SEQUENTIAL AND DISTRIBUTED ALGORITHMS FOR FAST GRAPH PARTITIONING

by

Md. Shahadatullah Khan
B.Sc.(Eng.), Bangladesh University of Engineering & Technology 1992

A Thesis Submitted in Partial Fulfillment of the Requirements
for the Degree of

Master Of Applied Science

in the Department of Electrical and Computer Engineering

We accept this thesis as conforming
to the required standard

Dr. K.F. Li, Supervisor (Dept. of Elec. & Comp. Eng.)

Dr. N.J. Dimopoulos, Departmental Member (Dept. of Elec. & Comp. Eng.)

Dr. Z. Dong, Outside Member (Dept. of Mech. Eng.)

Dr. H.A. Müller, External Examiner (Dept. of Comp. Science)

Supervisor: Dr. Kin F. Li

Abstract

In this thesis, we deal with the following k-way graph partitioning (GP) problem: given an undirected weighted graph $G(V, E)$, partition the nodes of G into k parts of almost equal size such that the partition-cost (sum of the weights on edges with nodes in different parts) is minimized. We propose some simple and fast algorithms for this problem for both sequential and distributed computing environments.

We give three main algorithms for graph partitioning: direct algorithm AUCTION; and iterative algorithms GREEDYPASS and GREEDYCYCLE. In the algorithm AUCTION, we introduce the idea of using auction and biddings for the GP problem. This is an inherently distributed algorithm. To the depth of our knowledge this is the first distributed algorithm for this problem. The algorithm GREEDYPASS is a greedy iterative algorithm. In each iteration we send a node from the current part to another part in order to get maximum decrease in partition-cost, and iteration can continue taking the destination as the next current part. Our third algorithm GREEDYCYCLE is another greedy algorithm where we introduce the idea of cyclic node passing among parts during the iterative improvement stage. In every iteration: at first the parts compute their node passing interests, and then the algorithm finds and processes the cycles of node passing interests. Cyclic node passing is a k-way generalization of the 2-way node exchange found in the Kernighan-Lin approach.

We present sequential algorithms for AUCTION, GREEDYPASS and GREEDYCYCLE, distributed algorithms for AUCTION and GREEDYCYCLE, and some of their combinations. We have implemented the sequential algorithms in the Unix environment, and the distributed algorithms in the PVM environment. We compared the experimental performances with Lee's algorithm, and the results show that our algorithms are extremely fast and they produce reasonable quality of solutions.

Examiners:

Dr. K.F. Li, Supervisor (Dept. of Elec. & Comp. Eng.)

Dr. N.J. Dimopoulos, Departmental Member (Dept. of Elec. & Comp. Eng.)

Dr. Z. Dong, Outside Member (Dept. of Mech. Eng.)

Dr. H.A. Müller, External Examiner (Dept. of Comp. Science)

Table of Contents

List of Figures

List of Tables

Acknowledgments

At first I would like to express my heartiest appreciation to my supervisor Dr. Kin Li for his encouragement, moral confidence, and active cooperation during this whole research.

The unending love, inspiration and confidence of my family members and friends provided the base for this research. I owe to all of them for this, and they deserve special thanks. I want to specially thank Mohsin whose letters always kept me in touch of all the developments with friends and families in Dhaka.

All the fellow researchers in the lab were very cooperative. In particular many discussions with Vassilios, Mahmood, and Andrew played an important role for the progress of this work. This is my opportunity to thank all of them.

Special thanks goes to Nargis Vabi, Nahar Vabi and Erika for their delicious cuisine, and also to Siddiquee Vai, Mahmood Vai, Ali Vai, Mahboob Vai, and Moinul for the addas.

To my mother, the source of all my light.

Chapter 1

Introduction

Dividing a graph into sets of nodes with the aim of optimizing certain cost-functions constitute an important combinatorial problem of interest to both theoreticians and practitioners. It has got many diverse applications, such as VLSI circuit partitioning and layout [1, 2, 3, 4, 5], multiprocessor task allocation [6, 7, 8, 9, 10, 11, 12], distributed divide-and-conquer algorithms [13], matrix factorization [14], network simulation, and manufacture planning [15]. In this thesis, we propose some algorithms for the k-way graph partitioning (GP) problem for both sequential and distributed computing environments.

1.1 The Problem

Generally speaking, in graph partitioning problems, we are given a weighted graph $G(V, E)$ as input, and the problem is to divide the graph into some disconnected components optimizing some given function under certain given constraints. As this description reveals, we see that the problem may have many formulations, with different choices of constraints and optimization functions.

For this work, we use the following formulation of the k-way graph partitioning problem.

k-way GP problem:

Input:

- Undirected graph $G(V, E)$ of n nodes, and m edges with an edge cost function $c : E \rightarrow \mathcal{N}$.
- The required number of partitions $k \geq 2$, a positive integer .

Output:

- Node disjoint partition $\Pi = (V_1, V_2, \ldots, V_k)$ of V minimizing the objective function under the following constraint.

Constraint:

- $\forall i, j : \qquad |V_i| \simeq |V_j|$.

Objective function:

- $c(\Pi) = \frac{1}{2} \sum_{i=1}^{k} \sum_{e \in E_{ext,i}} c(e)$,
  where $E_{ext,i} = \{e \in E \mid e \cap V_i \neq \emptyset \mid e \setminus V_i \neq \emptyset\}$ is the set of external edges of part V_i. Here an edge $e = (v_1, v_2)$ is considered a set of two nodes.

The node disjoint partition $\Pi = (V_1, V_2, \ldots, V_k)$ of V is referred to as a k-way *cut* or *partition* of the graph. Each subset of nodes V_i in the partition is referred to as the *part* number i, and the number of nodes in that part $|V_i|$, is called its *size*. The constraint is imposed in the problem to achieve a *balanced* partition, where each part gets almost an equal number of nodes. This requirement comes from many practical applications, for example, for task allocation on similar processors, or for divide-and-conquer type of algorithms. The cost $c(\Pi)$ is called the *cutsize* of the cut. The edges that connect different parts of the cut are called *cross-edges* and are said to *contribute to this cut.* Thus the cutsize is the weighted sum of the external connections of a part, summed over all parts. The factor $\frac{1}{2}$ in front of the cutsize is given so that the cost contributed by each external edge is counted once, but not twice (once for each end part).

We note that the meaning of the cost of an edge is flexible, which is a definite plus for its application. For example, for VLSI layout the edges may be a function of wire length, its importance etc. A wire whose delay is very critical for the performance of the whole design is not advisable to be cut, and this can be taken care of by putting higher weight on such an edge.

1.2 Overview

The rest of the thesis is organized as follows.

Chapter 2 discusses some background materials. It introduces the terminologies, traces the major developments of the GP algorithms, explains our motivations, and introduces the distributed platform used in this work.

Chapter 3 presents our AUCTION algorithm. It introduces the basic algorithm and discusses some implementation issues.

Chapter 4 presents our iterative algorithms GREEDYPASS and GREEDYCYCLE.

It describes these algorithms in detail along with some of its properties and implementation issues.

Chapter 5 presents our experiments and performance results. It describes our implementations, and presents, compares and explains the performance results.

Finally, chapter 6 concludes the thesis, pointing to some of the contributions and future work.

Chapter 2

Background

2.1 Terminology

In this section, we will give brief definitions for some of the terms we will need in this thesis.

Algorithms and Complexity

An *algorithm* is any well-defined computational procedure that takes some set of values, as *input* and produces some values, as output. An algorithm is thus a sequence of computational steps that transform the input into the output [16]. We can also view an algorithm as a tool for solving a well-specified computational problem. The statement of the *computational problem* specifies in general terms the desired input/output relationship; and an algorithm describes a specific computational procedure for achieving that input-output relationship. If the inputs (satisfying whatever constraints are imposed in the problem statement) needed to compute a solution are all specified, it is called an *instance* of the problem.

Analyzing an algorithm means predicting the resources it requires. The two most important resources an algorithm requires are the running time and memory. The *running* or *execution time* of an algorithm on a particular input is the number of primitive operations or *steps* executed. The notion of step is convenient to make the algorithm analysis machine-independent. The *memory* requirement of an algorithm means how much memory the algorithm requires to represent and solve the problem. For the analysis of distributed algorithms, another important issue is how many messages the algorithm needs to exchange, and how much information. This is called the algorithm's *communication* requirement. In this thesis, we are mainly interested to find the maximum resource requirement on *any* input of a particular size. This is called the *worst-case analysis*. The other alternative here would be the *average-case analysis* where one computes the average-resource requirement of an algorithm. Let us assume that, for a problem of size n, the worst-case running time of an algorithm is $an^2 + bn + c$ for some constants a, b and c. For large n, the most dominating term of this expression will be an^2, and therefore we say that this algorithm has a time complexity of order n^2. Thus the *order* of an algorithm is defined as the growth rate of the leading or dominant term in the worst-case resource requirement of an algorithm. Let the worst-case resource requirement of an algorithm for a problem instance of size n be given by $f(n)$. We say that this algorithm is of order at most $g(n)$, or symbolically $f(n) = \mathcal{O}(g(n))$, if there exist positive constants c_0 and n_0 such that, $0 \leq f(n) \leq c_0 g(n)$ for all $n \geq n_0$.

