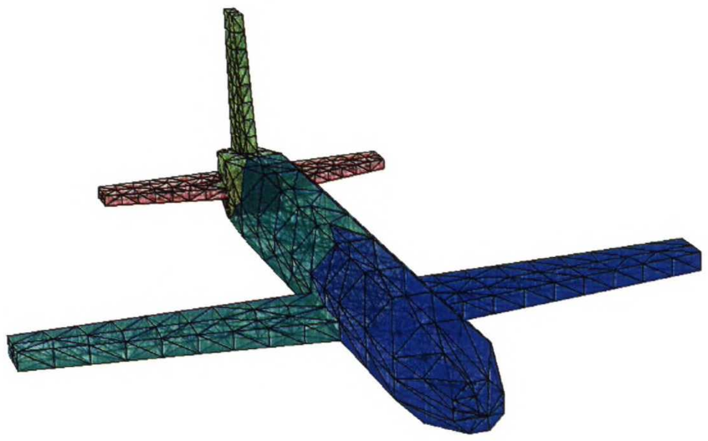
Graph Partitioning for Parallel Processing



Master's Thesis

Petri Kallberg

Teknillinen korkeakoulu	Helsinki University of Technology
Tietotekniikan osasto	Faculty of Information Technology
Tietojenkäsittelyopin laboratorio	Department of Computer Science

Espoo 1996

HELSINKI UNIVERSITY OF TECHNOLOGY ABSTRACT OF THE MASTER'S THESIS

Author:	Petri Kallberg	
Name of the thesis:	Graph partitioning for parallel processing	
Date:	May 29, 1996	Number of Pages: 6+67
Faculty:	Information Science	
Professorship:	Tik-76 Computer Science	
Supervisor:	Professor Heikki Saikkonen	
Instructor:	MSc Harri Hakula	

The efficient use of distributed memory parallel computers requires data distribution that minimizes communication between processors and at the same time takes care of load-balancing aspects. Communication and loadbalancing problems are equivalent to the general graph partitioning problem.

Graph partitioning is a NP-complete problem, however, some quite successful heuristics exist. Currently the most effective methods are based on the multilevel scheme and greedy partitioning algorithms. However, direct methods, that are based on graphs Fiedler-vector, are still very competitive. In this thesis the current state-of-the-art in graph partitioning heuristics is reviewed. Also a partitioning software for graph partitioning is developed and practical examples of partitioning 2D and 3D element meshes are presented.

Keywords: graph partitioning, parallel processing, unstructured meshes

TEKNILLINEN KORKEAKOULU DIPLOMITYÖN TIIVISTELMÄ

Tekijä:	Petri Kallberg	
Työn nimi:	Verkon jakaminen rinnakkaislaskentaa silmälläpitäen	
Päivämäärä:	29. toukokuuta 1996	Sivumäärä: 6+67
Osasto:	Tietotekniikan osasto/Tik-laitos	
Professuuri:	Tik-76 Ohjelmistojärjestelmät	
Työn valvoja:	Professori Heikki Saikkonen	
Työn ohjaaja:	DI Harri Hakula	

Hajautetunmuistin moniprosessoritietokoneiden tehokas käyttö edellyttää laskentatyön jakoa niin, että prosessorien välinen kommunikaatio tulee minimoitua ja samalla kunkin prosessorin työmäärä on tasapainotettu. Yleinen verkonjakoongelma sisältää molemmat em. vaatimukset.

Verkon jako on tunnetusti NP-täydellinen ongelma. Onneksi kuitenkin hyviä heuristiikoita ongelman ratkaisemiseksi on olemassa. Tällähetkellä parhaimpiin tuloksiin päästään monitasoisilla ahneilla jakoalgoritmeilla. Ns. suorat, verkon Fiedler-vektoriin perustuvat menetelmät ovat kuitenkin hyvin kilpailukykyisiä. Tässä työssä luodaan kattava katsaus tämän hetken verkon jako algoritmeihin. Lisäksi esitellään työn osana kehitetty verkonjako-ohjelmisto ja käytännön esimerkkejä 2D ja 3D elementtiverkkojen jaosta.

Avainsanat: verkon jako, rinnakkaislaskenta, epäsäännölliset elementtiverkot

Acknowledgements

First I would like to thank my instructor Harri Hakula for directing me to the interesting topic and for the valuable discussions and comments during the work. Without his support this work wouldn't be possible.

I would also like to thank my colleagues at the HUT Computing Centre for their valuable advices and technical support.

The research for this thesis was done while working as systems analyst working in Helsinki University of Technology (HUT) Computing Centre. This work was also supported by Center for Scientific Computing (CSC).

Espoo, May 1996

[illegible]

Petri Kallberg

Contents

Chapter 1

Introduction

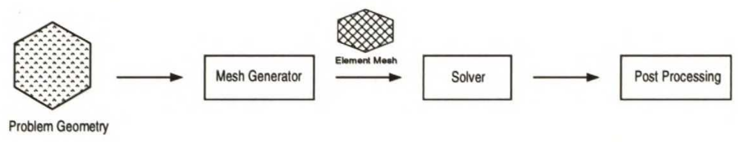
This thesis is a part of VIRKE-project whose goal is to develop a parallel computational fluid dynamics program ELMER [1], based on the Finite Element Method, which can be utilized in the industrial processes involving heat transfer due to conduction, convection or radiation, incompressible flows, laminar or turbulent, and free boundaries.

The ELMER-package includes a preprocessor, finite element routines, a matrix equations solver and a postprocessor. At the preprocessing phase the element mesh is generated and distributed to the processors. After that a system of equations is created and solved in parallel. The postprocessing collects the numeric results and visualizes them.

In a traditional serial program-architecture the solving of equations generated by the problem parameters and element mesh is the most time consuming phase. In addition to single CPU performance, also the available memory can be a limiting factor when solving really large problems. A typical architecture of a serial program is shown in Figure 1.1.

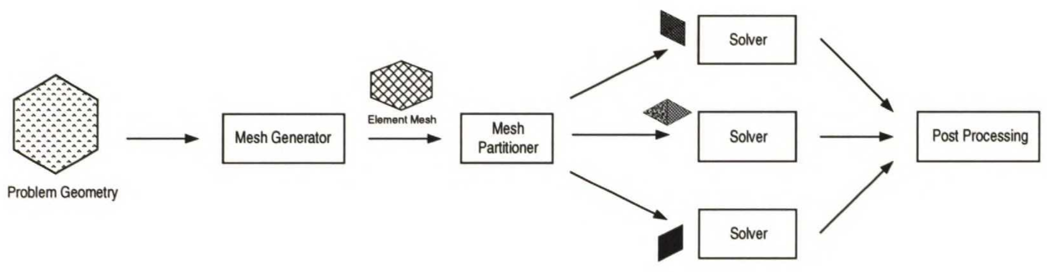
As the size of a mesh grows, more computing power and memory is required. Eventually the limits of a single CPU are encountered. Natural way to continue is to use multiple CPUs and parallel solvers. In addition to a

Figure 1.1 Serial program architecture



parallel solver, a separate partitioning program is also needed for distributing the data so that computational load can be balanced across processors and interprocessor communication is kept at minimum. This kind of program architecture could be called "partially parallel" because the first part of a pipeline is still serial as show in Figure 1.2. This is also the current goal of the VIRKE-project.

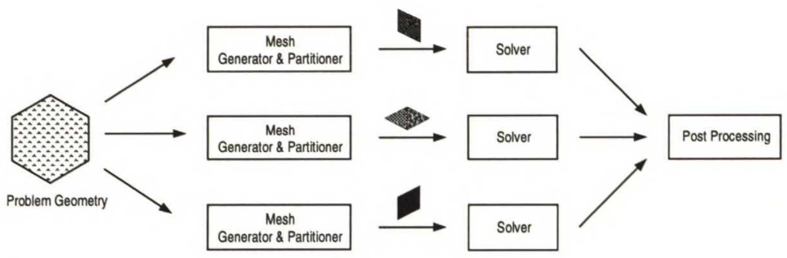
Figure 1.2 Partially parallel program architecture



At some point adding more solver processes won't decrease the total running time any more because the mesh generation and partitioning phases take most of the time. Additional partitions also increase communication between processes and require extra work at partitioning phase. In some cases, raising the degree of parallelism can actually decrease the total performance.

Higher performance can still be achieved by making the whole pipeline work in parallel as shown in figure 1.3. Graph partitioning, at its current form, is probably just a passing stage on a way to fully parallel graph generation. However, parallel mesh generation and graph partitioning are still very much research topics and no general implementations exists.

Figure 1.3 Fully parallel program architecture


Problem Geometry
Mesh Generator & Partitioner
Mesh Generator & Partitioner
Mesh Generator & Partitioner
Solver
Solver
Solver
Post Processing

This thesis is focused on graph partitioning during the preprocessing phase of the FEM solution. In parallel FEM environments typical load balancing problems can often be reduced to graph partitioning problems. Since the problem is NP-complete, only heuristic algorithms are available, though one might argue whether the Fiedler -methods are heuristic or not, but experience has shown that they perform well in most situations. However, there is no guarantee for optimality of solution. In general this is not a problem since the difference between generated partitions and the optimal solution tends to be very small.

Algorithms are divided into iterative and direct ones by the way they form the partitions. *Iterative algorithms* create partitions step-by-step, making little improvements until the local optimum is found. *Direct algorithms* are based on the computation of the *Fiedler vector* of the graph. Partitions are created directly by the vector without any intermediate stages.

The most attractive partitioning algorithms employ the *multilevel* scheme. The idea is to reduce the size of the graph before partitioning by collapsing groups of vertices together. Reduced graph can then be splitted in a fraction of time that would be required for the original graph. The complexity of partitioning algorithms is typically between $O(N \log N)$ - $O(N^3)$ while graph reduction can be done in $O(N)$. By using a multilevel partitioning algorithm substantial performance gain can be obtained without sacraficing the partition quality.

The rest of the thesis is organized as follows :

- In Chapter 2, the graph partitioning problem is defined in more detail. Also the exact solution and some not so successful heuristics are discussed.
- Chapter 3 presents a selection of iterative partitioning methods such as *Kernighan-Lin* and *Graph Growing* algorithms. The complexity of iterative algorithms is typically between $O(N \log N)$ and $O(N^2)$.
- Chapter 4 focuses on the *Spectral Bisectioning* algorithm. The actual computational challenge of the algorithm is the computation of the second eigenvector, so called *Fiedler vector*, associated with the Laplacian matrix of the graph. The complexity of Fiedler vector computation is roughly $O(N^3)$.
- Chapter 5 introduces the idea of multilevel graph partitioning. Multilevel is an extension of other partitioning algorithms rather than a totally new algorithm. The idea is to reduce the size of graph before partitioning algorithm is applied. While the complexity of partitioning algorithms varies from $O(N \log N)$ to $O(N^3)$, the graph reduction can be done in $O(N)$. Multilevel algorithm can cut down CPU usage during the partitioning phase without sacrificing the partition quality.

- In Chapter 6 performance, both CPU usage and partition quality, is discussed. While there is no significant difference in partition quality between iterative and spectral algorithms, iterative algoritms are much faster. Currently the best performance and partition quality is obtained by iterative multilevel algorithms.
- Chapter 7 summarizes the thesis and draws some conclusions.

Chapter 2

Problem

To use distributed memory parallel computers efficiently one must be able to balance computational load across processors in a way that gives each processor equal sized problem and keeps interprocessor communication at minimum level.

The key to solve load balancing problem is to think of the computational problem (in this case, solving a system of equations) as a graph where vertices are atomic subproblems (degrees of freedom) and edges are dependencies between subproblems. It is also possible to assign a different computational 'weight' to each vertex and/or edge, to better describe the original problem. A graph G obtained that way can be used as an description of the original problem. Partitioning the G so that minimum number of cut-edges are created between partitions and finding an effective distribution for the original problem can be seen as an equivalent tasks.

2.1 Definitions

This thesis is focused on partitioning of an undirected graph $G = (V, E)$. Each vertex $v_i \in V$ and edge $e_j \in E$ may have an additional attribute, weight. Weights can be used to describe a computational size of a vertex and communication load over an edge. If weights of the edges are collected into a matrix C, where $C(i, j)$ is the connection from vertex i to vertex j, then C is the connection matrix of the graph G. An unweighted graph is a special case of a weighted graph where all connection weights are constant. Since we are focused on undirected graphs, the connection matrix is always symmetric.

Creating a k-way partitioning of graph G means dividing G into k subsets $G_1 \ldots G_k$, such that $G = G_1 \cup \ldots \cup G_k$ and $G_1 \cap \ldots \cap G_k = \emptyset$. This is a generalization of the graph bisection problem (i.e. bisection is equal to a 2-way partitioning). From now on our main focus will be on the bisection problem. Thus, a 2^k-way partitioning can be obtained easily using bisectioning recursively.

The cost of bisection, or k-way partitioning, is defined as the sum of weights of such edges that have their endpoints in different partitions. In a case of bisection the cost of partitioning can be defined as

$$Cost = \sum C(a, b) \quad , where\ a \in A\ and\ b \in B.$$

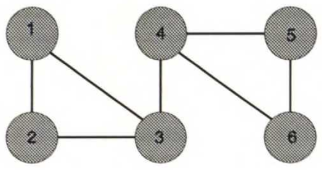
For successful load-balancing we want partitions that have certain properties. A good partitioning can be defined as follows :

- Controlled partition size. It is not always necessary or possible to define the exact number of vertices and edges in each partition, but there must be a way to control a magnitude of a partition size.
- Minimal cost. Fetching data from another CPU is usually much more expensive than using a local data. Partition cost tells us how much remote data will be used. A lower cost means more local data and thereby better performance.

