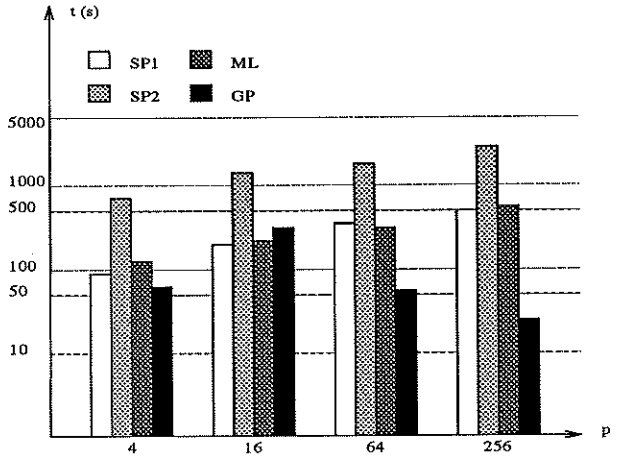

Figure 6: Overall times

8 Conclusion

We have observed that the performance of each heuristic depends on the size of the graphs or/and on the number of subsets of the partition. It appears that recursive methods are competitive for graphs with small number of nodes and edges. These methods are all the more interesting when the required number of subsets is small (≤ 16). On the contrary, the greedy method performs very well for large graphs or for large number of subsets, as the ratio of partitioning times increases dramatically and the ratio of intersubset edges goes to 1.

All methods are geometry independent with possibly the exception of the notion of boundary nodes for GP. In particular, holes in the meshes do not affect the partitioning processes. Moreover, the methods do not depend much on the dimension of the meshes or on the type of element.

Finally, from the user point of view we have found that GP like SP 1 are of very simple use because of the small number of input parameters on which they depend. The SP 2 and ML codes, on the other hand, are more sensitive to the input parameters which make these codes not so easy to use. Among the restrictions we were faced with, not being able to divide into non power of two subsets for SP 2 and ML was a severe limitation.

Future work

One of the shortcomings of our greedy heuristic is that it does not work so well for partitions into small numbers of subsets. To fix this weakness, the next step is to develop a multilevel version of GP. As it is working well for large numbers of subsets, or equivalently small subsets, the idea is to first partition the graph into very small subsets, then partition again the induced graph and so on until the desired partition is obtained.

On the other hand, it might be interesting to use these partitioning heuristics for parallel applications, such as parallel sparse matrix vector products or parallel iterative solvers, and to compare their respective efficiencies.

Acknowledgements

We thank Horst Simon for providing the code of SP 1, Bruce Hendrickson and Robert Leland for allowing us to use the Chaco package delivered to Barry Smith. We also want to thank them for their useful remarks and comments on the first draft of this paper. Many thanks to Barry Smith for introducing us to PETSc and for his help during the writing of the interfaces.

References

[1] S.T. Barnard and H.D. Simon. A fast multilevel implementation of recursive spectral bisection for partitioning unstructured problems. Technical report, NASA Ames Research Center, RNR-092-033, 1992.

[2] T.F. Chan, P. Ciarlet, Jr, and W.K. Szeto. On the optimality of the median cut spectral bisection graph partitioning method. Technical report, UCLA, CAM-93-14, 1993.

[3] P. Ciarlet, Jr and F. Lamour. An efficient low cost greedy graph partitioning heuristic. Technical report, UCLA, CAM-94-1, 1994.

[4] C. Farhat. A simple and efficient automatic FEM domain decomposer. *Computers and structures*, vol 28, n^o 5, pp 579-602, 1988.

[5] C. Farhat and H.D. Simon. TOP/DOMDEC- a software tool for mesh partitioninig and parallel processing. Technical report, NASA Ames Research Center, RNR-93-011, 1993.

[6] C.M. Fiduccia and R.M. Mattheyses. A linear-time heuristic for improving network partitions. *Proceedings of the 19th IEEE Design Automation conference*, IEEE, pp 175-181, 1982.

[7] M. Fiedler. Algebraic connectivity of graphs. *Czechoslovak Mathematical Journal*, 23, (98), pp 298-305, 1973.

[8] W. Gropp and B. Smith. Portable, Extensible Toolkit for Scientific Computation (PETSc). *Available via anonymous ftp at* **`info.mcs.anl.gov`** *in the directory* **`pub/pdetools`**.

[9] W. Gropp and B. Smith. Scalable, extensible, and portable numerical libraries. *Proceedings of the Scalable Parallel Libraries Conference, IEEE*, pp 87-93, 1993.

[10] B. Hendrickson and R. Leland. An improved spectral graph algorithm for mapping parallel computations. Technical report, SAND 92-1460, 1992.

[11] B. Hendrickson and R. Leland. The Chaco User's Guide (version 1.0). Technical report, SAND 93-2339, 1993.