Graphs

A *graph* is defined as a set of *vertices* (or *nodes*) V interconnected by a set of *edges* E, symbolized as $G = (V, E)$ [17]. If the edges have no direction, it is called *undirected*, otherwise it is called *directed*. If the edges have weights, the graph is called *weighted*, otherwise it is *unweighted*. Since, in this work, we deal only with weighted undirected graphs, the term 'graph' will imply a weighted undirected one.

An edge e that connects vertices v_i and v_j is written as $e = (v_i, v_j)$, and is said to be *incident with* v_i and v_j. If $(v_i, v_j) \in E$, then v_i and v_j are *adjacent to* each other, or alternatively they are called *neighbors*. The *degree* of a node v, denoted by $d(v)$, is the number of neighbors v has.

Optimization Problems

In an *optimization problem* one seeks to maximize or minimize a specific function, called the *objective* [18]. Optimization problems can be broadly classified into mathematical programs and combinatorial optimization. A *mathematical program* is an optimization problem where both the problem and the objective are given as mathematical functions or functional relations. On the other hand, a *combinatorial optimization* problem is usually given by a finite *input set*, a *legal configuration condition* and an *objective function*. The legal configuration condition says how the elements of the input set can be arranged in different permutations to generate the *set of solutions*. Now the optimization problem is to find one such solution which will maximize or minimize the objective.

In this thesis, we deal with the graph partitioning problem which falls into the combinatorial optimization problem category. Here the graph $G(V, E)$ specifies the input set, and the legal configuration condition requires that the nodes of the graph should be divided into k parts of almost equal size. Partition-cost is the objective function. All the permutations of nodes in k parts that obey the legal configuration condition comprise the set of solutions. Now the optimization problem is to find a solution from the solution set which will give the minimum value of the objective function (i.e. the partition-cost). We call such a solution *an* optimal solution to the problem, as opposed to *the* optimal solution, since there may be several solutions that achieve the optimal value.

One way to solve the combinatorial optimization problems is *iterative improvement* or *hill climbing*. Here the algorithm starts with a solution, evaluates the solutions which can be obtained from the current solution by some small rearrangements, and if it finds a solution with a better value for the objective function the iteration starts again with the new solution as the current solution. If in a certain iteration the algorithm does not find any improvement in objective function, it is called a *local optimum*, and the algorithm terminates. Thus a local optimum implies a better value of the objective function compared to the solutions reachable from it in one step. We note that, the hill-climbing algorithm has the risk of being stuck at a poor local optimum value rather than achieving a *global optimum* which will be optimum compared to all other solutions. One of the most popular ways to improve the solution is *simulated annealing*. This method also tries to iteratively improve the objective function, but here the algorithm permits some occasional move to solutions against the objective function improvement.

Distributed Systems

A *distributed system* can be defined as a collection of autonomous computers interconnected by a communication network to achieve an overall goal [19]. For example, a collection of workstations connected by a local area network (LAN) and running a distributed operating system is a distributed system. Usually the computers in a distributed system do not have any shared memory, and all communication is via message-passing among processes (A process is a program in execution). Since there is no common time reference among the processes, a distributed system is inherently *asynchronous*. Some of the promising benefits of distributed systems include resource sharing, high system performance, system heterogeneity, flexibility, incremental growth, and better reliability.

A *distributed algorithm* is an algorithm designed for a distributed system. A distributed algorithm usually proceeds with multiple processes running in parallel on different processors. If the processes of a distributed algorithm run in specified phases of computations, and communicate after each such phase, then the algorithm is called *synchronous*. Whereas if the processes can continue computation without any necessity of communication among them at prespecified points of time then this algorithm is called *asynchronous*. To implement synchronous algorithms in an inherently asynchronous distributed system, we need some communication mechanisms called *synchronization* mechanisms.

2.2 Pseudocode and Notations

For the presentation of our algorithms we used a C-style pseudocode system for the following reasons: C is a small structured language supporting many useful operators; it is popular; and it is also the language for our implementation. We use the following structures in our algorithms with their usual meaning.

- declaration identifiers (**int** and **procedure**),
- operators (++, !, =, >, <, %, {, }, (,)),
- conditional structures (**if, else, case - switch**), and
- loop structures (**for, while, do - while, break** and **continue**).

In our algorithms, when we come across the symbol '//', the rest of that line is considered to be a comment. The use of pseudocode permits us to blend text with code, which is very helpful for expressing algorithms. To keep the algorithm presentation clean, we did not use many comments inside the algorithms.

However for ease of understanding the algorithms, we always supplemented it with helpful descriptions within the text.

In this thesis, we frequently used the following symbols, operators and notations.

$G(V, E)$	Graph with node set V and edge set E
V	Set of nodes
E	Set of edges
d	Maximum node degree of the graph
n	Number of nodes in V
m	Number of edges in E
k	Number of parts graph G will be divided into
$\mathcal{N}$	Set of natural numbers
$a \in A$	element a is an element of set A
$A \cup B$	Union of sets A and B
$A \cap B$	Intersection of sets A and B
$A \setminus B$	Subtraction of set B from A
$\|A\|$	Number of elements in set A
$a \bmod b$	Remainder when a is divided by b
$a \,\%\, b$	Remainder when a is divided by b
$!\, a$	Not a
$\forall\, i$	For all i
$\mathcal{O}(g(n))$	function $f(n)$ where there exists positive constants c_0 and n_0 such that, $0 \leq f(n) \leq c_0 g(n)$ for all $n \geq n_0$
$\lceil x \rceil$	Smallest integer greater than x
$\lfloor x \rfloor$	Largest integer smaller than x

2.3 Related Work

In this section, we briefly trace some of the major developments to solve the GP problem. An exact solution of the GP problem is known to be NP-Hard even for $k = 2$ [20, 21], that is, it is not expected to be solved optimally in polynomial time. However, it is too important a problem to be abandoned merely because it is intractable. There has been intensive work done to find some approximate solutions of the GP problem using heuristics. To start talking about some of these approximate solutions, we need some sort of metric to characterize and compare these solutions. The first question that comes to mind is how close these solutions are to the optimal solution. One good measure could be the *approximation ratio*, defined as the ratio of the cutsize given by the algorithm to the optimal cutsize. But practically, this measure is not useful because we do not yet know the optimal cutsize. For this reason, to characterize the *quality of solution* of an algorithm in this work, we just compare the cutsize given by this algorithm with cutsizes given by other algorithms. A lower cutsize means a better quality.

Direct and Iterative Algorithms

The GP problem may be approached in two ways: (a) We can try some *direct* or *constructive* way of solution where we construct the solution directly using some form of computation on the input information. For this computation, we can use some form of heuristics to get an approximate solution. As the search space grows very fast with the input size, it is very difficult to get this kind of solution, and the quality of solution is not expected to be very satisfactory. However, in general, this method gives fast solutions. (b) We can use *iterative improvement* techniques, where we start with some initial partition, and then iterate again and again in order to improve the objective function. Almost all of the existing algorithms for the GP problem are based on this approach, and the quality of solution expected from these algorithms is in general much better than by the first technique. However, the cost we pay for it is a higher time complexity. As a matter of fact, the iterative improvement method may also use the direct approximation method to get its initial partition before the iterations start.

The few constructive algorithms found in the literature on the GP problem are mainly based on graph theory, mathematical programming or clustering. As an application to the distributed task allocation problem, Stone and Bokhari [11] had done elaborate studies on the graph bipartition (2-way partition) problem using the max-flow min-cut flow technique. Later Lo [8] extended this work for k-processor partitioning by repeated use of the max-flow min-cut technique.

Most of the popular GP algorithms are based on the iterative improvement strategy. In 1970, Kernighan and Lin [2] proposed a heuristic procedure (the KL algorithm) for graph partitioning which became the basis for most of the iterative improvement algorithms generally used. This algorithm deals with the partitioning of a graph $G(V, E)$ where $|V| = n$ is even, into 2 parts of equal size. The basic idea is a natural search method: start with an initial bisection, and continue exchanging nodes across the cut to improve the cutsize. The algorithm consists of a series of *passes*. Each pass starts with all the nodes as *unmarked*. In each pass the algorithm iterates $n/2$ times. In each iteration, (a) a pair of nodes is chosen, one from each part, which would give a maximum decrease (or a minimum increase when no decrease is possible) in cutsize from among the nodes which are not yet *marked* for exchange in the current pass, (b) the chosen pair of nodes is colored as *marked* for this pass, and (c) finally the expected cost considering the so-far marked exchanges is calculated and saved. The $n/2$ costs produced during the pass are examined, and the one with the smallest cost is chosen as the starting partition for the next pass. Passes are performed until no improvements can be obtained.