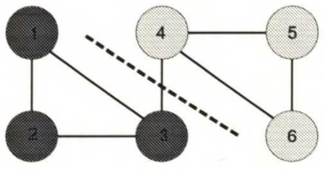
- Connected partitions. It is desirable that all vertices in one partition are connected together. This follows directly from the Minimal cost requirement because disconnected vertices increase the cost of a partitioning.

- Hardware limitations. Computation hardware used can have some limitations that must be considered. For example communication between different processors can cost a different amount of time. Partitioning algorithm should be able to take care of these limitations.

Figure 2.1 Graph G, Connection Matrix C and Sample Bisection



$$C = \begin{pmatrix} 0 & 1 & 1 & 0 & 0 & 0 \\ 1 & 0 & 1 & 0 & 0 & 0 \\ 1 & 1 & 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 & 1 & 1 \\ 0 & 0 & 0 & 1 & 0 & 1 \\ 0 & 0 & 0 & 1 & 1 & 0 \end{pmatrix}$$



2.2 Exact Solution

There are some good and some bad news. The Bad news is that the graph partitioning is NP-complete problem and an exhaustive search for finding the best partitioning is usually impossible because of a vast number of possible combinations. Assume a graph G with n vertices which are partitioned into a k subsets, each containing $p = n/k$ vertices. Then there are $\binom{n-p*(i-1)}{p}$ different ways to choose the ith subset. Since the order of subsets is unimportant, the number of different partitionings is

$$\frac{1}{k!}\binom{n}{p}\binom{n-p}{p}\cdots\binom{p}{p}.$$

Even a very small values like $n = 40$ and $k = 4$ ($\Rightarrow p = 10$) produce more than 10^{20} different partitionings. For a practical problems this method is clearly out of question.

The Good news is that usually we don't need the optimal solution. In most cases it's better to find a 'good' solution quickly rather than finding the optimal solution by searching all possible solutions.

2.3 False Starts

Here are a couple of unsuccessful heuristics for the partitioning problem. However, these heuristics might be useful in some restricted cases, since they are fast and easy to implement.

Random partitions

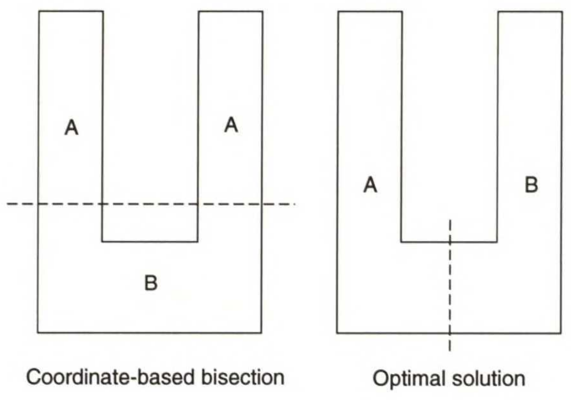
It is always possible to generate random partitions and take the best one after some predetermined time or number iterations is reached. This is a very simple and fast way to generate partitionings. Unfortunately there are a huge number of possible partitionings, of which only few are optimal or near-optimal. For example, graphs with 32 vertices with random connections has $\frac{1}{2}\binom{32}{16}$ different 2-way partitionings. Typically only 3 to 5 of these are feasible, which means that probability of finding a one by chance is less than 10^{-7} [2].

Vertex Coordinates

Another very simple way to do partitioning is to use the coordinates of the vertices. Suppose that each vertex has n coordinates $v_i = (c_1, \ldots, c_n)$, then by finding the direction of longest expansion of the domain and sorting the vertices by that coordinate, we can create a 2-way partitioning, where vertices with a smaller values form one partition and vertices with a larger values form another partition.

Obviously this method can not be used if the coordinate information is not available. Another serious problem is that the connectivity information isn't used in any way. This often leads to very undesirable results with disconnected partitions [3]. In Figure 2.2 a simple example where coordinate-based bisection is fooled by a geometry of the mesh is depicted. Picture on the left shows bisection that is generated using vertex coordinates. Compared to the optimal solution on the right, coordinate-based bisection has more cut-edges because partition A is not connected.

Figure 2.2 Coordinate-based bisection vs. optimal solution



Max Flow-Min Cut

The *Max flow-Min cut theorem* [4] states that the maximal flow values between any pair of vertices is equal to minimum cut capacity of all cuts which separate the two vertices. In our case the graph is treated as a network in which edge costs correspond to flow capacities, cut is a 2-way partitioning and the cut capacity is the cost of partitioning.

While the algorithm finds a cut with the minimum cost, it doesn't have any constraints for the partition size. Unfortunately load *balancing* isn't possible without some control over a partition size. However, since the algorithm actually finds the minimal cost 2-way partitioning, it can be used to find a lower bound for the cost of 2-way partitioning [2].

Chapter 3

Iterative Methods

The idea of iterative partitioning algorithm is to improve existing partitions gradually by swapping vertices from one partition to another. The starting point can be fully random, but better results are achieved if some more sophisticated method is used. Typically iterative methods are used to refine partitions generated by some other method.

In this chapter we look closely at two iterative graph partitioning algorithms. Because of their different nature they are suitable for complementary partitioning problems. The first, *Kernighan-Lin -algorithm* (KL), is quite dependent on an initial partitioning generator. It is possible to use a fully random initial partitions but best results are achieved when it is used for refining partitions generated by some other algorithm. The second type, *Graph growing -algorithm* (GG), does not use any initial partitions but starts from a single vertex and grows a partition around it. Results depend on a choice of initial vertex and it is difficult to predict initial which vertices lead to good partitions. GG is normally used for generating initial partitions for some other algorithm such as KL.

3.1 Definitions

If we have a graph G that has two partitions $A \subset G$ and $B \subset G$ with no common vertices, $A \cap B = \emptyset$, then we can define the internal and external cost for each vertex $a \in A$ as follows :

- Internal cost is the sum of connections from a to $x \in A$:
$$I_a = \sum_{x \in A} C(a, x).$$
- External cost is the sum of connections from a to $y \in B$:
$$E_a = \sum_{y \in B} C(a, y).$$

And similarly for I_b and E_b. Difference between external and internal costs $D_x = E_x - I_x$ can then be defined for all vertices $x \in G$.

If we now take any pair of vertices (a, b) such that $a \in A$ and $b \in B$ and swap them, what is the gain of this swap, i.e. how much is the total cost of partitioning changed ? Let Z be a total cost of such connections between partitions A and B that do not involve neither of vertices. Then we can write the cost of partitioning, T_{old}, before vertices are swapped

$$T_{old} = Z + E_a + E_b - C(a, b).$$

And after swapping vertices, new cost T_{new} is

$$T_{new} = Z + I_a + I_b + C(a, b).$$

because connections that were external become now internal and vice versa. Cost of partitioning is now reduced by $T_{old} - T_{new}$:

$$Gain = T_{old} - T_{new} = D_a + D_b - 2C(a, b).$$

This value can be used when selecting vertices that are swapped. Negative gain means that resulting partitions are 'worse' than the originals.

3.2 Kernighan-Lin

The basic version of Kernighan-Lin algorithm [2] splits a graph G into two partitions of equal size with minimal connection cost. However, algorithm can be easily modified to create n-way partitioning with more complex partition size constraints.

The algorithm starts with two initial partitions. In each iteration it tries to find such subset of vertices from each partition that swapping them gives a lower cost partitioning. Algorithm stops when no such subset can be found.

Kernighan-Lin Iteration

1. Compute D_x for all vertices $x \in G$.
2. Start a new iteration step.
3. Choose an unused pair $(a_i,\ b_i)$ that gives the maximum gain $g_i = D_{a_i} + D_{b_i} - 2C(a_i, b_i)$. Finding such a pair quickly is not a trivial task and we shall return to this subject later.
4. Swap a_i to B and b_i to A. Vertices that have been swapped are marked as 'used' so they don't get swapped again at same iteration step.
5. Recompute D-values for vertices by

$$D'_x = D_x + 2C(x, a_i) - 2C(x, b_i), \quad x \in A - a_i$$

and

$$D'_y = D_y + 2C(y, b_i) - 2C(y, a_i), \quad y \in B - b_i$$

Note that only such vertices that have connections to a_i or b_i need to be recomputed. The D-values of the other vertices are not affected by the swap.

6. Repeat steps 3 - 5, obtaining a sequence of n swapped pairs (a_i, b_i) and an associated gain-values g_i, until there are no more unused vertices.

7. Find sequence of swapped pairs that gives the maximum total gain by

$$G(k) = \sum_{i=1}^{k} g_i.$$

8. If $G(k) > 0$, interchange the first k pairs and start a new iteration step. If no such k can be found it means that all changes will lead to worse partitions and algorithm has found local minimum cost partitions.

Notice that if all vertices are swapped, i.e. $k = n$, partitions stay the same and the gain is 0. This means that swapping some pairs must have a negative gain. It might seem like a clever idea to stop searching when a first pair with a negative gain is found, but it isn't. Function G is not monotonous and after negative values there can be more pairs that yield new maximum value for G. It is necessary to swap all pairs from 1 to k to obtain gain $G(k)$ and that's why we can't stop the search at first pair with negative gain [11].

3.2.1 Finding the best pair

Finding a pair of unused vertices that gives the best gain is a dominant part of the iteration. There are three alternatives for selecting such a pair [2]. If we want to be sure that we always get the pair with the maximum gain we should use the sorting method. When D-values are sorted so that

$$D_{a_1} \geq D_{a_2} \geq \ldots \geq D_{a_n}$$

and

$$D_{b_1} \geq D_{b_2} \geq \ldots \geq D_{b_n},$$

only a few candidates for the best pair need to be considered because when scanning down the lists of D_a's and D_b's if a pair (D_{a_i}, D_{b_j}) whose sum does not exceed the maximum gain seen so far in this pass is found, then there cannot be another pair (a_k, b_l) with $k \geq i$ and $l \geq j$ with a better gain and we can stop scanning the lists. This is assuming that all connection costs are non-negative. If D-values were not sorted, it would be necessary to check all combinations of a's and b's.

The other way is simply to find vertices with the largest D-values and use them. It requires very little extra work, since the largest values can be saved during the re-computation of D-values. This method works especially on sparse connection matrices, where the probability that $C(a, b) > 0$ is small.

It is also possible to save two or three largest D-values from each partition and select the best pair among them. This requires little more work, but also performs better in situations where the largest pair does not give the maximum gain because the connection cost between vertices, $C(a, b)$, is too high. Experience indicates [2] that the saving of the three largest D-values from each partition is enough even for relatively dense connection matrices. This method is a good compromise between speed and quality. It is faster than sorting and yet it finds almost always the pair with the best gain.

3.2.2 Variations of Kernighan-Lin

The original KL algorithm has been improved in many ways. Here are some common variations of the basic algorithm.

Orphan vertices

An orphan vertices are vertices that does not have any connections to other vertices. Because they don't have connections, they don't change the connection cost of partitionings. With orphan vertices it is possible to relax partition size constraints. Suppose that we have a graph with n vertices and we want partitions that contain at least n_1 and at most n_2 vertices ($n_1 + n_2 = n$).

We can now add $2n_2 - n$ orphan vertices and create partitions as usually. Orphan vertices are assigned to both partitions so that the connection cost is minimized. After removing orphans, we have two partitions that satisfy previously given size constraints [2].

Boundary Kernighan-Lin

The idea of the boundary version is based on the observation that most of the swapped vertices are on the partition boundaries. If we forget all vertices not on the boundary, we can save a lot of work when choosing the optimal subsets for swapping. Keeping count of the boundaries requires some extra work, but the total computational effort is usually smaller, because of smaller number of active vertices. Use of the boundary version of the algorithm doesn't effect significantly the quality of partitions generated [5].

K-way partitioning

In a many cases the partitioning algorithm must be able to handle more than two partitions. Usually this is solved by recursive bisection. That is, we first obtain a 2-way partitioning and then divide each part again. After $\log k$ steps the graph is partitioned into k parts. However with a couple of simple modifications it is possible to generalize the original KL algorithm to handle an arbitrary number of partitions directly [8]. Instead of a single gain-value, gains need to be computed against every partition. Gain-values are computed for single vertices instead of pairs, because we might want to move vertices some other fashion than a strict swaping. This also gives a possibility to generate partitions with unequal number of vertices. When choosing the next vertex to swap only such moves that satisfy partition size constraints are considered. For example, if we say that no more than a 5% imbalance is allowed, vertices cannot be moved to partitions that are already too large or from partitions that are already too small, even if those moves would give the best gains. If we want equal-sized partitions we should move vertices only from partitions of at least average size to partitions below average size.

Linear-time algorithm

Each iteration of the original KL algorithm takes $O(|E| \log |E|)$ operations, where E is the number of vertices. By using appropriate data structures complexity of the algorithm can be reduced to $O(|E|)$. One such modification is presented by Fiduccia and Mattheyses in [10].