[12] B. Hendrickson and R. Leland. Multidimensional spectral load balancing. Technical report, SAND 93-0074, 1993.

[13] B. Hendrickson and R. Leland. A multilevel algorithm for partitioning graphs. Technical report, SAND 93-1301, 1993.

[14] C.A. Jones. *Vertex and Edges Partitions of Graphs.* PhD thesis, Computer Science Department, Pennsylvania State University, August 1992.

[15] B.W. Kernighan and S. Lin. An efficient heuristic for partitioning graphs. *The Bell System Technical Journal*, Vol. 49, N° 2, pp 291-307, 1970.

[16] H.D. Simon. Partitioning of unstructured problems for parallel processing. *Computing Systems in Engineering*, vol 2, n^o 2/3, pp 135-148, 1991.

Appendix

Here we reproduce partitions into 16 subsets obtained from the four methods. The grid is Eppstein refined once. The corresponding data are:

$N = 2113$

$M = 6192$

$d = 5.86$

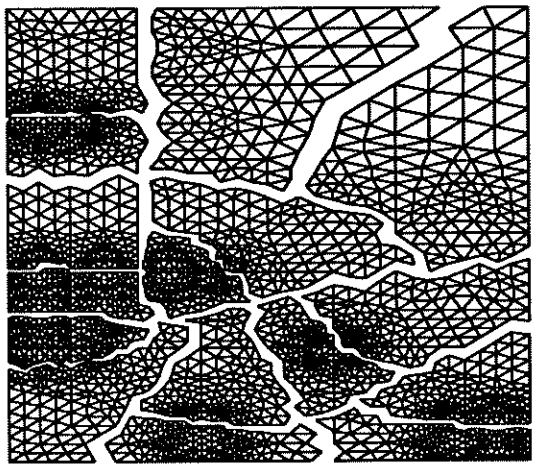

Figure 7: SP1: $\sigma/n = 0.2\%$, $m/M = 9.27\%$, $t = 0.8s$

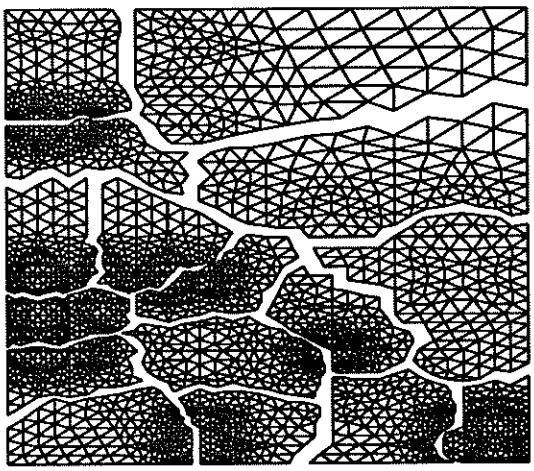

Figure 8: SP2: $\sigma/n = 0.2\%$, $m/M = 8.62\%$, $t = 4.5s$

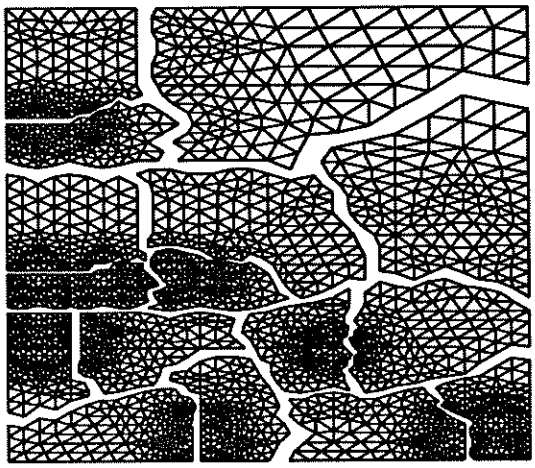

Figure 9: ML: $\sigma/n = 0.2\%$, $m/M = 8.70\%$, $t = 1.8s$

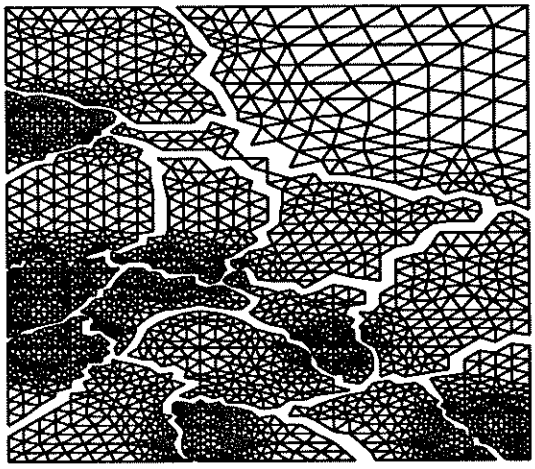

Figure 10: GP: $\sigma/n = 0.3\%$, $m/M = 10.63\%$, $t = 0.1s$