Several works have been done for improved heuristic algorithms based on the basic idea of the KL method. Fiduccia and Mattheyses [4] suggested a fast heuristic for improving partitions by using efficient data structures. Here a single node is moved across the cut in an iteration, which extends the flexibility of the algorithm to handle unbalanced partitions, nonuniform vertex weights, and hyper-edges (edges connecting two or more nodes, nets in VLSI terms). Later Krisnamurthy [22] again improved the method by using more sophisticated heuristics.

Another line of development in this field was due to the introduction of simulated annealing (SA[1] method) by Kirkpatrick *et al.* [3] in 1983. SA is a randomized local search strategy using probabilistic hill-climbing. As applied to GP problem, the SA method will iterate among passes to decrease the cutsize like the KL method, but unlike KL, it will occasionally allow increases in cutsize between passes. The main objective to allow occasional uphill moves is to improve the probability that the algorithm is not stuck at a poor local minimum. However the cost we pay here is extra running time. Johnson *et al.* [23] had done detailed study and experiments with SA, and compared it with the KL method. According to their results, the time for one run of KL from an initial partition is much smaller than that for SA. However, KL generally gives a partition-cost that is about 10% larger than those provided by SA. On the other hand, if the running times of both are equalized by making many more runs of KL with randomly chosen initial partitions and then choosing the best bisection results of all these runs, KL gives a partition-cost typically within a few percent of SA's best partition-cost, although SA continues to provide better results.

[1]The abbreviation SA in this section refer to 'Simulated Annealing' although in the rest of the thesis it will refer to 'Sequential Auction'. However this use should be clear by its context.

k-way Partitioning: Hierarchical and Nonhierarchical Algorithms

So far, we have discussed the graph bisection problem which is a special case of the k-way GP problem for $k = 2$. For the general problem, with $k \geq 2$, two classes of algorithms are identified in the literature — hierarchical and nonhierarchical. The *hierarchical* method is applicable only for $k = 2^r$ where $r \in \mathcal{N}$. In this method, the graph bisection algorithm is applied iteratively as follows: (a) divide the graph into two parts, and (b) apply (a) on both parts until we have 2^r parts. Thus the computation follows a tree structured computation from higher level to lower levels. This method was first mentioned in [2]. It has at least two problems: it cannot be applied for any arbitrary value of k, and a poor partition at a higher level is propagated to lower levels. One advantage of this algorithm is that it is easily parallelizable, all the computations at the same level of the partition tree can proceed in parallel without affecting one another. Gilbert *et al.* [14] gives a hierarchical k-way GP algorithm, and its implementation on hypercube multiprocessors.

On the other hand, a simple strategy for *nonhierarchical* partitioning, may be as follows. Start with any initial k-way partition, and then iterate on exchanging pairs of nodes among parts if that reduces the cutsize. Now the problem at any iteration is more complex than [2], we have to first choose two parts, and then choose the nodes to be exchanged. The algorithm will be definitely more complex than simple KL method, and complex data structures are required for efficient implementation. An idea very similar to this was actually used by Sanchis in [24]. For the iterative improvement computation, she uses a vector of gains extending the idea of gain in [22]. She also extends the data structure of [4] for efficient implementation. Recently Lee *et al.* [25] also described another nonhierarchical algorithm for k-way partitioning. Actually, we have used this algorithm throughout our studies as a standard to compare the performance of our algorithms. For this reason, we will put more light on this algorithm in the following section.

Parallel Graph Partitioning

In recent years, there has been some interest in developing solutions to the GP problem in parallel environments. Gilbert *et al.* [14] has proposed a parallel hierarchical algorithm for a message-passing multiprocessor system based on the basic KL-method. In this algorithm parallel computations of the KL-method proceed on different processors in a tree-structured fashion. Also parallel SA method has been proposed in [26] and has been implemented on a Connection machine by Wong *et al.* [27], and on an Intel Hypercube by Banerjee *et al.* [28]. In [29], Savage and Wloka have shown that parallelizing the KL-method is P-complete, and parallelizing SA is P-hard. For this reason, they have proposed another new local search based heuristic (the mob heuristic) which they have also implemented on a Connection machine, and reported good speedup with a little compromise in partition quality. The parallel algorithms we have just pointed out are all based on 2-way partitioning, and they can deal with k-way partition only with a hierarchical approach.

The first nonhierarchical parallel algorithm for the k-way GP problem was proposed by Isomoto *et al.* in 1993 [30]. They have proposed an iterative algorithm using a total of $k/2$ processors in a shared bus shared memory multiprocessor system. Each iteration starts with $k/2$ subgraphs, one at each processor. At the start of the iteration, each processor bisects its subgraph with the intention of minimizing the internal bisection-cost, which now makes a total of k subgraphs. In the next step these subgraphs are again paired with the intention of minimizing the external cost so that there is one pair of subgraphs in every processor. A new iteration can now start again. The iterations can be continued until the solution converges, or up to a predetermined number of iterations. We note that for this method, k has to be even.

2.4 Lee's Algorithm

In 1990, Lee *et al.* [25] gave a new and efficient nonhierarchical algorithm for the general k-way partitioning. In the performance studies for our algorithm, we have used this algorithm as a benchmark. Here are some of the reasons for our choice:

- It is one of the most recent algorithms for the general k-way graph partitioning.
- In their performance studies the authors have shown that this algorithm outperforms the hierarchical KL algorithm both in cutsize and elapsed time.
- The main idea of the algorithm is new and simple.
- It uses a simple data-structure which is easy to understand and implement.

According to [25], the GP problem has to take care of two things: the balancing constraint and the minimization of the partition cost. They solved these two problems in the following two steps:

1. transform the problem into another maximization problem with no constraint, and then
2. solve the latter problem.

For step 1, they have proposed a simple transformation. Let $G = (V, E)$ with $|V| = n$ be the problem graph which has to be partitioned into k subgraphs. They used the adjacency matrix $c[n][n]$ to represent the graph, where $\forall i, j : 1 \leq i, j \leq n$, if $(i, j) \in E$, $c[i][j]$ is the cost of edge (i, j), else $c[i][j]$ is zero. Let us consider another graph $G' = (V, E')$ represented by matrix $c'[n][n]$ obtained from the following transformation: $\forall i, j : 1 \leq i, j \leq n, c'[i][j] = R - c[i][j]$, where R is a constant greater than $\sum_{i=1}^{i=n} \sum_{j=i}^{j=n} c[i][j]$. In their paper, the authors have shown that the k-way partition of G' to maximize the partition-cost without any constraint also corresponds to the original GP problem with the balance constraint.

To find a k-way partition of G' to maximize the partition-cost in step 2, they have proposed a simple iterative algorithm similar to the KL-method. The algorithm is given in figure 2.1. The basic approach is to start with an arbitrary partition and to improve it by iteratively choosing one node from one part and moving it to another. The node to be moved is chosen so that a maximum increase in partition-cost may be obtained (or minimum decrease if no increase is possible).

Before we discuss the algorithm, let us introduce some of the data-structures used in the algorithm. The global matrix $Cost[n][n]$ is used for the problem graph, and the array $p[n]$ is used to keep the nodes-to-part mapping. Among the local structures, $gain[i][j]$ gives the expected gain in partition-cost if node i moves to part j, $history[n]$ keeps the moving history of the current pass, $temp[n]$ keeps the temporary gains corresponding to array $history$, and array $state[n]$ keeps track of whether a node is moved in this pass or not.

The algorithm consists of a series of passes (loop of lines 3–22): in each pass, we do n iterations to move the n nodes. In each iteration (loop of line 7–18) we perform the following steps. At first, we choose a node which will

```
int n, k, Cost[n][n], P[n];

procedure LEE ()
{
    int gain[n][k], state[n], history[n][2], temp[n], G, Gmax, R;
    Construct initial partition and choose R
    do {
        Initialize states: ∀i : 1 ≤ i ≤ n, state[i] = 0.
        Compute initial gains, that is ∀i, j : 1 ≤ i ≤ n, 1 ≤ j ≤ k compute:
            gain[i][j] = Σ_{1≤l≤n, P[l]=P[i]} (R − Cost[i][j]) − Σ_{1≤l≤n, P[l]=j} (R − Cost[i][j]);
        for (i = i; i ≤ n; i++){
            Find gain[d][l] = max_{1≤i≤n, 1≤j≤k, state[i]=0} gain[i][j], p=P[d];
            Move node d to part l: state[d] = 1, P[d] = l, temp[i] = gain[d][l],
                history[i][1] = d, history[i][2] = P[d];
            for (j = 1; j ≤ n; j++) {
                if (P[j] = p) gain[j][l] = gain[j][l] − 2 * R + 2 * Cost[d][j].
                elseif (P[j] = l) gain[j][p] = gain[j][p] + 2 * R − 2 * Cost[d][j].
                else {
                    gain[j][p] = gain[j][p] + R − [d][j].
                    gain[j][l] = gain[j][l] − R + [d][j].
                }
            }
        }
        Find t such that G = Σ_{j=1}^{t} temp[j] is maximum at value Gmax.
        If (Gmax > 0) restore all interchanges after t:
        for (j = t + 1; j ≤ n; j++) P[history[j][1]] = hisrtory[j][1].
    }while (Gmax > 0);
}
```

Figure 2.1: Lee's algorithm for k-way graph partitioning.

give the maximum gain (or minimum decrease) in partition-cost from among the ones that have not yet been considered during the pass (line 8). Then we move it to the corresponding part (line 9), and adjust the gain array (line 10–17). The n partitions produced during the pass are examined and the one with the maximum partition-cost is chosen as the starting partition for the next pass(line 19–21). Passes are performed until no improvement in partition-cost can be obtained (line 22).