3.3 Graph Growing

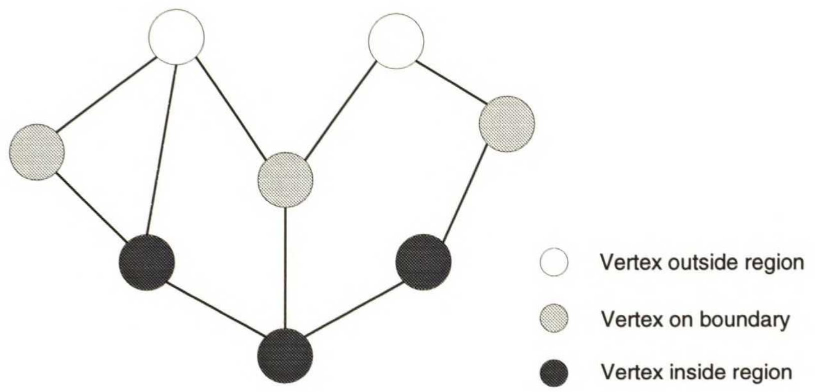
Graph Growing

The Graph Growing algorithm [5] starts from a random vertex and grows a region around it in breath-first fashion, until desired number of the vertices or vertex weight has been included. Partitions generated by graph growing are, by definition, always connected. However, the quality of partitions depends totally on a chose of the initial vertex.

The main problem in the GG algorithm is that there is no general way to know which vertices lead to good partitions. Easy but time consuming solution is to generate multiple partitionings, by selecting many starting vertices, and then simply choosing the one with the lowest connection cost.

In Figure 3.1 the darker vertices are added to the partition in previous iterations and the grey vertices will be in the partition after the next iteration. Note that there are no rules that would tell the order tell in which the vertices on a partition boundary must be included. If we need only a one more vertex, it can be any of those on the current boundary.

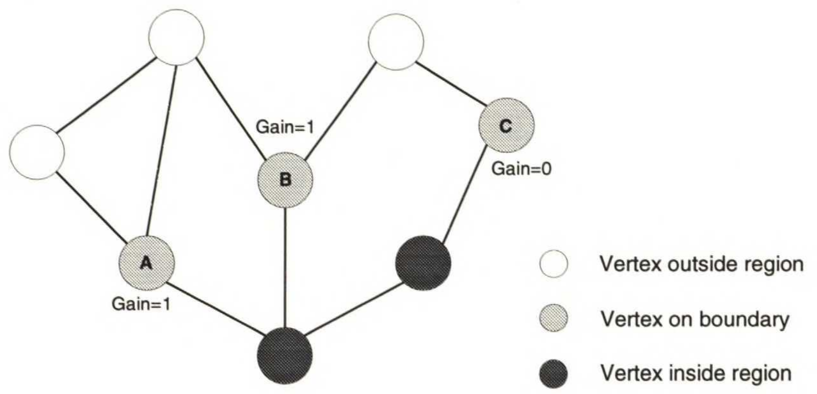
Figure 3.1 Graph Growing



Greedy Graph Growing

Greedy Graph Growing (GGG) [5] is a modification of GG. Instead of growing a region in a strict breath-first fashion we can compute a gain value for each vertex on a region boundary. Gain value tells how the costs of the partitions will change if a vertex is added to a growing region. Vertices are then sorted by their gains and the vertex that has the biggest decrease (or smallest increase) of partition cost is inserted first. Then the gains of adjacent vertices are updated and new vertices are joined to region boundary. Gain values are needed only for the vertices on boundary, so it is not necessary to precompute all gain values as in KL.

Figure 3.2 Greedy Graph Growing



In Figure 3.2 we have three vertices, A, B, C, on partition boundary. C is added first, because it doesn't increase the connection cost of partition. Both A and B would add one additional connection.

GGG has the same problem as basic GG. The quality of partitions depends on the choice of initial vertex. However this dependency is not as strong as in GG. Usually GGG gives better partitions with less work than GG. Even though some extra work is done while computing gains, greedy version can still be faster because to find a good partitioning it isn't necessary to generate as many different partitions as with GG.

Chapter 4

Direct Methods

The name of this chapter could be 'Eigenvector based methods', because all widely used direct graph partitioning algorithms are based on the second eigenvector, the so-called *Fiedler vector*, associated with the graph. The effective computation of Fiedler vector is the actual computational challenge of these methods, otherwise these algorithms are very simple.

4.1 Definitions

Graphs laplacian and incidence matrices have some interesting properties that can be used in graph partitioning. For simplicity these properties are just listed here. For details and proof see [12] and [13].

Laplacian matrix $L(G)$ is quite similar to $C(G)$. If we mark connections with -1 and add degrees of nodes on the diagonal we get $L(G)$.

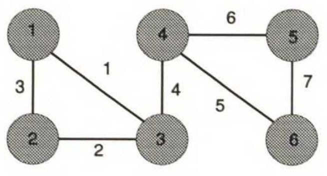
$$L(G)(i,j) = \begin{cases} \deg(i), & \text{if } i = j. \\ -1, & \text{if } i \neq j \text{ and } C(i,j) \neq 0. \\ 0, & \text{otherwise.} \end{cases}$$

Incidence matrix $I(G)$ is a matrix with one row for each node and one column for each edge. If G has an edge e between vertices i and j then $I(G)(e,i) = 1$ and $I(G)(e,j) = -1$. All other elements in a column e of $I(G)$ are zeros.

$L(G)$ and $I(G)$ have the following properties :

- By definition $L(G)$ is symmetric. This means that the eigenvalues of $L(G)$ are real and its eigenvectors are real and orthogonal.
- $L(G) \cdot e^T = 0$ where $e = [1 \ldots 1]$.
- $I(G) \cdot I(G)^T = L(G)$.
- Eigenvalues of $L(G)$ are non-negative. $0 \leq \lambda_1 \leq \lambda_2 \leq \ldots \leq \lambda_n$.
- The number of connected components of G is equal to number of λ_i equal to 0. In particular, $\lambda_2 \neq 0$ if and only if G is connected.

Figure 4.1 Graph G, I(G) and L(G)



$$I(G) = \begin{pmatrix} -1 & 0 & -1 & 0 & 0 & 0 & 0 \\ 0 & -1 & 1 & 0 & 0 & 0 & 0 \\ 1 & 1 & 0 & -1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & -1 & -1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 & -1 \\ 0 & 0 & 0 & 0 & 1 & 0 & 1 \end{pmatrix} \qquad L(G) = \begin{pmatrix} 2 & -1 & -1 & 0 & 0 & 0 \\ -1 & 2 & -1 & 0 & 0 & 0 \\ -1 & -1 & 3 & -1 & 0 & 0 \\ 0 & 0 & -1 & 3 & -1 & -1 \\ 0 & 0 & 0 & -1 & 2 & -1 \\ 0 & 0 & 0 & -1 & -1 & 2 \end{pmatrix}$$

4.2 Spectral Bisection

Spectral Bisection (SB) is based on the properties of the second eigenvalue, λ_2, and the associated eigenvector, v_2, of the laplacian matrix of the graph. The tricky part of SB is a computation of eigenvalues and vectors. The bisectioning algorithm itself is very simple after we have found the right eigenvalue and eigenvector pair.

Spectral Bisection algorithm

1. Compute λ_2 and the associated eigenvector v_2 of $L(G)$.
   This is the tricky part, see chapter 4.3 for details.
2. Divide nodes into partitions N^- and N^+ by
   $$n \in \begin{cases} N^-, & \text{if } v_2(n) < 0. \\ N^+, & \text{otherwise.} \end{cases}$$
3. Apply algorithm recursively to new partitions.

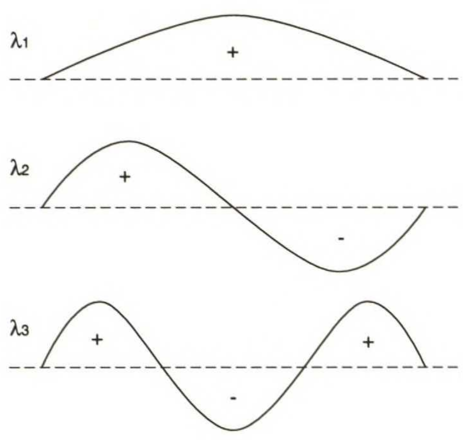
Fiedler has proven in [15] that if the graph G is connected and partitions N^- and N^+ are generated by this algorithm, then N^- is always connected. Similarly N^+ is also connected if for every n holds that $v_2(n) \neq 0$. However, if there are 0's in v_2 partition connectivity can not be guaranteed.

4.2.1 'Vibrating string' -analogy

It might be difficult to understand how eigenvectors can be used to partition graphs. The following analogy [13] with a vibrating string tries to give some motivation for the spectral bisection algorithm.

Picture a taut string that begins to vibrate when it is plucked. We know from physics and music that it has certain modes of vibration or harmonics. If we take snapshots of these modes they would look like strings in Figure 4.2.

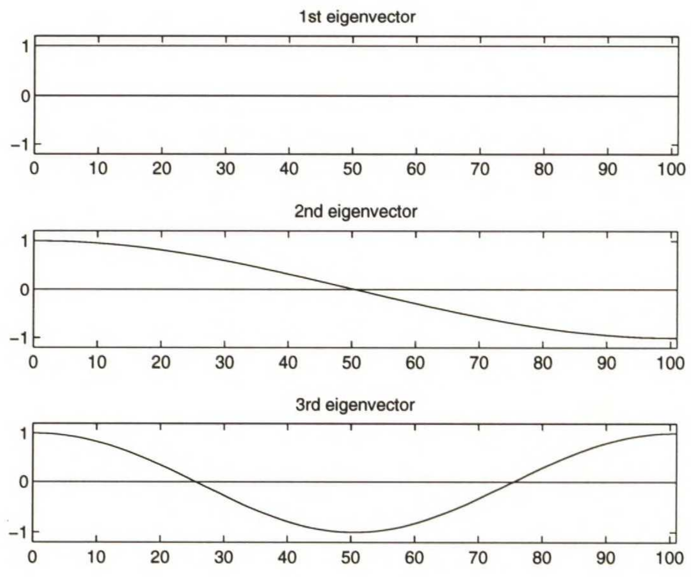
Figure 4.2 Modes of a Vibrating String



Those parts of a string that are above the rest position are labeled by '+' and parts that are below it are labeled by '-'. In the case of second frequency, λ_2, half of the string is above and half below the rest position. Effectively this is bisecting the string into two connected components that are equal sized. It turns out that if we build this vibrating string from a finite set of identical masses (vertices) connected by identical springs (edges), write down Newton's Laws of motion for the masses, and solve for the frequencies and shapes of the vibrational modes, we will get precisely the eigenvalues and eigenvectors of the laplacian matrix $L(G)$.

In Figure 4.3 are the first three eigenvectors of 100 vertex long chain. There is actually a little difference between the vibrating string and eigenvectors. Strings were fixed at both ends but eigenvectors have no fixed points. To make analogy between spectral bisection and vibrating string exact, we have to make a little modification to our example. Instead of free masses there are n horizontal rods and on each rod a mass m can slide frictionlessly. These masses are connected with identical springs as before. The only difference is that masses at the end of chain are now connected to one moving mass rather than to one mass and one fixed point.

Figure 4.3 Eigenvectors



In case of a more complex graphs than simple chains, the same intuition applies. Another easy to understand case is generic planar graph. We can think a planar graph as a kind of trampoline. The second mode of vibration bisects the graph (trampoline) into two parts. Vertices are divided into two partitions so that ones above the rest position form one partition, and ones below it form another.

4.3 Estimating the Fiedler vector

Computing of λ_2 and v_2 of $L(G)$ is the most important part of spectral bisection algorithm. Many mathematical libraries have functions to compute eigenvalues and eigenvectors of matrices, such as *eig* in Matlab, *dsyevx* in LAPACK or *pdsyevx* in ScaLAPACK. However, in this case these are not very cost effective solutions because they use dense matrices, while our matrices are typically sparse, and their running time is proportional to N^3 where N is a matrix dimension.

Lanczos algorithm offers another, more attractive, method for finding required the λ_2 and v_2. The idea of Lanczos iteration is to find a tridiagonal matrix $T(G)$ whose eigenvalues are good approximations of the original matrix $L(G)$. After we have found this $T(G)$ it is quite trivial to compute the right eigenvalue and vector. More details are presented in Chapter 4.3.1.

There is also another reason to use tridiagonal approximation of $L(G)$. Since we are not actually interested in eigenvalues, but eigenvectors associated with them, a good estimate of eigenvalues is good enough for our purposes. It is not necessary to know the actual eigenvalues. This means that $T(G)$ can be much smaller than $L(G)$ which gives us an additional speed-up. The size of a matrix depends on how accurate estimates we need. As the matrix size grows, eigenvalue approximations get more accurate.[1] The convergence of eigenvalues is fastest at the both ends of spectrum. Because the only eigenvalue that we are interested in, λ_2, is at the end of spectrum, we need only a few iterations to get accurate results.

Our Lanczos algorithm is based on a version presented in [16]. This is only a basic version of the algorithm, but in most cases it performs quite well. There are also a lot of variations of the algorithm for more special applications like the ones presented in [17], but in our case they are 'overkill' because we don't need very accurate eigenvalue approximations (see Chapter 4.3.2).