Lee *et al.* have also shown that if the algorithm converges in r passes then the complexity of algorithm LEE will be $\mathcal{O}(rkn^2)$ the details of which can be found in [25].

In the original algorithm of [25], Lee *et al.* suggested an initial partition where all the nodes are initially put in part 1. However, our implementations and experiments show that in general Lee's algorithm gives better quality of solutions if it initially starts with n parts where node i goes to part i. For this reason we have modified the initial partition of Lee's algorithm as mentioned above. Thus from now on, when we say 'Lee's algorithm' we mean the modified Lee's algorithm, and when we want to refer to the algorithm given in [25], we call that by the *original* Lee's algorithm.

2.5 Our Motivation

One of the main motivations of our work comes from the current outburst of interests in distributed systems and algorithms. In recent years, distributed computing systems (DCS) have been conjectured to be one of the most promising solutions to the computing requirements for the future. A DCS is a collection of autonomous computers interconnected by a communication network. During the process of computation, the processors communicate among them by exchanging messages through the network. One major objective of distributed systems is better utilization of resources, and hence achieve better performance. For example, a network of workstations or personal computers connected by a LAN running a network operating system is nowadays very common in many institutions. As the properties of DCS as a computing platform are much different as compared to sequential or shared memory parallel computers, intensive research effort has been attracted to design and study this new environment [31, 32, 33, 19, 34, 35] and also to design suitable algorithms [36, 37, 38, 39] so as to give the best possible performance improvement. In the literature, distributed algorithms are available for many important problems, such as, shortest path problem [40, 41, 42, 43], graph searching [44, 45], minimum spanning tree problem [46, 47], and network flow problems [39, 48].

To the best of our knowledge, there seems to be no distributed algorithm currently existing in the literature for the GP problem. However, there are at least two motivations as to why graph-partitioning may be an interesting problem to be considered for distributed implementations. Firstly, utilization of multiple computers in the distributed environment may lead to potential speed-up in terms of execution time as compared to its sequential performance, which is very important for this inherently intractable problem. Also a distributed algorithm might be able to handle bigger sizes of input which a sequential implementation might fail to handle. Secondly, partitioning a graph arises in many scenarios of design and running of a DCS, both at system and application levels. For example, given a number of processes, and a number of computers for running them, one has to solve the problem of allocating the processes to the processors, which can be modeled as a GP problem. Again, in many combinatorial problems, the input can be modeled as a graph, and an efficient approach to solve the problem is to divide the problem into smaller subproblems, send each to a processor, solve them locally, and then combine the solutions to get the overall solution. For this divide-and-conquer type of algorithms, partitioning the input graph is the first problem to be solved efficiently.

In our investigation for this project, we started with the belief that some simple algorithms should be better for distributed environments as opposed to the sophisticated ones. Here by the word 'simple' we imply less computation and communication complexities. This is because simple algorithms should be easy on the communication and synchronization requirements. Since we did not find any such suitable algorithm in the literature, we had to go our own ways, and came up with some simple interesting ideas which were later turned into the algorithms Auction, GreedyPass and GreedyCycle. Of these Auction is a direct algorithm, and GreedyPass and GreedyCycle are iterative algorithms. It is also interesting to note that, even though we started this project for some distributed GP algorithms, we found that all of the above algorithms also have efficient sequential implementations with very good time performance. Specifically, GreedyPass turned out to be an inherently sequential algorithm, which is not very easy to implement in distributed environments.

Hence at the final stage of the project, we worked on some general k-way GP algorithms — both for sequential and distributed environments. We discussed the characteristics of these algorithms in full detail, implemented them, and did some experimental performance tests comparing them with other existing GP algorithms.

2.6 Distributed Environment — PVM

To implement our distributed algorithms we need a distributed programming environment. There are quite a few distributed programming environments available from many research projects, for example, Amoeba [49, 34], Linda [50], and PVM [35, 51]. These different environments lead to different programming paradigms. For our implementations, we used the PVM environment. Therefore for the rest of the section we will discuss some of the features of PVM related to our works. For detailed discussion on PVM we refer to [35, 51].

PVM (Parallel Virtual Machine) is a message-passing system for heterogeneous collections of networked computers developed jointly at Emory University, University of Tennessee, and Oak Ridge National Laboratory. It is a public domain software distributed freely available from the *ftp* site *netlib@ornl.gov.* Some of the key features are given below:

- PVM consists of two parts: a daemon process that any user can install on a machine, and a user library that contains routines for initiating processes on other machines, for communication between processes, and changing configuration of the machines.
- It is portable over many architectures such as Alliant FX/8, BBN Butterfly, DEC Alpha, Connection Machines, Crays, KSRs, SUNs, etc. It can also run in a network of heterogeneous computers.
- It supports C and Fortran library functions for the programs to interact with the system. For our programming, we used some of the primitive routines: for process spawning, data packing and unpacking, point-to-point message passing, and message multicasting.
- PVM has the following advantages which attracted us to work with it.
  - It is free and easy to obtain from the network.
  - It is small, simple, and easy to program.
  - It is portable across many Unix[2]-based systems.

[2]Unix is a registered trademark of AT&T Bell Laboratories

- Heterogeneity is well supported.
- There is a large user group.

- However it has also got some weaknesses.
  - Message passing performance is not very good. One reason for this is that the designers concentrated on heterogeneity and portability rather than performance.
  - Process spawning and message broadcasting are done in a linear fashion.
  - Uses a simple round robin load balancing.
  - There is little support for debugging, and no support for program tracing.

For our implementation we used PVM version 3 released in May 1993. It was installed on a total of 28 workstations interconnected by an Ethernet local area network.

Chapter 3

A Direct Algorithm

Our main objective in this chapter is to devise a constructive algorithm for the GP problem. We take the following simple idea: we start with k nodes as k parts, and then let the parts grow as they collect new nodes. This idea is intuitive and is common in many natural phenomena. For example, during the crystallizing process of salt from brine, the crystallizing starts when some very small crystals are somehow formed at certain points. Then these crystals get bigger and bigger as new molecules accumulate on them. Here the chemical and natural laws govern which crystals get which molecules. In our algorithm, however, we use the weights of graph connectivities to simulate the chemical and natural force factors; and a central algorithm simulates the enforcement of the physical laws by ensuring that a node moves to the direction of highest connectivity 'force'. Following this analogy of our algorithm to crystallizing, the algorithm can be termed as a simulated crystallizing algorithm for the GP problem.

This chapter is organized as follows. In the first section, we describe the AUCTION algorithm in its generic form. In the next section, we give some algorithms as implementation choices for the AUCTION algorithm. In the last two sections, we describe the implementation issues and complexities in the sequential and distributed environments.

3.1 The Auction Algorithm

The main idea of the algorithm is very simple and intuitive. It can best be presented using a master-workers kind of distributed computing model. Here the main computation is controlled by a master process. To divide a graph into k parts, the master spawns k worker processes. The computation proceeds in the following three stages:

- At first the master somehow chooses one initial node, called seed, for each of the workers, and broadcasts this information along with the input graph to the workers.
- Then each worker computes a bid[1] for the graph nodes giving higher priority to the nodes closer to its seed node, and send this bid to the master.
- Finally, after the master has got the bids from all the workers, it assigns the nodes to the workers by comparing the bids for each node. The worker who gave the highest priority to a particular node gets it. If there is a tie in the priority given by the workers, the node may be given to any one of the competitors.

In this algorithm, we note that, after initiating each worker by a seed, the main underlying idea is to encourage each worker to fetch its seed's neighbors in a

[1] Here *bid* means a one-to-one mapping of graph node indices to integers 1 to n. Thus when we say, 'a worker computes its bid', we mean that it puts a distinct priority number to each graph node based on how much it likes the node to get for itself. A lower priority number means higher preference.

```
int n, k;                  // number of nodes, and parts
int D[n];                  // node u has D[u] neighbors
int *GList[n];             // jth neighbor of node u is at GList[u][2*j],
                           // and this edge-cost is at GList[u][2*j+1]
int P[n];                  // node u goes to part P[u]
int Worker[k], Seed[k];    // workers' pids, and initial seeds
int Bid[k][n];             // bid from Worker[j] is kept in array Bid[j][1:n]
```

```
procedure AUCTION()
{
    for (i = 1; i ≤ k; i++)                   // initialization
        spawn Worker[i] with AUCTIONW;
    for (i = 1; i ≤ k; i++)                   // seed computation
        calculate Seed[i];
    broadcast n, k, D, GList, Worker          // data broadcast
        and Seed to all the workers;
    for (i = 1; i ≤ k; i++) {                 // bid receiving
        wait to receive a bidding response;
        answer from Worker[j] is kept in array Bid[j];
    }
    for (i = 1; i ≤ n; i++)                   // node distribution
        find Worker[j] to assign node i, and set P[i] = j;
}
```

Figure 3.1: Distributed auction algorithm for graph partitioning: Algorithm at the master AUCTION.

int $n,\ k$; // number of nodes, and parts
int $D[n]$; // node u has $D[u]$ neighbors
int $*GList[n]$; // jth neighbor of node u is at $GList[u][2*j]$,
// and this edge-cost is at $GList[u][2*j+1]$
int $me,\ master,\ myseed$; // my pid, master's pid, and my initial seed
int $Mybid[n]$; // my array of bids for n nodes
int $Worker[k],\ Seed[k]$;

procedure AuctionW()
{
receive $n,\ k,\ D,\ GList,\ Worker$
and $Seed$ from the master; // initialization
$myseed = Seed$[my position in $Worker[1:k]$];
calculate $Mybid[i], \forall i: 1 \leq i \leq n$; // bid computation and sending
send $Mybid$ to the master;
}

Figure 3.2: Distributed auction algorithm for graph partitioning: Algorithm at each worker processor AuctionW.

competitive basis. Since the assignment of a node to a particular part is decided based on the priorities (bids) given for this node from different workers, we refer to this algorithm as the AUCTION algorithm. This strategy is based on the intuitive assumption that, the problem graph will have some locality properties, and the priorities given by the bids will also have useful global significance when used for partitioning.

Figure 3.1 gives our basic AUCTION algorithm in its generic form. Specific choices for the implementations will be discussed in the next sections. In figure 3.1, we have basically two algorithms, algorithm AUCTION for the master, and algorithm AUCTIONW for each worker.

3.1.1 Global Data Structures

In all our algorithms including AUCTION, we use the same global data-structures to represent the problem graph, and the partition information. Integer n gives the number of nodes, and integer k gives the number of parts we want from the graph. The degrees of the nodes are kept in array $D[1:n]$ which means that node u has a total of $D[u]$ neighbors. The graph edges and their weights are kept in a linked list structure given by $*GList[1:n]$ which is an array of pointers to arrays. Thus all the information of the edges adjacent with node u is kept in an array $GList[u][1:2*D[u]]$ of $2*D[u]$ elements which keep the indices of the neighbors and the weights of the corresponding edges. In this array, this information is kept in a linear fashion where the jth neighbor of node u is found at element $GList[u][2*j]$ and the cost of the edge connecting node u to this neighbor is found at $GList[u][2*j+1]$. To keep the mapping of the nodes to the parts (that is which node is where), we use the global array $P[1:n]$ which means that, at a certain point of time, node u belongs to part $P[u]$.

3.1.2 Master Algorithm AUCTION

Algorithm AUCTION proceeds in five steps: initialization, seed computation, data broadcasting, bid receiving, and finally node distribution. At first, the master spawns the worker processes with the algorithm AUCTIONW (line 1–2). Secondly, the master computes the initial seeds for the parts (line 3–4). Thirdly, the master broadcasts the problem information and the seeds to all the workers (line 5). After getting the graph and the seeds, the workers compute their bids, and send them to the master. Therefore at the next step (line 6–9), the master starts to wait for the bids. Answer from $Worker[j]$ is kept in array $Bid[j]$. The master waits until bids are received from all the workers. Finally, the master assigns the nodes to the workers, one node at a time, considering and comparing all the worker bids according to some strategy; and saves this mapping in array $P[n]$ (line 10–11).

3.1.3 Worker Algorithm AUCTIONW

We have two stages in algorithm AUCTIONW: initialization, and bid computation and sending. At first, each worker receives the initial data about the problem and seeds from the master (line 1). It finds its seed $myseed$, and then processes the problem inputs using this seed to find an array of bids $Mybid[n]$, one bid for each node (line 2–3). A lower bid for a node means higher priority. Finally the worker sends its bid array to the master (line 4).

We observe that, if we have $k+1$ processors to run algorithm AUCTION, we can run the master and each worker at different processors. Each worker process can compute its bid, which is the main computation in our algorithm, independent of any other. Thus the main computation in the algorithm is inherently distributed in nature.

3.2 Some Implementation Issues

The algorithm in figure 3.1 is generic in the sense that there are still some open issues where many variations of implementations are possible. Here we mention some possible implementation choices on some issues.

3.2.1 Seed computation strategy for master

To choose the initial seeds for the partitions, the master can use many strategies. In the simplest implementation, the master can choose nodes randomly from the graph. Another choice may be doing some preprocessing on the input graph to find some better seeds heuristically. The idea here is to find the seeds such that they are spread over the graph as opposed to being mutually close. For example, one might feel that some connectivity based search techniques may be helpful here. Actually, to find a good heuristic for computing the seeds is a difficult problem, possibly as difficult as the GP problem itself. In this thesis, due to time limitation we could not pursue this issue deeply, and leave this as a future work. Therefore, in our implementations, we used the simple random selection strategy to find the seeds.

3.2.2 Bid computation strategy for the workers

Another implementation issue is the bid computation algorithm for the workers. This is not a trivial problem. Many choices are possible. For example, we can simply try a searching technique, such as breadth-first-search, to have the bidding order. The nodes closer to the seed will be expanded earlier, and given higher priority.

Let us now discuss another greedy heuristic of expanding the nodes around

```
int n, k;                    // number of nodes, and parts
int D[n];                    // node u has D[u] neighbors
int *GList[n];               // jth neighbor of node u is at GList[u][2*j],
                             // and this edge-cost is at GList[u][2*j+1]
int con[n];                  // for node i, con[s] = the weight-sum of edges radiated
                             // from i to part s
int count = 0, color[n];     // current bid counter, and node color
```

```
procedure ComputeBid (myseed: input, Mybid: output)
int myseed, Mybid[n];       // my seed and array of bids
{
    for (i = 1; i <= n; i++)
        color[i] = WHITE, con[i] = 0;
    con[myseed]=1;
    do {
        find u, such that: con[u] = max_{v∈V, color[v]=WHITE} con[v];
        Mybid[u] = count, color[u]=BLACK;
        count = count + 1;
        for (i = 1; i <= D[u]; i++){
            j = GList[u][2*i];
            if (color[j] == WHITE){
                con[j] = con[j] + GList[u][2*i+1];
            }
        }
    } while (count <= n);
}
```

Figure 3.3: A bid computation algorithm for the workers using highest core connectivity.

a core. Algorithm ComputeBid given in figure 3.3 is one such algorithm. In this algorithm, we start with a one-node core. As the algorithm continues more and more nodes enter into the core. It proceeds in iterations. In each iteration one node is allowed to enter into the core. To select the next node to enter the core, we use an index what we call a node's *core_connectivity*. The core_connectivity of a node is the sum of the costs of all its edges to its neighbors who are already inside the core. Mathematically, core_connectivity of node i is given as, $core\_connectivity(i) = \sum_{j \in \text{core}} (\text{cost of edge } [i, j])$. Having all the core_connectivities of the currently external nodes at hand, the external node with the highest core_connectivity is chosen to enter the core at any iteration.

Let us now introduce the data-structures we use in algorithm ComputeBid. We use an array $con[n]$ to keep the core_connectivities of the graph nodes. Also to keep track of which nodes are inside the core and which are outside, we use the array $color[n]$. If $color[i]$=WHITE, node i is outside the core, else $color[i]$=BLACK which means that node i is inside the core. To track the priority number a worker puts to a node (that is the bid), we use the index *count*.

Algorithm ComputeBid of figure 3.3 goes as follows. It takes the seed *Myseed* as an input parameter, and produces the bids in global array $Mybid[n]$. At first, we initialize the local arrays $color[n]$ and $con[n]$. We put an artificial connectivity of value 1 to node *myseed* in order to ensure that the seed is the first node taken into the core (line 3). Then we enter the loop (line 4–14), where we insert nodes into the core one at a time. In each iteration, we first select the node to be taken into the core in this pass by finding the external node u with maximum core connectivity and then take the node u into the core by assigning the current value of *count* to node u and then coloring it to BLACK (5–6). We increment the counter *count* and then properly update the connectivities of the remaining external nodes (line 7–13). If there is still some external node(s), we start the next iteration from line 4. Otherwise, we terminate the algorithm.

The straight-forward algorithm ComputeBid of figure 3.3 is easy to implement, but that will not be very efficient in time performance. In an investigation for better time performance, we felt that some use of sophisticated data-structures may be suitable for this algorithm. As we discussed earlier, in the ComputeBid algorithm, we always pick the highest connectivity out-of-core node to be absorbed inside, and after the absorption we update the connectivities for the related out-of-core nodes. For such an application some kind of heap structure are usually suitable. Heap is a very important data-structure discussed in all the data-structure or algorithm textbooks like [16]. Discussing heaps in detail are beyond the scope of this thesis. Therefore we confine ourselves to a very short introduction here.