[1]Although it is not possible to grow $T(G)$ endlessly because of the well-known 'breakdown phenomenon.

4.3.1 Lanczos algorithm

Following algorithm for computing kth eigenvalue is presented in [18].

Let T_r denote the leading r-by-r principal sub-matrix of

$$T = \begin{pmatrix} a_1 & b_1 & & \cdots & 0 \\ b_1 & a_2 & b_2 & & \vdots \\ & b_2 & \ddots & \ddots & \\ \vdots & & \ddots & \ddots & b_{n-1} \\ 0 & \cdots & & b_{n-1} & a_n \end{pmatrix}$$

and define the polynomials $p_r(x) = \det(T_r - xI)$, $r = 1 : n$. A simple determinental expansion can be used to show that

$$p_r(x) = (a_r - x)p_{r-1}(x) - b_{r-1}^2 p_{r-2}(x)$$

for $r = 2 : n$ if we set $p_0(x) = 1$. Because $p_n(x)$ can be evaluated in $O(n)$ flops, it is feasible to find its roots by using the method of bisection. For example, if $p_n(y)p_n(z) < 0$ and $y < z$, then the iteration shown in Figure 4.4 is guaranteed to terminate with $(y + z)/2$ an approximate zero of $p_n(x)$, i.e., an approximate eigenvalue of T. The iteration converges linearly in that the error is approximately halved at each step.

Sometimes it is necessary to compute only the kth largest eigenvalue of T for some prescribed value of k (in our case $k = 2$). This can be done efficiently by using the bisection idea and the Sturm Sequence Property.

Sturm Sequence Property

If the tridiagonal matrix is un-reduced, then the eigenvalues of T_{r-1} strictly separate the eigenvalues of T_r:

$$\lambda_n(T_n) < \lambda_{n-1}(T_{n-1}) < \lambda_{n-1}(T_n) < \cdots < \lambda_2(T_n) < \lambda_1(T_{n-1}) < \lambda_1(T_n).$$

Figure 4.4 Eigenvalue iteration

```
while $|y - z| > \varepsilon(|y| + |z|)$
        $x = (y + z)/2$
        if $p_n(x)p_n(y) < 0$
                z = x
        else
                y = x
        end
end
```

Moreover, if $a(\lambda)$ denotes the number of sign changes in the sequence

$$p_0(\lambda), p_1(\lambda), \ldots, p_n(\lambda)$$

then $a(\lambda)$ equals the number of T's eigenvalues that are less than λ.

After λ_2 is found, the Fiedler vector can be computed via inverse iteration algorithm [19] shown in Figure 4.5.

Figure 4.5 Inverse Iteration

```
for $k = 1, 2, \ldots$
        Solve $(A - \mu I)z^{(k)} = q^{k-1}$
        $q^{(k)} = z^{(k)}/||z^{(k)}||_2$
        $\lambda^{(k)} = q^{(k)^H} A q^k$
end
```

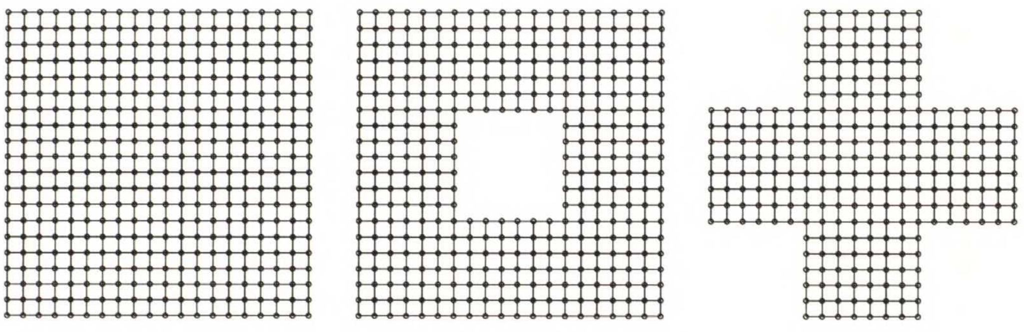
Inverse iteration is just the power method applied to $(A - \mu I)^{-1}$.

4.3.2 Performance of Lanczos algorithm

Previously in this chapter it was said that we don't need to know the exact eigenvalues of L to be able to split the graph. The next question is, how accurate eigenvalues and vectors have to be in order to generate good partitions ?

Three test graphs shown in Figure 4.6 were used to compare Lanczos algorithm with accurate values based on the original laplacian matrices. The purpose of the first graph, Basic (400 vertices), is to test algorithm on large structured areas. The second graph, Doughnut (364 vertices), is like the first one, but because there is a hole in the middle of graph, it has both inner and outer boundaries. The third graph, Cross (256 vertices), is not connected in the same way as previous graphs, but it has four extensions that are connected only at the middle.

Figure 4.6 Lanczos-test Graphs

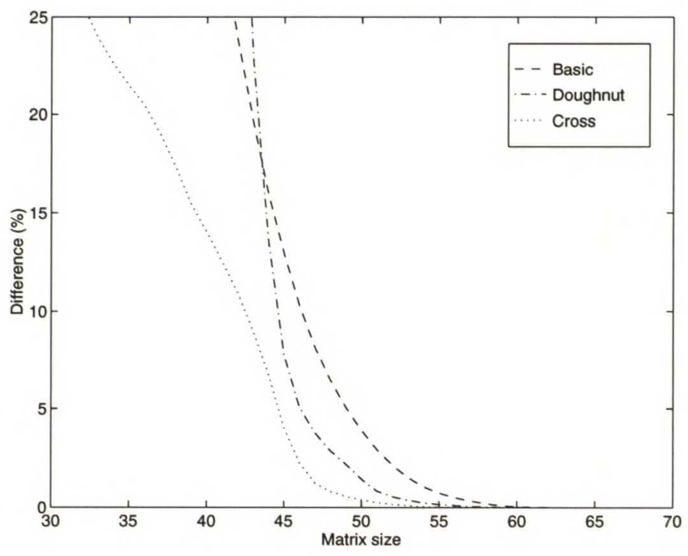


(a) Basic (b) Doughnut (c) Cross

Eigenvalues

The first test compared eigenvalues of tridiagonal Lanczos matrix and original laplacian matrix. Figure 4.7 shows a convergence of the second eigenvalue. On the horizontal axis is the number of Lanczos iterations and on the vertical one is the difference between λ_2 of L and λ_2 computed for Lanczos matrix. We can see that all three test cases got good approximation of λ_2 after 45-50 iterations, i.e. the dimension of Lanczos matrix was between 45-50. This is relatively fast and small compared to the original laplacian matrices whose dimensions were 400, 364 and 256, respectively.

Figure 4.7 Eigenvalue convergence

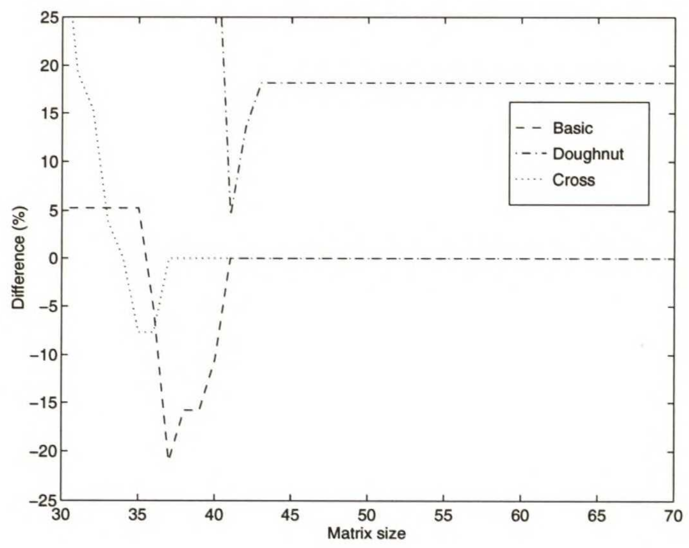


Partitions

The second test compared the quality of the partitions. Computation of eigenvector approximations was based on eigenvalues computed in previous tests. We didn't compare these approximations directly with the real eigenvector because eigenvectors are only used to generate partitions.

Instead we compared bisections generated by eigenvector approximations and the real eigenvector. In Figure 4.8 on the horizontal axis is the number of Lanczos iterations. This is the same scale that was used in Figure 4.7. On the vertical axis is the difference in partition quality measured by connection cost. Values below 0% mean that eigenvector approximation computed with Lanczos algorithm gives better partitions than the real eigenvector of laplacian matrix. It is worth noting that there is no quarantee for optimality in neither case.

Figure 4.8 Connection cost



If we compare Figures 4.7 and 4.8 we see that the eigenvectors, measured by connection cost of partitions, converged even faster than the eigenvalue approximations !

The reason why the Lanczos iteration didn't find the correct partitions in "Doughnut" -case is in the original laplacian matrix that has a dual-eigenvalue. Because values are equal it is impossible to say which one is λ_2 and which is λ_3. Unfortunately the eigenvectors associated with λ_2 and λ_3 are not equal and in this case the algorithm has chosen a wrong vector. This kind of behavior can only be avoided if in case of multiple equal eigenvalues all associated eigenvectors are checked and the one that gives the lowest connection cost is selected.

Chapter 5

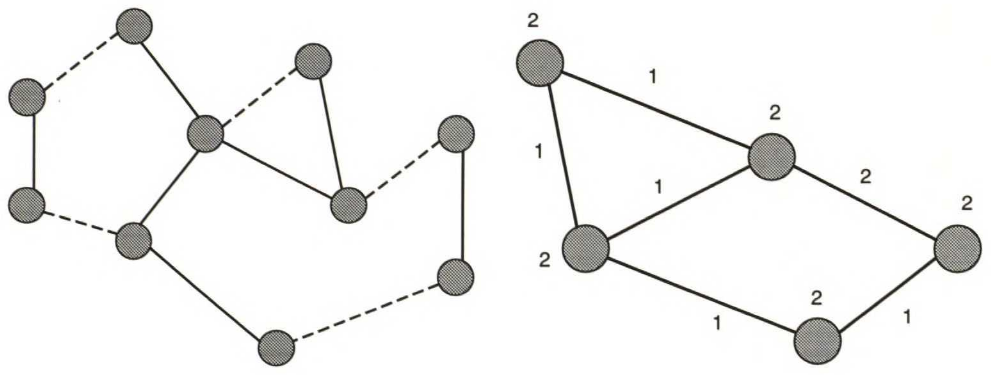
Multilevel Partitioning

The idea of multilevel graph partitioning is to approximate the original graph by a sequence of smaller graphs. The smallest graph is then partitioned using some suitable method and the partitions are then projected back to the original graph. The advantage of multilevel partitioning algorithm is that the graph coarsening can be done in time proportional to the number of edges, while the complexity of partitioning increases exponentially with the number of vertices.

In Figure 5.1 on the left is the original graph. Edges that will be collapsed at the next stage are marked with dashed lines. We shall explain later how these edges are selected. In the same figure on the right is the same graph after collapsing those marked edges.

Note that even though the original graph was unweighted, the reduced graph has weights on both vertices and edges. This is necessary because we want to preserve information about the structure of the original graph. Without weights it would be impossible to tell the actual size or connection cost of partitions after they were propagated back to the original graph. It is important that the partitioning algorithm used with multilevel partitioning can handle graphs with weighted edges and vertices.

Figure 5.1 Original and Reduced Graph



(a) Original (b) Reduced

5.1 Graph Reduction

There are basicly two kinds of graph reduction algorithms. Some algorithms, like *Random Matching* (RM), just match random vertices together, while others, like *Heavy Edge Matching* (HEM) and *Heavy Clique Matching* (HCM), use the connectivity information to find a groups of tightly connected vertices.

Random algorithms work quite well when degrees of vertices are close to the average degree of the graph. Graphs from finite element applications (FEM) are typically in this category. However, if a graph has tightly connected components it is usually better to keep those components together and use something like HEM or HCM. Splitting tightly connected components into different partitions typically leads to unnecessary increase in the connection cost of partitioning.

Since reduction algorithms have roughly similar complexity, it is usually better to use some more advanced algorithm like HEM or HCM than try to save a little time using RM. Carelessly done reduction can harm partitioning algorithm and lead to significantly higher cost partitions.

Random Matching

Random Matching (RM) [5] visits vertices in random order. If vertex has not been matched yet, we randomly select one of its unmatched neighbors and mark both the vertices and the edge between them as matched. If vertex has no unmatched neighbors, it remains unmatched. This continues until no more vertices can be matched. After that, matched vertices are joined together and are marked as unmatched for the next reduction step.

Heavy Edge Matching

Heavy Edge Matching (HEM) [5] visits vertices in random order, same way as RM did. However, when selecting a neighbor to match with, it chooses the one that is connected with the heaviest edge. Naturally only such neighbors that are not already matched are considered.

The idea of HEM is to minimize the edge weights of the reduced graph. Smaller edge weight typically leads to smaller connection cost when reduced graph is partitioned. This algorithm doesn't guarantee that the edge weight of the reduced graph is minimized, but experience has shown that it works very well.

Modified Heavy Edge Matching

Modified Heavy Edge Matching (MHEM) [7] tries to minimize the average degree of the graph. Again vertices are visited in random order and matched with the neighbor that has the heaviest connection. If there are more than one vertex to choose from, the vertex that has most connections from it's neighbors to a matching vertex, is chosen.

Analysis of the multilevel bisection algorithm in [6] shows that a good edge-cut of a coarser graph is closer to a good edge-cut of the original graph if the average degree of the coarser graph is small and/or the average weight of the edges in the coarser graph is small [7].

Light Edge Matching

Light Edge Matching (LEM) [5] is like HEM, but instead of matching heavily connected vertices it matches neighbors that has the lightest edge between them. Reduced graphs produced by LEM have typically much higher average degree than the original graphs. This kind of graphs are easier to handle for some partitioning algorithms like Kernighan-Lin. The choice between HEM and LEM depends on what kind of partitioning algorithm is selected for the reduced graph.

Heavy Clique Matching

Heavy Clique Matching (HCM) [5] tries to find subgraphs that are fully or almost fully connected. The idea is very similar to the HEM but instead of just matching vertices with the heaviest edge between them, HCM joins vertices that have the highest edge density.

For a pair of vertices (u, v) edge density is defined as follows :

$$EdgeDensity = \frac{2(CE(u) + CE(v) + EW(u,v))}{(VW(u) + VW(v))(VW(u) + VW(v) - 1)}$$

where $VW(x)$ is the weight of vertex x, $EW(x,y)$ is the weight of edge between vertices x and y, and $CE(x)$ is the total weight of edges already collapsed into a vertex x. Vertices that are not connected in any way have edge density of 0 and vertices that form a clique have edge density of 1.

5.2 Graph Refinement

After the reduced graph is partitioned the results have to be projected back to the original graph. This can be done by simply assigning all vertices to the same partition as their parent in the reduced graph. However, since the original graph is much finer and has many more degrees of freedom than the reduced one, these projected results can usually still be improved by swapping some vertices from one partition to another.

Three common refinement-algorithms are presented here. They are all based on Kernighan-Lin algorithm that was presented previously in Chapter 3.2. KL-based algorithms suit very well in this kind of situations because good initial partitions are already available.

Kernighan-Lin Refinement

Kernighan-Lin refinement (KLR) [5] simply runs KL partitioning algorithm with projected partitions. Since those partitions are already quite good, algorithm converges fast, typically within three to five iterations.

To further improve the performance of the algorithm, some additional stopping conditions can be set. For example, continue until N swaps that do not decrease the cost of partitioning, are made. Since the original partitions were already good there is only a small number of swaps that will lead to better partitions. All other moves will increase the cost of partitioning.

Greedy Refinement

Experiments show that the largest gain is obtained during the first iteration step. *Greedy refinement* (GR) [5] runs only a single iteration of KL algorithm. Iteration is stopped immediately when no more swaps with positive gain are found. This reduces the complexity of refinement phase. Unfortunately the number of swapped vertices and total running time does not change in asymptotic terms because a lot of work has to be done while building appropriate data structures before iteration.

Boundary Refinement

Almost all of the swaps in refinement phase are done between vertices on a partition boundary. *Boundary refinement* [5] uses this information to skip unnecessary work.

The idea is to focus on those vertices that are on a partition boundary and forget all other vertices. After every iteration algorithm has to check for new vertices that might have got onto a boundary due to swapping of vertices that were already on the partition boundary. Boundary-idea can applied to both KLR (BKLR) and GR (BGR).

Boundary method can save a lot of work on large graphs because the data structures used by refinement algorithm become much smaller and are faster to update. More complex the original refinement algorithm is, more advantage is gained by using a boundary method.

5.3 Aspects of Multilevel Partitioning

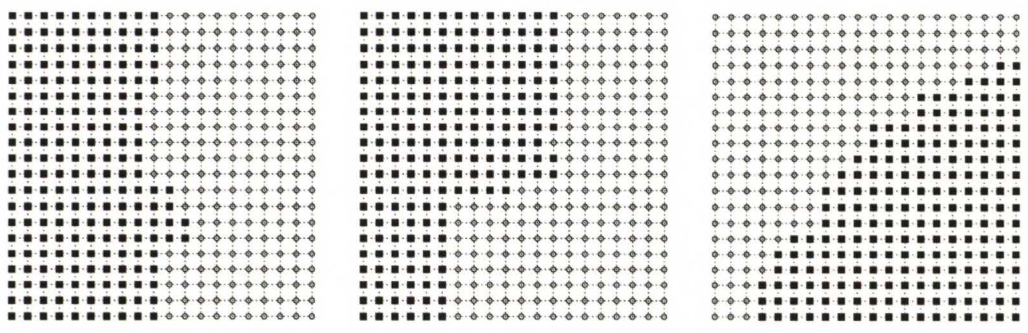
Effective use of multilevel partitioning algorithms requires some understading of how different parameters affect resulting partitions. The use of unappropriate partitioning methods or parameters can cause badly shaped partitions, high connection costs and long running times. The choice of methods and parameters depends on a graph that is partitioned. If one wishes to split multiple graphs that have somewhat similar connection structure, it might be worth while to make some experiments with different combinations of algorithms and parameters.

Figures 5.2 and 5.3 show how changing the coarsening level can affect the bisection of graph. The graph is constructed from 400 vertices that are laid on a regular grid. In each figure the same graph is bisected using the multilevel partitioning algorithm. Coarsening was made by using HEM and combination of BKLR and BGR algorithms was used to refine partitions. The actual bisection was made by using Spectral Bisectioning algorithm.

In Figure 5.2 are three bisections that were generated by coarsening the original graph down to 4, 8 and 16 vertices. To see better how coarsening affects bisectioning algorithm, no refining was used after bisectioning.

The Effect of coarsening is shown on the edge of partitions. As the size of bisected graph grows from 4 to 16 vertices, the edge gets more details, i.e. becames less "block-like". If no coarsening and no refining is used the edge of partitions would split square diagonaly. This kind of behavior can be seen in the rightmost picture.

Figure 5.2 Partitions before refining



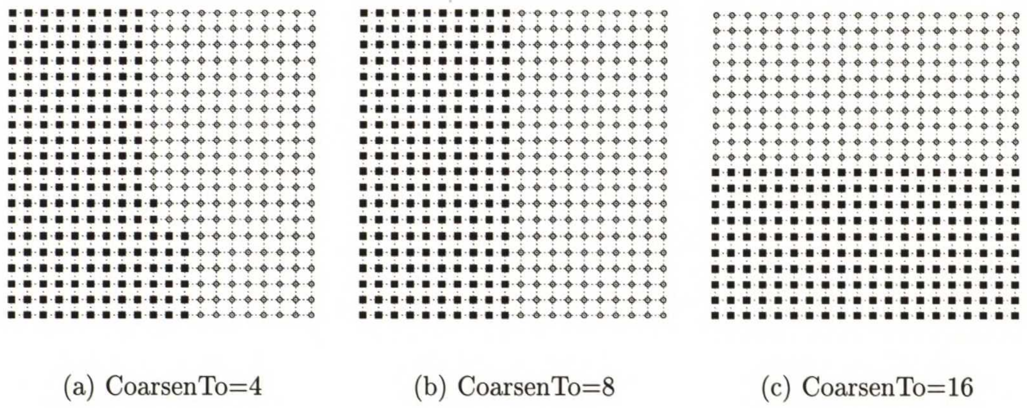
(a) CoarsenTo=4 (b) CoarsenTo=8 (c) CoarsenTo=16

Coarsening can significantly reduce CPU time used in graph partitioning. However, using too much coarsening can destroy some of the connection information and thereby lead to higher connection costs and badly shaped partitions. On the other hand, coarsening can also help bisectioning algorithm by hiding "unimportant" information. There is no general rule, how much coarsening one should use. Typically iterative partitioning algorithms work better with larger, less coarsened, graphs and eigenvector based like more coarsened graphs.

In Figure 5.3 are the same three partitionings after refining. Both second and third bisection, where the coarsest graph was 8 and 16 vertices respectively, have minimal connection costs after refining.

However in the leftmost picture, where graph was coarsened down to four vertices before partitioning, refinement algorithm was no longer able to find optimal partitions. This is due to excessive coarsening, too much connection information was lost during graph coarsening.

Figure 5.3 Partitions after refining


(a) CoarsenTo=4 (b) CoarsenTo=8 (c) CoarsenTo=16

Chapter 6

Results

In this Chapter the performance of iterative and direct partitioning algorithms is compared. Also the effect of the multilevel scheme on both connection cost and CPU usage is reviewed.

Performance of graph partitioning algorithm can be measured on two different scales: *connection cost* and *CPU usage.* Partitioning algorithms are trying to minimize the connection cost, so that inter-processor communication is kept at the minimum level and parallel FEM-solver can obtain the maximum performance. However, even a very good partitioning can be unsatisfying if too much CPU time is wasted in the partitioning phase.

Whether to emphasize on partition quality or CPU time depends on the underlaying application. If partitionings can be re-used it is justified to use more CPU time to generate good partitions. On the other hand, if only a "throwaway" partitions are required it is better to cut the CPU usage at the expense of partition quality.

6.1 Testing Environment

6.1.1 Hardware

The Performance-tests were run on a Digital AlphaServer 2100 with three 300MHz CPUs and 1GB of memory. One of CPUs was dedicated for the graph partitioning process while the other two had moderate interactive load at that time.

All file-I/O was left out because disks were mounted over network by NFS. There were also many programs running on the other two CPUs that used some unknown amount of machines I/O-capacity. Under these conditions it would have been meaningless to measure time taken by I/O. Since all partitioning algorithms read and write almost same amount of data and use same I/O-routines, the time used in file-I/O can be seen as function of graph size that is same for all algorithms. It is clear that when the size of graph grows it will become a major problem. However the focus of this thesis is on graph partitioning algorithms, not on I/O-performance.

6.1.2 Software

For testing purposes a graph partitioning software called `heli` was implemented according to methods presented in previous chapters.

As an input `heli` reads a mesh generated by a separate program. After that it builds a connection graph, generates partitions and finally writes each partition to separate files on a disk. Parallel solver processes can then read only the information they are interested in.

The actual graph partitioning in `heli` is done by using the multilevel algorithm implemented by the `metis`-library [20]. Metis is a publicly available library and collection of sample programs for unstructured graph partitioning and sparse matrix ordering. Both `heli` and `metis` are supporting the following matching, partitioning and refinement algorithms :

Matching: Random (RM), Heavy-edge (HEM), Light-edge (LEM), Heavy-clique (HCM), Modified Heavy-edge (MHEM), Sorted Random (SRM), Sorted Heavy-edge (SHEM) and Sorted Modified Heavy-edge (SMHEM).

Partitioning: Graph Growing (GG), Greedy Graph Growing (GGG), Spectral Bisection (SB) and Combination of Graph Growing & Boundary Kernighan-Lin (GGKL).

Refinement: Greedy (GR), Kernighan-Lin (KLR), Combination of Greedy and Kernighan-Lin (GKLR), Boundary Greedy (BGR), Boundary Kernighan-Lin (BKL) and Combination of Boundary Greedy & Boundary Kernighan-Lin (BGKLR).

Sorting versions of matching algorithms first sort vertices in increasing order of vertex degree and while finding matchings they browse vertices in this order. Sorting matching algorithms usually find larger matchings than their non-sorting counterparts.

By default SHEM is used for graph reduction, SB for partitioning the reduced graph and GKLR for partition refinement. If the size of the reduced graph is not explicitly defined, the graph is reduced down to 100 vertices before partitioning. All these parameters can be modified via command-line interface.

The program can process both 2D and 3D element meshes. The basic elements of two dimensional meshes are triangles and in three dimensional cases, tetrahedra. The type of the elements doesn't affect the actual partitioning phase in any way. The only difference between processing 2D and 3D meshes is in the input- and output-routines.

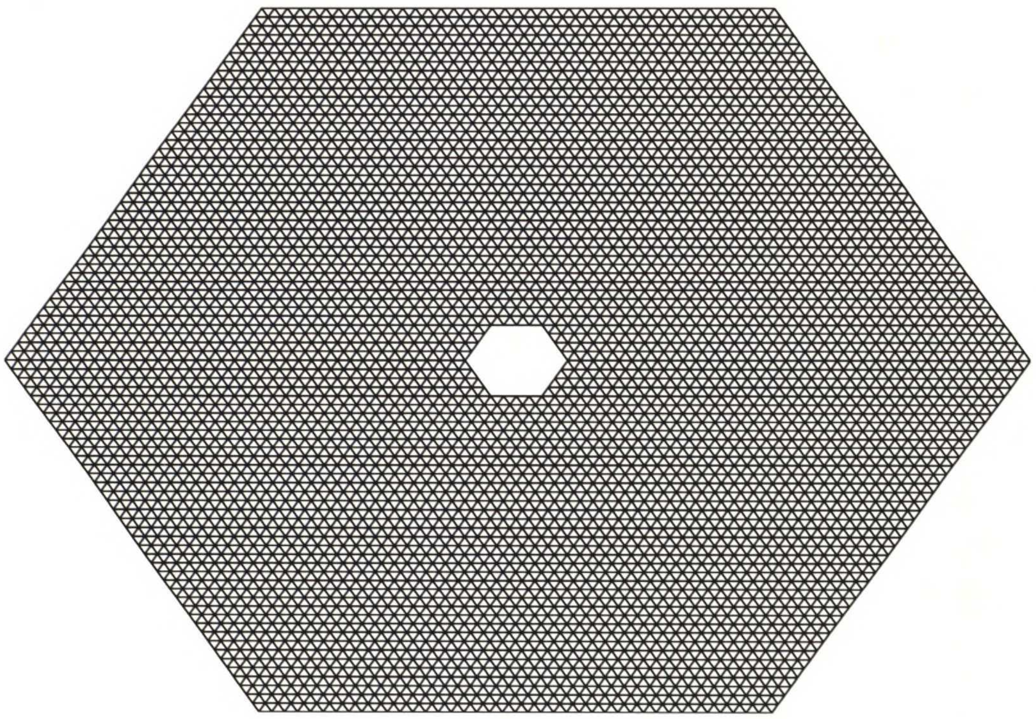
6.1.3 Graphs

All test graphs have regular structure and identical geometry. The only difference is the number of elements. Graphs of 10k, 40k, 55k, 80k and 100k vertices are used.