A heap data-structure can be viewed as a tree structure of a list of *elements* each having a *key* with the property that a child's key is never bigger than its parent's key. It supports the following operations: Insert to insert a new element with a given key, FindMax to get a pointer to the maximum key element, ExtractMax to find and extract the element with maximum key value from the list, IncreaseKey to increase the value of a key, Delete to delete an element, and Union to unite two heaps together. There are many types of heap-structure known and studied with their different strengths and weaknesses. In our application for bid computation, we observe that in every iteration, we need one ExtractMax and $D[u]$ IncreaseKeys. Therefore the best suited heap for us is *Fibonacci heap*, where the amortized cost of ExtractMax is $\mathcal{O}(\lg n)$, and the amortized cost of IncreaseKey is $\mathcal{O}(1)$ [16]. For these reasons we used the implementation ComputeBid2 of figure 3.4 for our programs.

```
int n, k;                     // number of nodes, and parts
int D[n];                     // node u has D[u] neighbors
int *GList[n];                // jth neighbor of node u is at GList[u][2*j],
                              // and this edge-cost is at GList[u][2*j+1]
int con[n];                   // for node i, con[i] = the weight-sum of edges radiated
                              // from i to the nodes in the core
int count = 0, color[n];      // current bid counter, and node color


procedure ComputeBid2 (myseed: input, Mybid: output)
int myseed, Mybid[n];         // my seed and array of bids
{
    for (i = 1; i ≤ k; i++){
        color[i] = WHITE, key[i] = 0;
        HeapInsert(i);
    }
    IncreaseKey(myseed, 1);
    do {
        u=ExtractMax();
        Mybid[u] = count, color[u]=BLACK;
        count = count + 1;
        for (i = 1; i ≤ D[u]; i++){
            j = GList[u][2*i];
            if (color[j] == WHITE){
                newcon = con[j] + GList[u][2*i+1];
                IncreaseKey(j, newcon);
            }
        }
    } while (count ≤ n);
}
```

Figure 3.4: Bid computation algorithm using Fibonacci heap.

3.2.3 Node distribution strategy for the master

After the master has got all the bids from the workers, the master assigns and distributes the nodes to the parts. Here implementation may again have some flexibility. In the simplest way, one might think that the master can just distribute the nodes based on the node ordering in the received bids. For a particular node, the part which has given the lowest order to it, gets the node in that part. Actually, this is a really greedy approach, where the algorithm has no control over the number of nodes each part is getting which might violate our balancing constraint. Figure 3.5 shows one such node distribution algorithm with balancing. Here the variable min is used for tracking the minimum bid, and $jmin$ to track the corresponding part. Which part is getting how many nodes is kept using array $got[k]$.

The algorithm uses a very simple idea. At the first phase, nodes are distributed on a purely competitive basis among the parts who did not get $\lfloor n/k \rfloor$ nodes yet. Once every part has got at least $\lfloor n/k \rfloor$ nodes each, the condition of line 4 will never be true again. If there is still some nodes left to be distributed, those are distributed just by using a simple mod function[2] as given in line 2. Hence algorithm DistributeNodes guarantees that parts 1 to $(n - \lfloor n/k \rfloor * k)$ will get $\lceil n/k \rceil$ nodes, and the rest of the parts will get $\lfloor n/k \rfloor$ nodes, and the difference between the number of nodes assigned to any two parts will never be more than one.

[2]Throughout this thesis, when we say we used a mod function for the initial partition or a simple starting partition, we mean that node i initially goes to part $[(i \bmod k) + 1]$

```
int n, k;                    // number of nodes, and parts
int D[n];                    // node u has D[u] neighbors
int *GList[n];               // jth neighbor of node u is at GList[u][2*j],
                             // and this edge-cost is at GList[u][2*j+1]
int P[n];                    // node u goes to part P[u]
int Worker[k], Seed[k];      // workers' pids, and initial seeds
int Bid[k][n];               // bid from Worker[j] is kept in array Bid[j][1:n]
int got[k];                  // part s currently has got[s] nodes
int min, jmin;               // to find part with minimum bid
```

```
procedure DistributeNodes()
{
    for (i = 1; i ≤ n; i++){
        min = ∞, jmin = i%k + 1;
        for (j = 1; j ≤ k; j++){
            if (Bid[j][i] < min && got[j] ≤ ⌊n/k⌋)
                min = Bid[j][i], jmin = j;
        }
        P[i] = jmin;
        got[jmin] = got[jmin] + 1;
    }
}
```

Figure 3.5: A node distribution algorithm with balancing for the master.

3.3 Algorithm Complexity

Before we investigate the complexity of algorithm Auction, let us first investigate the complexities of functions ComputeBid2 and DistributeNodes.

In loop 1–4 of ComputeBid2, all the nodes inserted into the heap have the same key value of zero, hence this can be done in $\mathcal{O}(n)$ time. Now using the knowledge that, every ExtractMax has an amortized average cost of $\mathcal{O}(\lg n)$ and every IncreaseKey has an amortized average cost of $\mathcal{O}(1)$, the complexity of algorithm ComputeBid2 is,

$$\begin{aligned}
t_{CBid2} &= \mathcal{O}(n) + \sum_{u \in V} \{\mathcal{O}(\lg n) + \sum_{i=1}^{i=D[u]} \mathcal{O}(1)\} \\
&= \mathcal{O}(n) + \mathcal{O}(n \lg n) + \sum_{u \in V} \sum_{i=1}^{i=D[u]} \mathcal{O}(1) \\
&= \mathcal{O}(n) + \mathcal{O}(n \lg n) + \mathcal{O}(m) \\
&\approx \mathcal{O}(n \lg n + m). \\
\text{Hence,} \quad t_{CBid2} &= \mathcal{O}(n \lg n + m).
\end{aligned}$$

In algorithm DistributeNodes, loop 1–9 iterates n times, and loop 3–6 iterates k times. Hence the complexity of DistributeNodes is,

$$t_{DNodes} = \mathcal{O}(kn).$$

The time-wise sequence of events happening in the whole master-workers AUCTION algorithm starting from the invocation of the master to the production of output are as follows:

1. The master spawns the k worker processes.
2. The master computes the seeds.
3. The master broadcasts the initial information; and in parallel, the workers wait and receive the inputs.
4. The master waits and receives the k bids; and in parallel, each worker computes its bids and sends it to the master.
5. The master distributes the nodes.

We note that there are two send-receive synchronization points in the algorithm. Once when the workers wait to receive the initial information, and finally when the master waits for the workers' response. To simplify the analysis, we assume the following expressions or functions:

$S_p(k)$ = the time required by the master to spawn k worker processes,
$B(k)$ = the time required by the master to broadcast the initial information to the workers,
S_{yw} = the time required for the synchronization overhead at each worker. It is the delay at each worker between the receiving of the initial information and the extraction of data.
$S_e(k)$ = the time required by a worker to send a bid to the master, and
$S_{ym}(k)$= the time required for the synchronization overhead at the master. It is the sum of the delays in master between the receiving of a bid, and extracting the data from it.

Now following the above steps of operations, the time complexity of algorithm AUCTION can be found as,

$$\begin{aligned} t_{Auction} &= S_p(k) + B(k) + \mathcal{O}(S_{yw} + t_{CBid2} + S_e(k) + S_{ym}(k)) + t_{DNodes} \\ &= \mathcal{O}(n \lg n + m + kn) + S_p(k) + B(k) + S_{yw} + S_e(k) + S_{ym}(k) \\ &= \mathcal{O}(n \lg n + m + kn) + S_p(k) + B(k) + S_e(k) + S_y(k). \end{aligned}$$

Here the total synchronization cost, $S_y(k) = S_{yw} + S_{ym}(k)$.

In the above expression of $t_{Auction}$, two components of costs can be identified: the first part is $\mathcal{O}(n \lg n + m + kn)$ is the computation component and the last part $(S_p(k) + B(k) + S_e(k) + S_y(k))$ is the communication and synchronization component. The ratio of these two components depends on the communication support of the distributed environment the algorithm is implemented, and also on the problem size in n, m, and k.

3.4 Discussion

Some of the key features of algorithm AUCTION are given below.

- Algorithm AUCTION is a very simple, heuristic and direct GP algorithm.
- It is an inherently distributed algorithm, and it seems that it is the first distributed algorithm proposed for the GP problem. If we use $(k+1)$-processor to solve the problem, and the communication overhead is ignored, then the time complexity of algorithm AUCTION will be $\mathcal{O}(n \lg n + m + kn)$. In comparison, if Lee's algorithm converges in r passes, then its time complexity will be $(\mathcal{O}(rkn^2))$.

- Since it uses adjacency list format, the per-processor memory requirement is ($\mathcal{O}(2m)$) which is less compared to other methods such as Lee's where adjacency matrix format requires more memory ($\mathcal{O}(n^2)$). However if we use p processors, the total memory requirement will be ($\mathcal{O}(2pm)$).