The use of multiple precisions of the same geometry tries to mimic the iterative nature of typical design process. In the first simulation rather coarse mesh is used to get results quickly. Model can then be altered if necessary. When the right geometry is found more precise mesh is used to get accurate results.

The geometry of test graphs is shown in Figure 6.1.

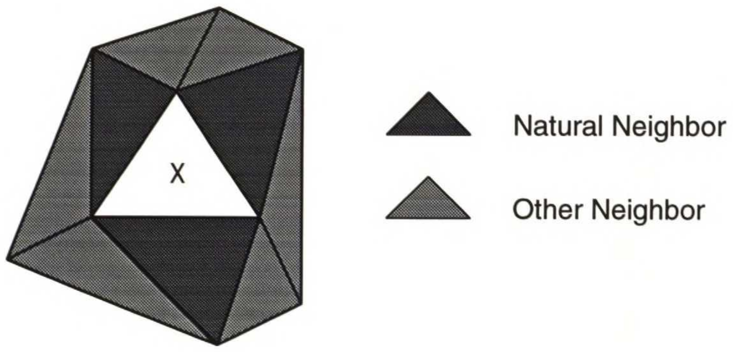
Figure 6.1 Test Graph



In addition to mesh geometry, also the way that elements are connected to their neighbors affects resulting partitions. When constructing a connection graph for a given mesh, elements were connected only to their "natural neighbors". Natural and Non-natural neighbors are defined as follows :

- Natural neighbor of a triangle is an element that shares edge with the given triangle. Natural neighbors of tetrahedra share a face with the given element.
- Non-natural neighbor is an element that shares a vertex with the given element, but is not a natural neighbor of that element.

Figure 6.2 Natural and other neighbors in 2 dimensions



Depending on application, there can be significant difference in communication required between natural and non-natural neighbors. If appropriate weights are assigned to different types of connections, partitioning algorithm can take care of these requirements and create partitions that have lower communication costs.

6.2 Performance

Three different partitioning algorithms were tested both with (solid lines) and without (dashed lines) refining. Greedy and Spectral algorithms are described in Chapter 4 and *Hybrid* is a combination of these two. The Hybdir algorithm uses Spectral Bisection to do the first bisection and after that Greedy algorithm is used. The refining algorithm used in all cases was a combination of Greedy and Kernighan-Lin. In all test cases, the same graphs were partitioned into 8 partitions, i.e. 3 recursive bisections were made.

CPU times measured here are shown only to make it possible to compare different algorithms and to give a rough estimate how long graph partitioning takes. They are not meant to be taken as ultimate performance results because only default optimization was used when compiling code and no performance analysis was made to find and/or correct possible bottlenecks in `heli` and `metis`-library.

6.2.1 Connection cost and CPU usage

The next four pairs of figures show how the connection cost and CPU usage are releated together and what are the effects of the multilevel scheme. In the first picture, Figure 6.3, graph is reduced down to 8 vertices before partitioning, i.e. partitioning algorithms are appied to a graph with 8 vertices. Similar cases with 100 and 1000 vertices are shown in Figures 6.4 and 6.5. The fourth picture, Figure 6.6, shows the situation when no coarsening is used but graphs are partitioned directly.

In the first two cases, Figures 6.3 and 6.4, where the size of the reduced graph is 8 and 100 vertices, all three algorithms produce almost identical results. Due to excessive coarsening, the actual work is done in graph reduction and refinement phases and most of the connectivity information is hidden during the partitioning phase. There are no significant difference between partitioning algoorithms when the size of the graph is small, in this case 8 or 100 vertices. However, this proves that multilevel scheme can be very effective. The results are good despite the fact that most of the partitioning work is done by reduction and refinement algorithms instead the actual partitioning algorithm.

When less coarsening is used, as in Figure 6.5, the real characteristics of the partitioning algorithms are beginning to show. The CPU usage of the Spectral Bisectioning is clearly higher than other two algorithms. This could be expected as the complexity of the Spectral Bisectioning is almost $O(N^3)$ while Greedy partitioning algorithm is only $O(N \log N)$.

The effect of multilevel scheme is shown clearly when no multilevel algorithm is used. In Figure 6.6 are the same five graphs partitioned without coarsening. There are many explanations to the rapid growth of CPU usage of the Spectral algorithm. The higher complexity is one reason, but it surely is not the only one. Running out of cache is probably another reason why Spectral Bisectioning performs so badly. The Greedy algorithm is more "cache-friendly" since it is focused on making local improvements to partitions while Spectral algorithm has a more global view of the situation. Also the version of Spectral algorithm implemented in `metis` is not very suitable for large graphs since it uses a constant tolerance in computation of the Fiedler-vector. If Spectral algorithm is used with graphs of varying size, it is recommended that tolerances are computed as functions of graph size.

Partition refining is profitable with all combinations of graph size and coarsening level. With only nominal extra CPU usage, refining can give considerable savings in connection cost. Note that refining can be used even if no coarsening is used, as in Figure 6.6.

Figure 6.3 Connection cost and CPU usage (coarsening to 8 vertices)

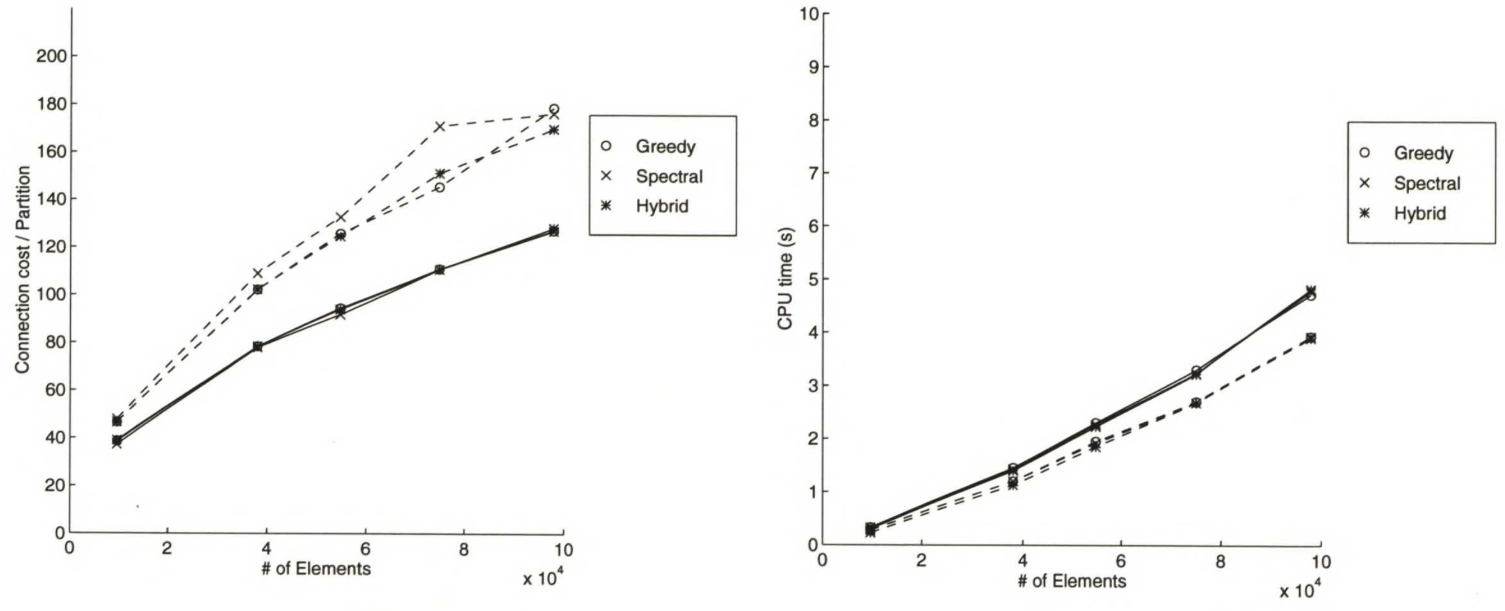


Figure 6.4 Connection cost and CPU usage (coarsening to 100 vertices)

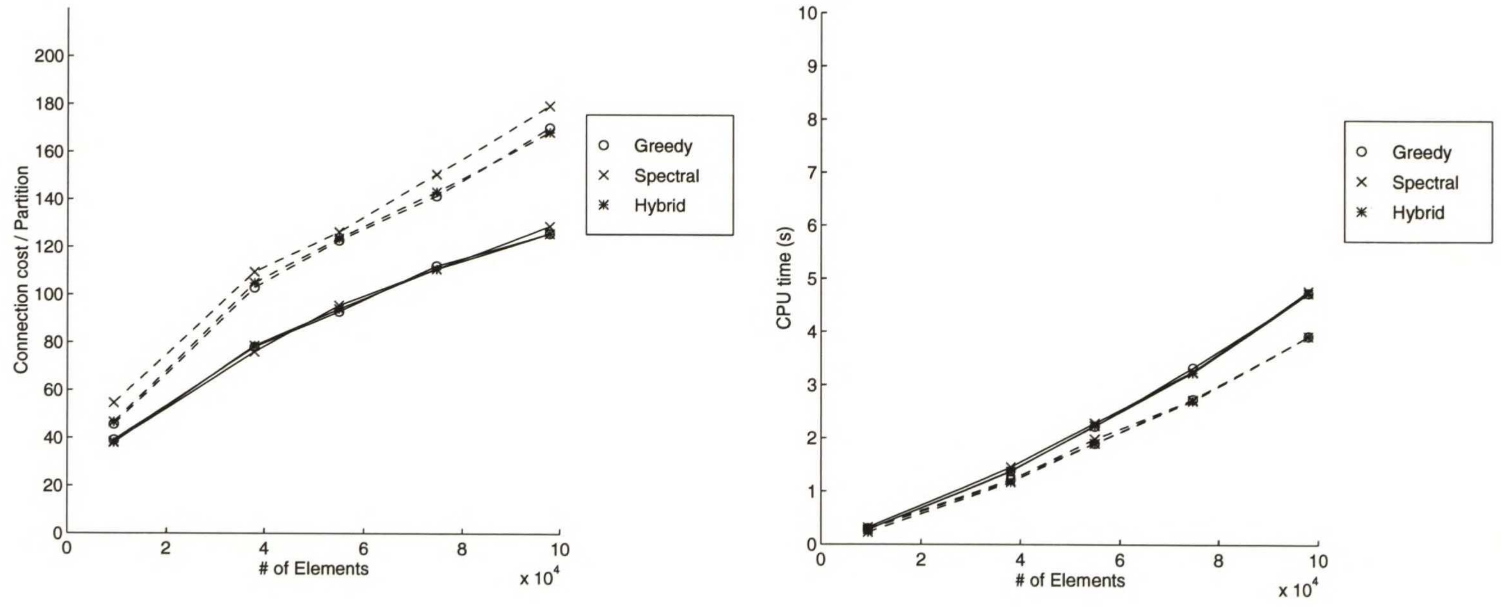


Figure 6.5 Connection cost and CPU usage (coarsening to 1000 vertices)

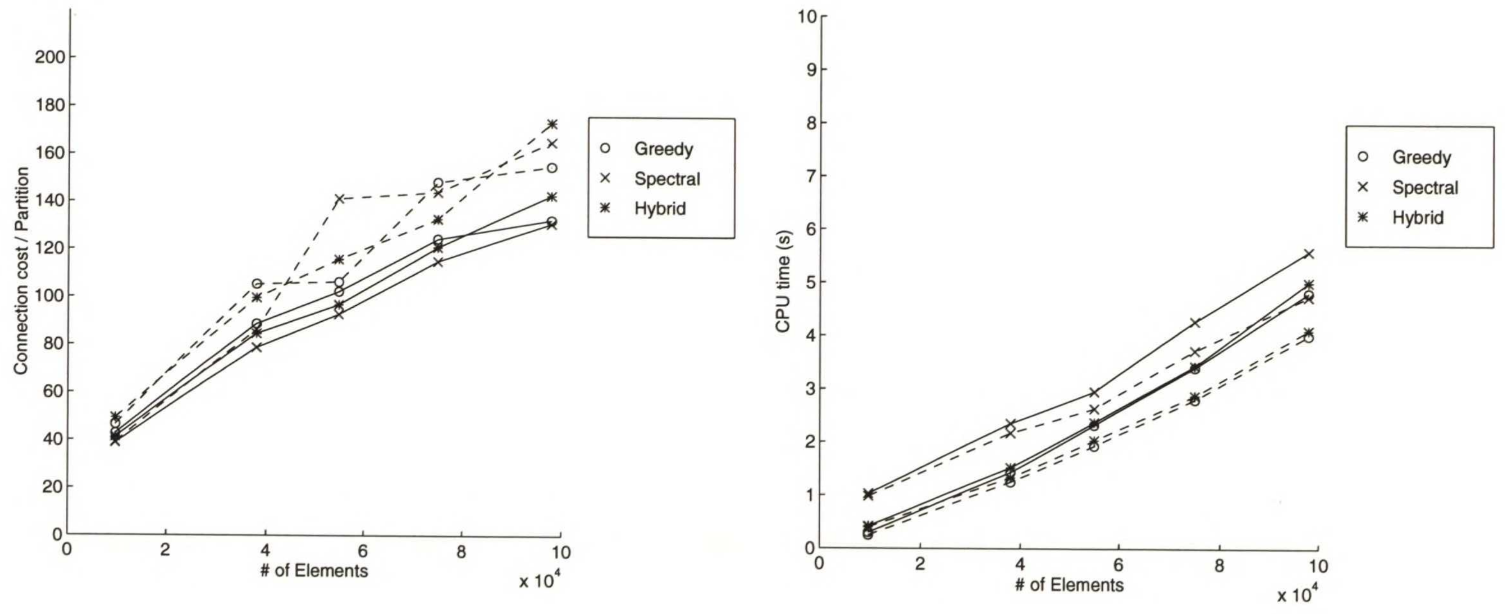
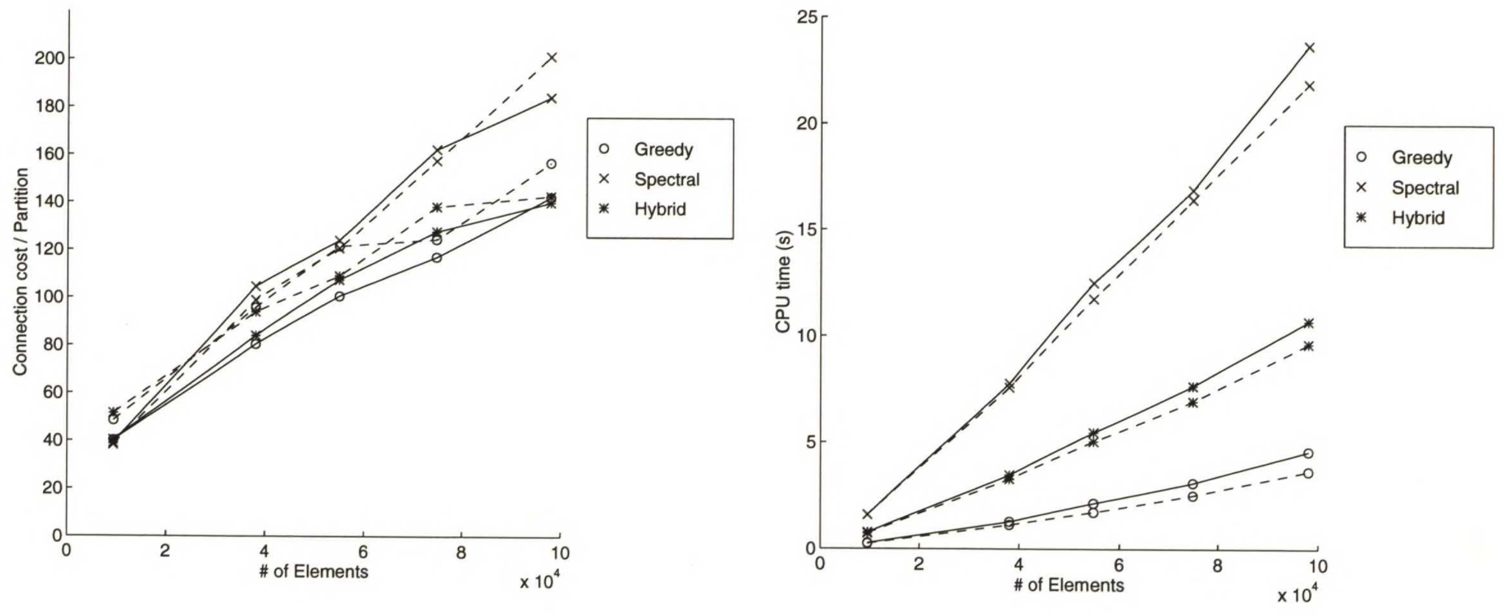


Figure 6.6 Connection cost and CPU usage (without coarsening)



6.2.2 Multilevel effects

As the multilevel scheme is clearly capable of outperforming traditional partitioning methods in both partition quality and CPU usage, it is still not clear how much graph should be reduced to get the optimal quality and/or CPU usage ? While the answer depends on how connection cost and CPU time are emphasized and what kind partitioning algorithm used, some general guidelines can be given.

In Figures 6.7 and 6.8 partition quality and CPU usage are shown as a function of the coarsening level. The graph that was used in these test is the same as in previous section (the one with 100k vertices).

As long as the size of the reduced graph is relatively small, from 100 to 1000 vectices, there is no difference in performance. However, when less coarsening is used, i.e. larger graphs are partitioned, spectral algorithm runs into problems. As said the in previous section, the rapid growth of CPU usage has multiple reasons: high complexity of algorithm, running out of cache and constant tolerance used in the Fiedler-vector computation.

A rule of thumb for multilevel partitioning is to coarsen the original graph down to few hundred vertices and use either greedy or spectral algorithm to create partitions. When the coarse partitions are projected back to the original graph some greedy refining algorithm like GKLR should be used.

Closer analysis of CPU usage per algorithm is shown in Figure 6.9 where each stacked bar shows how time is divided per recursion level, when graph is partitioned without coarsening (the rightmost points in Figure 6.8). The CPU time used in first bisection is shown on bottom of each bar and following recursion levels are stacked on top of it. It is interesting to notice, that time per recursion step is almost constant for both Greedy and Spectral algorithms, i.e. running time depends only on total size of the graph. It does not matter whether N vertices are partitioned into $2m$ partitions, each having n vertices, or into m partitions with $2n$ vertices.

Figure 6.7 Connection cost vs. Coarsening level

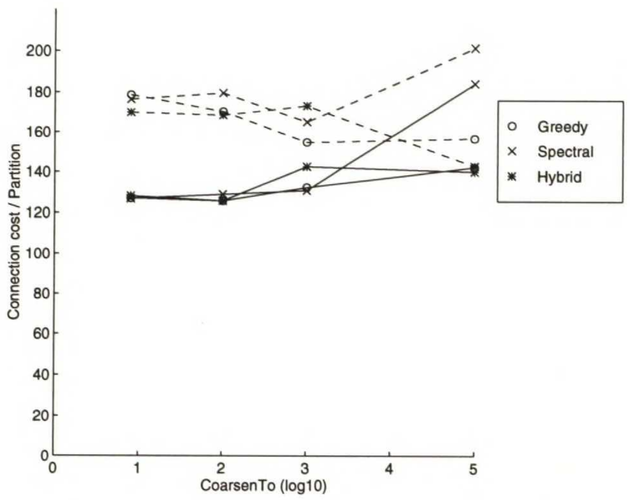


Figure 6.8 CPU time vs. Coarsening level

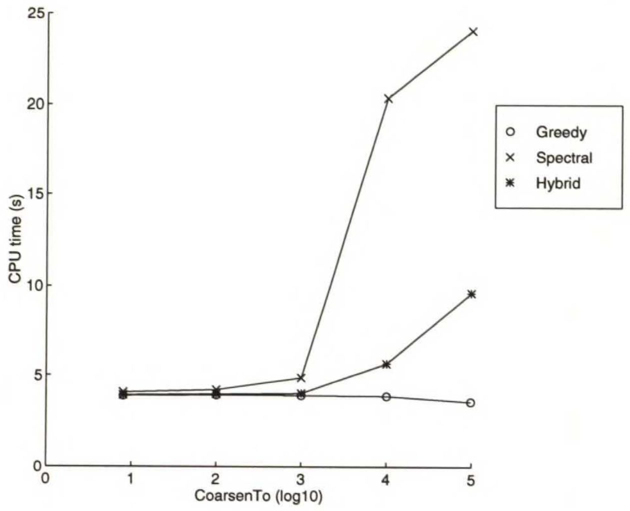
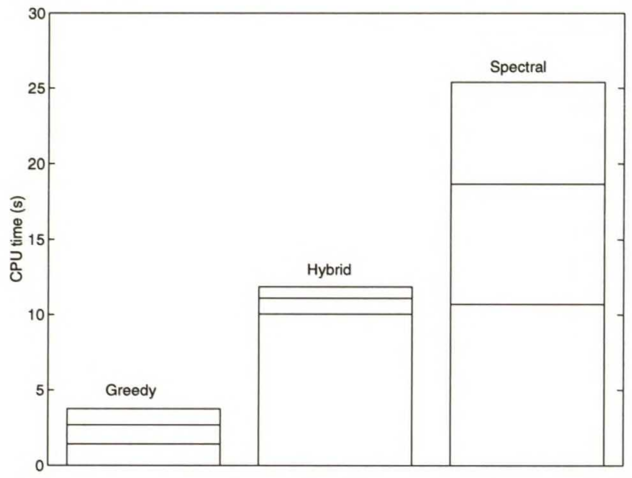


Figure 6.9 CPU usage per Recursion step



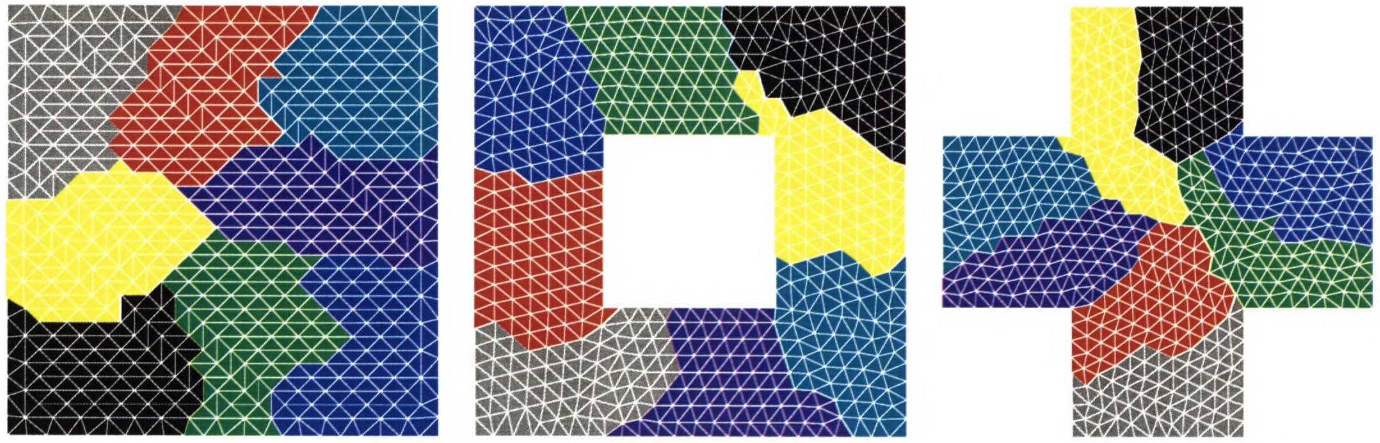
6.3 Examples

Here are three examples of partitioning 2-dimensional meshes. All meshes have about 1000 elements (vertices of connection graph) and geometries are the same that were used for testing Lanczos-algorithm in Chapter 4. However the connection structure is totally different. Graphs in Chapter 4 were built on top of a regular grid while examples here are irregular triangular meshes.

While partitions shown here are probpably not the optimal ones, they are still very good. No obvious improvements that would give considerable decrease of connection cost can be seen. These partitions were created using the default parameters of `heli`, i.e. SHEM for graph reduction, SB for partitioning and GKLR for partition refinement. The size of the reduced graph was 100 elements.

Sample partitionings of 2D and 3D meshes are shown in the Appendix A.

Figure 6.10 Examples of 2D meshes



Chapter 7

Conclusions

Graph partitioning is an NP-complete problem. At the begining there were some doubts if a partitioning of very large graphs is possible in reasonable time, so that the parallel solver could outperform its serial counterpart. Fortunately some very good heuristics exist and even more are in development. Graph partitioning algorithms can be divided into three basic groups :

- The iterative methods like Kernighan-Lin and Graph Growing improve existing partitions by making small local changes. The problem with these algorithms is that they can find only the local minimum of connection cost, i.e. the choice of the starting point is crucial. However, iterative partitioning algorithms can be very fast and they are quite simple to implement.

- Spectral methods are based on the properties of eigenvectors. Unlike iterative methods spectral bisection is done in one step and no intermediate results are available. The problem with spectral bisectioning is the complexity of algorithms. While the complexity of iterative methods is usually between $O(N \log N)$ and $O(N^2)$, the spectral methods are closer to $O(N^3)$. On the other hand, the spectral bisectioning has better global view of a graph than iterative methods do. If appropriate methods for estimating the second eigenvector of laplacian matrix can be found, spectral bisectioning could be very competitive with iterative methods.

- Multilevel method is really an extension of two previous groups rather than a totally new method. The idea is to reduce the size of the graph before partitioning. After the partitions are generated they are projected back to the original graph with refining.