- The average solution quality given by algorithm AUCTION is generally worse than iterative algorithms, such as Lee's algorithm. In chapter 5, we will show that, the solution quality of algorithm AUCTION is within about 20% of the quality given by Lee's algorithm, one of the best algorithms available for this problem in the literature. This quality of solution is reasonable, especially when we remember its excellent time performance.

Chapter 4

Iterative Improvement Algorithms

Among the GP algorithms available in the literature, iterative improvement methods are the most popular ones. In general, an iterative technique starts with some initial partition, and then repeatedly tries to move towards better and better objective function. It uses a local search strategy, where it considers a small neighborhood of states and the next iteration starts at the most promising one among them. In general, the iterative improvement heuristic algorithms give good quality of solutions in reasonable time. Another good characteristic of this technique is that, even if the computation stops abruptly at any point of time for certain reasons, we are not completely empty; at least the best solution known so far is at our hand.

In this chapter, our main objective is to devise some iterative algorithms for the GP problem both for sequential and distributed environments. In the first section, we give a very simple sequential algorithm GreedyPass for the k-way GP problem. We discuss its data-structures as well as its computational complexity. In the next section, we propose the algorithm GreedyCycle introducing the idea of greedy cyclic exchange. Cyclic exchange is a k-way generalization of the 2-way exchange proposed by Kernighan and Lin [2].

4.1 Greedy Passing

In this section, we give a simple heuristic algorithm GreedyPass for the GP problem in sequential environments. In this algorithm, we use a greedy and intuitive extension of the idea of profitable node transfers between parts found in the KL method [2, 4]. Here, we start with any balanced k-way partition, and then improve iteratively by choosing one node from one part, and moving it to another. The node to be moved is chosen so that a maximum decrease in cutsize (i.e., maximum gain) is obtained. From now on, we will use the expressions *decrease in cutsize* and *gain* interchangeably. To present algorithm GreedyPass given in Figure 4.2, we use the following sequence. We first describe the data-structures used, then we describe algorithm GreedyCompute which we need in GreedyPass and later in GreedyCycle, and finally we detail algorithm GreedyPass.

4.1.1 Algorithm GreedyCompute

Algorithm GreedyCompute of Figure 4.1 does the main computation in the algorithm. It takes a part number s as input parameter, and computes two output parameters *going* and *snext* using the input graph and current partition information, where *going* gives the node which obtains the maximum achievable gain under current circumstances, and this gain is obtained if the node is sent to part *snext*. It returns the maximum gain achievable by moving a node from part s. If there is no effective gain possible, it gives the value -1 for both *going* and *snext*.

For our computation of gain, we use a gain-vector $gvector[1 : k]$ for each

```
int n, k;                 // number of nodes, and parts
int D[n];                 // node u has D[u] neighbors
int *GList[n];            // jth neighbor of node u is at GList[u][2*j],
                          // and this edge-cost is at GList[u][2*j+1]
int P[n];                 // node u goes to part P[u]
int gvector[k];           // gain-vector: for node i, gvector[s] = the weight-sum
                          // of edges radiated from node i to part[s]
int maxgain = 0;          // maximum gain achievable in current pass


procedure GreedyCompute(s: input, going: output, snext: output)()
int s, going, snext;      // current part, best node to go, and where to
{
    Let indices s_1, s_2, s_3, ..., s_l give the nodes in part s;
    snext = going = −1;
    for (i = 1; i ≤ l; i++){
        ∀j, 1 ≤ j ≤ k : gvector[j] = 0;
        for (j = 1; j ≤ D[s_i]; j++){
            u = GList[s_i][2*j];
            gvector[P[u]] = gvector[P[u]] + GList[s_i][2*j+1];
        }
        Find t such that gvector[t] = max_{1≤j≤k} gvector[j];
        if ((gvector[t] − gvector[s] > maxgain)
            snext = t, going = s_i, maxgain = gvector[t] − gvector[s];
    }
    return maxgain;
}
```

Figure 4.1: Algorithm GreedyCompute for a part to compute which node to go and where.

node. Here $gvector[u]$ for a particular node is computed as the summation of the costs of all the edges which this node radiates to part u. Thus when node i currently in part s is moved to part k, the cutsize is decreased by $gvector[k] - gvector[s]$.

Algorithm GreedyCompute iterates on all the nodes of the current part to find the node which will give maximum gain. At the start of each iteration, we compute the array *gvector* for the current node (line 4–8). Then we find the part where this node will give maximum gain, and if this gain is the so-far maximum achievable gain we save the node number in *going*, the corresponding destination in *snext*, and the gain in *maxgain* (line 9–11). Thus at the end of the loop 3–12, we know that maximum gain we can have from this part is *maxgain*, and this will be achieved when node *going* is sent to part *snext*.

4.1.2 Algorithm GreedyPass

Figure 4.2 gives algorithm GreedyPass for graph partitioning. Each iteration starts with a current part, and tries to minimize the partition cost as the iterations continue.

At first, we choose a part from the initial partition as the current part denoted by part s (line 1). Then we start the series of iterations in order to minimize the cost gradually. Each iteration (loop 2–6) proceeds in the following sequence. We first call GreedyCompute to compute the *going* node and the corresponding *snext* (line 3). If the maximum gain returned is positive, we move the *going* node to *snext*, and the next iteration starts with *snext* as the current part (line 4–5). Finally, when we finish an iteration with no positive maximum gain, we exit the algorithm (line 6). Since the iterations continue only as long as we make gains, we call this a 'greedy' algorithm. We break the loop the first time we do not make any gain.

```
int n, k;                   // number of nodes, and parts
int D[n];                   // node u has D[u] neighbors
int *GList[n];              // jth neighbor of node u is at GList[u][2*j],
                            // and this edge-cost is at GList[u][2*j+1]
int P[n];                   // node u goes to part P[u]
int maxgain = 0;            // maximum gain achievable in current pass
int s, going, snext;        // current part, best node to go, and where to


procedure GreedyPass()
{
    s = any part number in initial partition with size ⌈n/k⌉;
    do {
        maxgain = GreedyCompute(s, going, snext);
        if (maxgain > 0)
            P[going] = snext, s = snext;
    }while (maxgain > 0);
}
```

Figure 4.2: Simple sequential algorithm GreedyPass for k-way graph partitioning.

4.1.3 Balance Constraints

Let us now discuss how algorithm GreedyPass takes care of the balancing requirement of the GP problem. The algorithm starts with a k-way balanced initial partition. If n is not divisible by k, then some parts have size $\lceil \frac{n}{k} \rceil$, and the rest have size $\lfloor \frac{n}{k} \rfloor$. Otherwise, all the starting parts have the same size $\frac{n}{k} = \lceil \frac{n}{k} \rceil = \lfloor \frac{n}{k} \rfloor$. At the start of the algorithm, we choose a part with size $\lceil \frac{n}{k} \rceil$ to ensure that there is no part bigger than this one in any case, and make it as the current part to consider. Then we start the main loop of passes. In each pass, the current part gives away a node, and moves it to the most profitable destination part. After this movement, there is no part bigger than the destination, and hence in the following pass it is made the current part. This is continued until the algorithm finishes. We note from the above description that, during the progress of the entire algorithm, it is guaranteed that no two parts will ever have a size difference of more than two nodes, which is a very reasonable balancing for most applications.

4.1.4 Termination

We now discuss whether algorithm GreedyPass terminates in finite number of iterations. We assume that n, k, and the cost of each graph edge are finite; and in algorithm GreedyPass we make some finite gain in each iteration. The number of feasible states where algorithm GreedyPass can go with the balance requirement fulfilled as discussed earlier is now finite, because there are finite such partitions possible. We can compute the partition-costs for all such feasible partitions, and as these costs are finite, there will be at least one minimum value. Let that value be C_{min}. Also, let the minimum possible finite gain achieved in each iteration be δ.

Now let us consider that our algorithm GreedyPass starts with an initial partition of cost C_{ini}. Since there is a finite minimum feasible cost C_{min}, and the minimum gain in each iteration δ is also finite, after at most $\frac{Cini-Cmin}{\delta}$ (a finite number) iterations, algorithm GreedyPass will reach a partition from where it cannot find any positive $maxgain$. Thus GreedyPass must terminate, and the number of iterations required for the algorithm to converge must be finite. Let the number of iterations be represented by r.