Based on the current knowledge the multilevel approach is the most attractive graph partitioning strategy. Using multilevel algorithms a problem size can be reduced in time proportional to number of vertices. The actual partitioning algorithms, that typically have complexity of $O(N^2)$ or $O(N^3)$, can then generate partitions in a fraction of time that would be required for original unreduced graph. There is no significant difference in partition quality between multilevel and traditional "non-multilevel" algorithms. In some cases multilevel algorithms can actually produce better partitions. However, the efficient use of multilevel algorithms requires some understanding of their inner structure. Selected graph reduction and refinement algorithms together with a reduction level have strong influence to resulting partitions.

Graph partitioning is still an active researh area and new results are published almost monthly. Because element meshes are perhaps the most obvious source of graphs, it is not very surprising that most of the projects are dealing with Finite Element Method problems. A good example of a project almost identical to ours is the Quake project [21] [22] that is working on an earthquake simulations. However, the same heuristics can be applied to any data dependency problem, for example, parallel compilers could use graph partitioning heuristics to distribute application data among processors.

Appendix A

Examples of 2D and 3D meshes

Here, in Figures A.1 through A.4, are four examples of 2D and 3D partitionings. All partitions were generated by using `heli` with the default parameters: Sorted Heavy Edge Matching, Spectral Bisection and Greedy Kernighan-Lin Refinement. The size of the coarsened graph was 100 elements.

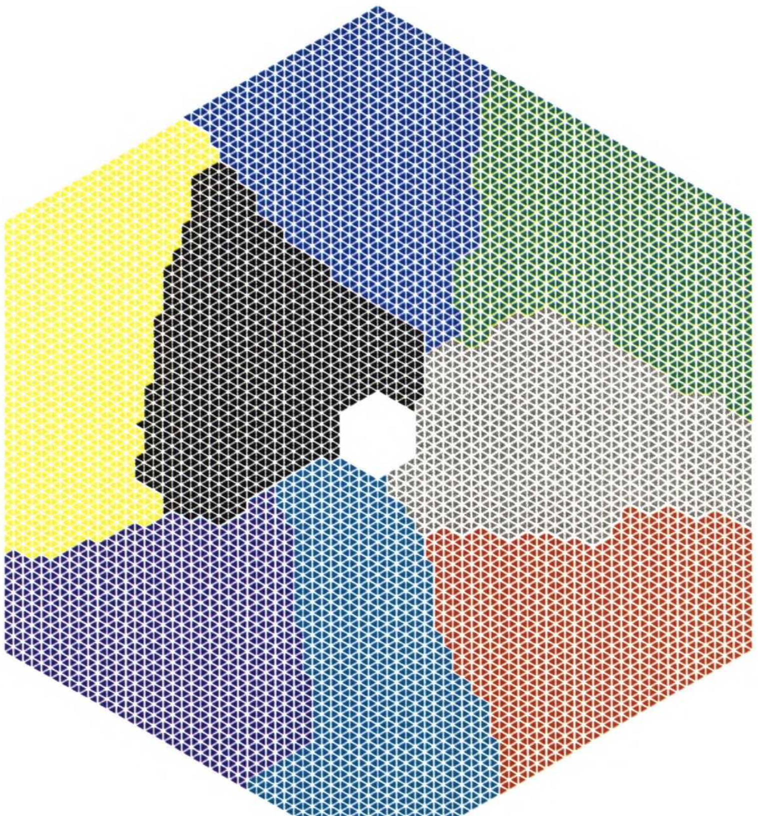


Figure A.1 'Helmholtz' -geometry

Figure A.2 Step in 2 dimension

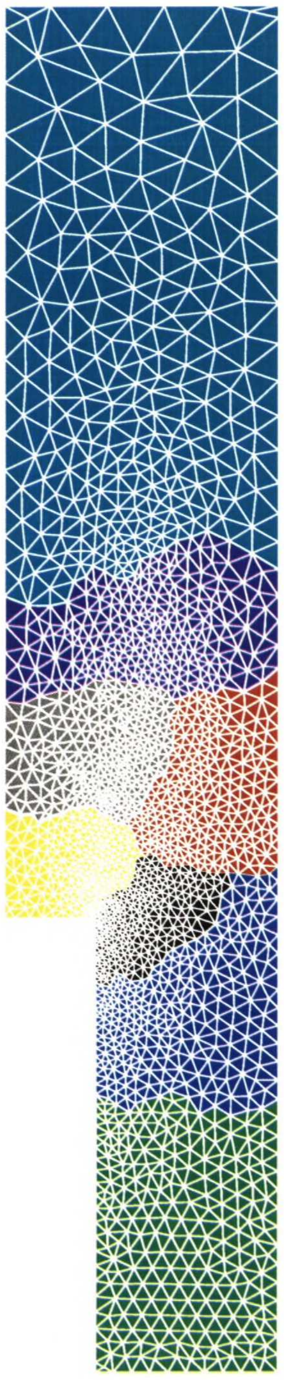
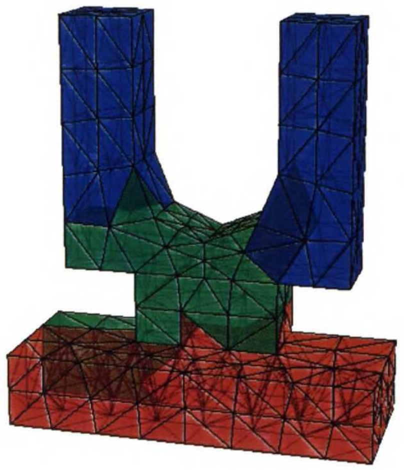


Figure A.3 'U'-shaped object


(a) Front

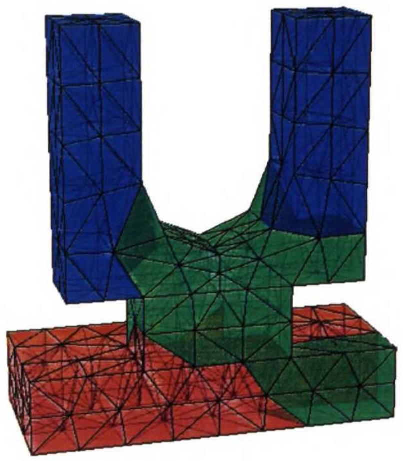
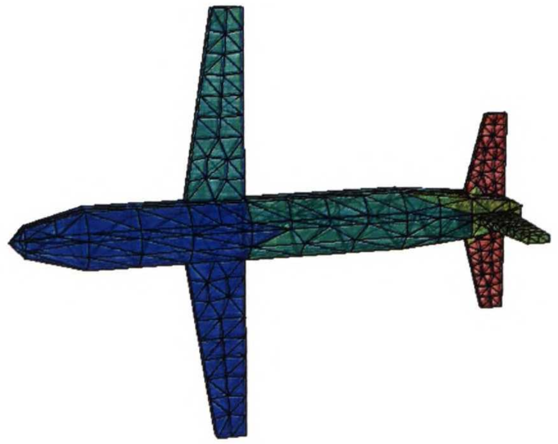
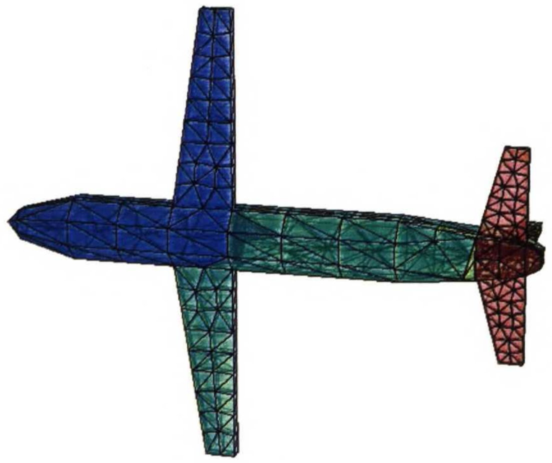

(b) Back

Figure A.4 Simple model of an airplane


(a) Top


(b) Bottom

Glossary

Connection cost
Total weigth of such edges that have their end-points in different partitions. Connection cost is usually used to measure the quality of partitions. Lower cost equals to better partitions.

Iterative partitioning method
Partitions are created step-by-step by making little improvements until the local minimum of connection cost is found. See *Direct partitioning method.*

Direct partitioning method
Partitions are based on the Fiedler-vector of the graph and they are created directly without any intermediate stages. See *Iterative partitioning method.*

Multilevel partitioning
The size of the original graph is reduced before partitioning in order to cut down CPU usage of the partitioning phase. Smaller graph is partitioned and results are projected back to the original one. See *Graph reduction* and *Partition refinement.*

Graph reduction

Reducing the size of the graph by collapsing groups of vertices together. How the groups are selected depends on a reduction algorithm. Typically tightly connected groups (cliques) are prefered.

Partition refinement

As partitions are projected back the original graph results are not always best possible. Small defects on partition boundaries can be corrected by running a few iterations of some greedy partitioning algorithm like Boundary Kernighan-Lin or Boundary Greedy. See *Multilevel partitioning.*

Connection matrix

A matrix that holds the connection information of graph. Elements of the matrix are defined as follows :

$$C(i,j) = \begin{cases} 1, & \text{if there is an edge from } i \text{ to } j. \\ 0, & \text{otherwise.} \end{cases}$$

Laplacian matrix

A close relative of the connection matrix. The second eigenvector of laplacian matrix, so-called Fiedler-vector, is used in direct partitioning methods. See *Direct partitioning method* and *Fiedler-vector.*

$$L(i,j) = \begin{cases} \deg(i), & \text{if } i = j. \\ -1, & \text{if } i \neq j \text{ and } C(i,j) \neq 0. \\ 0, & \text{otherwise.} \end{cases}$$

Fiedler-vector

The second eigenvector of the laplacian matrix associated with the graph. See *Direct partitioning method.*

Bibliography

[1] J. Järvinen. *Virtauslaskentaohjelmiston kehittäminen, projektisuunnitelma.* CSC-Tieteellinen laskenta Oy, 1995

[2] B.W. Kernighan, S. Lin. *An Efficient Heuristic Procedure for Partitioning Graphs.* The Bell System Technical Journal, pp. 291-307, 1970

[3] H.D. Simon. *Partitioning of Unstructured Problems for Parallel Processing.* NASA Ames Research Center, 1994

[4] L.R. Ford, D.R. Fulkerson. *Flows in Networks.* Princeton University Press, 1962

[5] G. Karypis, V. Kumar. *A Fast and High Quality Multilevel Scheme for Partitioning Irregular Graphs.* Technical Report TR 95-035, University of Minnesota, 1995

[6] G. Karypis, V. Kumar. *Analysis of Multilevel Graph Partitioning.* Technical Report TR 95-037, University of Minnesota, 1995

[7] G. Karypis, V. Kumar. *Multilevel k-way Partitioning Scheme for Irregular Graphs.* Technical Report TR 95-064, University of Minnesota, 1995

[8] B. Hendrickson, R. Leland. *Multidimensional Spectral Load Balancing.* Technical Report SAND93-0074, Sandia National Laboratories, 1993

[9] B. Hendrickson, R. Leland. *A Multilevel Algorithm for Partitioning Graphs.* Technical Report SAND93-1301, Sandia National Laboratories, 1993

[10] C.M. Fiduccia, R.M. Mattheyses. *A Linear-Time Heuristic for Improving Network Partitions.* Proc. 19th IEEE Design Automation Conference, pp. 175-181, 1982

[11] C.H. Papadimitriou, K. Steiglitz. *Combinatorial Optimization: Algorithms and Complexity.* Prentice Hall, 1982, pp. 467-468.

[12] B. Mohar. *The Laplacian Spectrum of Graphs.* Dept. of Matchematics, University of Ljubljana, Yugoslavia, 1988

[13] *Applications of Parallel Computers -course homepage.* `http://www.icsi.berkeley.edu/cs267/` University of California, Berkeley.

[14] M. Fiedler. *Algebraic Connectivity of Graphs.* Czech. Math Journal, vol.23, pp. 298-305, 1973

[15] M. Fiedler. *A Property of Eigenvectors of Nonnegative Symmetric Matrices and its applications to Graph Theory.* Czech. Math Journal, vol.25, pp. 619-637, 1975

[16] B.N. Parlett, H. Simon, L.M. Stringer. *On Estimating the Largest Eigenvalue With the Lanczos Algorithm.* Mathematics of Computation, vol.38, no.157, pp. 153-165, 1982.

[17] Roger G. Grimes, John G. Lewis, Horst D. Simon. *A Shifted Block Lanczos Algorithm for Solving sparse Symmetric Generalized Eigenproblems.* SIAM J. Matrix Anal. Appl., vol.15, no.1, 1994

[18] G. Golub, van Loan. *Matrix Computations, 2.ed,* pp. 437-438, 1989

[19] G. Golub, van Loan. *Matrix Computations, 2.ed*, pp. 383-385, 1989

[20] G. Karypis, V. Kumar. *Metis - Unstructured Graph Partitioning and Sparse Matrix Ordering System.* Department of Computer Science, University of Minnesota, 1995

[21] *The Quake Project -homepage.* `http://www.cs.cmu.edu/%7Equake/quake.html` School of Computer Science, Carnegie Mellon University.

[22] *Archimedes -homepage.* `http://www.cs.cmu.edu/%7Equake/archimedes.html` School of Computer Science, Carnegie Mellon University.