4.1.5 Complexity

At first we consider algorithm GreedyCompute. For balanced partitions, the size of a part is given by $l \approx n/k$. Line 5 requires k assignments, loop 6–9 requires $2D[s_i]$ operations, and line 10 requires k operations. The cost of lines 11–12 is constant. Thus the computational complexity of GreedyCompute is,

$$
\begin{aligned}
t_{GComp} &= \sum_{i=1}^{l=n/k} (2k + 2D[s_i]) \\
&= \sum_{i=1}^{l=n/k} 2k + \sum_{i=1}^{l=n/k} 2D[s_i] \\
&\approx 2n + \frac{2}{k}\sum_{i=1}^{n} D[s_i] \\
&= 2n + \frac{2}{k} \cdot 2m \\
&= 2n + \frac{4m}{k}. \\
\text{Hence,}\ \ t_{GComp} &= \mathcal{O}(n + \frac{m}{k}).
\end{aligned}
$$

Since GreedyCompute is the main computation in the loop 3–7, and other components do not have considerable effect on the order of the computational complexity of GreedyPass, the computational complexity of algorithm

GreedyPass is,

$$\begin{aligned} t_{GPass} &= \mathcal{O}(rt_{GComp}) \\ &= \mathcal{O}(rn + \frac{rm}{k}). \end{aligned}$$

4.2 Greedy Cycling

In this section, we give another heuristic iterative algorithm GreedyCycle for the GP problem where we introduce the idea of *cyclic node passing* among parts. This algorithm starts with an initial partition. The main idea of the algorithm is as follows. For each part in the current partition, we compute the most profitable node to go, and the corresponding destination, if any. Then the algorithm loops to find all the instances where there is a cycle of node-passing-interests, and whenever successful, passes the appropriate nodes along these cycles. Once all the cycles are processed, the iteration starts again. The iteration continues as long as we continue improving on the partition-cost.

Let us now see how the KL-method [2] suits for the k-way GP problem. KL-method is basically a 2-way partitioning which iteratively improves the partition to minimize the cost. At every step, this algorithm considers the gain that would be achieved by exchanging a pair of nodes between the parts, and iterates on that. As we have already mentioned in chapter 2, this idea is the basis of most of the iterative algorithms currently available in the literature. However, this does not seem to be very suitable for the generalized k-way GP problem. As an example, let us consider a simple case where part A wants to send a node to part B, part B wants to send to part C, and part C wants to send to part A. 2-way node exchange method will fail here because there is no profitable node exchange possible, and it is not able to see the cyclic interests of profitable node passing. Thus, we see that a KL-based 2-way exchange method might miss many of the partition-cost improvement opportunities, and therefore we need some algorithms which overcome this basic problem.

Algorithm GreedyCycle seems to be the first attempt targeted to overcome the above problem for the generalized k-way partitioning. Cyclic node passing is a new idea, and it is an intuitive k-way generalization of the 2-way node exchange found in the KL-method. It is a generalization because cyclic node passing include 2-way node exchange as a special case where $k = 2$. This approach seems very interesting for the following three reasons:

- It has the potential to give better optimization because it will be able to find optimization steps which the 2-way approach will miss.
- If there is more than two parts in the cycle, the node passing along a cycle should give better improvement per cycle compared to 2-way exchange.
- Larger gain per iteration should lead to faster convergence.

We present the algorithm GreedyCycle in the following sequence. We first present algorithm ProcessCycle which is used to find and process the cyclic node-passing interests. Then we give algorithm GreedyCycle, and its balance requirement, termination condition, and finally its complexity.

4.2.1 Algorithm ProcessCycle

Figure 4.3 gives the algorithm ProcessCycle which finds and processes the cycles in the node interests of the parts. Along with the global data-structures n, k, and $GList$, it uses the following additional data-structures. For the profitable cycle finding and processing, we need to track whether a part is scanned in this iteration and to which cycle it belongs to. For this purpose we use an integer array $color[k]$. To save the interests of parts to pass nodes to other parts, we use another array $xlst[2*k]$.

Let us define *parts-graph* as the directed graph G_p with k nodes where each node corresponds to a part of the original graph G. If part u wants to pass its *going* node to part v, there will be an edge between node u and node v, otherwise, the edge from node u will point to NULL. Therefore, in graph G_p, there will be k nodes, and k edges, one from each node. A node in the parts-graph is called a *part-node*, and an edge is called a *part-edge*, and a cycle is called *parts-cycle*. Such a graph will have the following properties.

- Any node in a cycle cannot have an outgoing edge to a node outside the cycle, because, since each node has only one outgoing edge, all the outgoing edges of the nodes in the cycle must be used up to form the cycle.
- No two cycles can be connected because no cycle can have any outgoing edge.

These two properties give us the algorithm ProcessCycle to find the cycles in parts-graph, and process them accordingly. Initially all the parts are WHITE which means that they are not yet touched by ProcessCycle. We scan the parts from 1 to k (line 1). We find the first part with color WHITE, (say part i), and color it as i. We call this as *pass* i. Then we follow the part-edge from this part, and if the destination is WHITE, we color it as i, and continue following. This is given in loop 3–4. If at any stage, we get a part whose color is not WHITE, we stop following. If this color is i, there must be a parts-cycle starting at part i, and therefore, we pass the appropriate nodes along this cycle (line 5–10). If the color is not i, we reached at a part already cycle-scanned earlier, and we have no new hope there.

```
int n, k;                 // number of nodes, and parts
int D[n];                 // node u has D[u] neighbors
int *GList[n];            // jth neighbor of node u is at GList[u][2*j],
                          // and this edge-cost is at GList[u][2*j+1]
int P[n];                 // node u goes to part P[u]
int color[n];             // node color
int xlst[2*k];            // part i has put its going node at xlst[2*i]
                          // and corresponding destination in xlst[2*i+1]
```

```
procedure ProcessCycle()
{
    for (i = 1; i ≤ k; i++, j = i){
        if (color[i] ≠ WHITE || xlst[2*j+1] < 0) continue;
        while (color[j] == WHITE && xlst[2*j+1] ≥ 0)
            color[j] = i, j = xlst[2*j+1];
        if (color[j] == i && xlst[2*j+1] ≥ 0){
            do {
                P[xlst[2*j]] = xlst[2*j+1];
                j = xlst[2*j+1];
            }while (color[j] == i);
        }
    }
}
```

Figure 4.3: Algorithm ProcessCycle for cycle detection and processing in the parts-graph.

4.2.2 Algorithm GreedyCycle

Figure 4.4 gives algorithm GreedyCycle for graph partitioning using the algorithms GreedyCompute and ProcessCycle. This algorithm also proceeds on iterations in order to improve the partition cost. At each iteration, the algorithm first computes the parts' profitable node passing interests, and then it finds and processes these interests to get a better partition. At the onset of each iteration, we first compute the most promising node offers from each part, and initialize the *color* array to WHITE (line 2–5). For each part i, we compute the most promising node to be transferred to another part, and the corresponding destination using algorithm GreedyCompute, and put it in $xlst[2*i]$ and $xlst[2*i+1]$ respectively (line 3). Now the parts-graph is available in the array $xlst[1:2k]$, and as such we call ProcessCycle to detect all the parts-cycles, and pass the nodes accordingly. After this we calculate the current partition cost (line 19). If we improve the partition-cost in this process, we now restart the iteration. However if the algorithm fails to make any profit in a whole iteration, it terminates (line 8–11).

4.2.3 Balance Constraints

We start algorithm GreedyCycle, with an initially balanced partition. Now in the algorithm, the only way we allow passing nodes among parts, is by cyclic passing in ProcessCycle where each part in the cycle gives exactly one node, and also gets exactly one node. Therefore as the iteration continues the balance of the part sizes is never interrupted.

```
int n, k;                       // number of nodes, and parts
int D[n];                       // node u has D[u] neighbors
int *GList[n];                  // jth neighbor of node u is at GList[u][2*j],
                                // and this edge-cost is at GList[u][2*j+1]
int P[n];                       // node u goes to part P[u]
int cost, oldcost = ∞;          // current cost, old cost
int color[n], terminated = 0;   // node colors, and termination flag
int xlst[2*k];                  // part i puts its going node at xlst[2*i]
                                // and corresponding destination in xlst[2*i+1]


procedure GreedyCycle()
{
    do {
        for (i = 1; i ≤ k; i++){
            GreedyCompute(i, xlst[2*i], xlst[2*i+1]);
            color[i] = WHITE;
        }
        ProcessCycle();
        cost = ComputeCost();
        if (cost ≥ oldost)
            terminated = 1;
        oldcost = cost;
    }while (!terminated);
}
```

Figure 4.4: Algorithm GreedyCycle for k-way graph partitioning.

4.2.4 Termination

Following the same line of logic as applied for algorithm GreedyPass, we can show that algorithm GreedyCycle will also terminate after a finite number of iterations.

4.2.5 Complexity

Let us first see the complexity of algorithm ProcessCycle. Each iteration has two stages — cycle-finding stage (line 3–4) and cycle-processing stage (line 5–11). Since in the cycle-finding stage, we follow a node only if it is WHITE, each part-node is considered once at the cycle-finding stage. Then in the cycle-processing stage, a part-node may be considered at most once more if it is a part of a cycle. When a node is considered for cycle-finding, it takes at most 6 operations, and each time a node is considered for cycle-processing it takes at most 6 operations. Hence, the computational complexity algorithm ProcessCycle is,

$$t_{PCycl} = \mathcal{O}(k).$$

Now we are ready to investigate the complexity of algorithm GreedyCycle. In line 3–6, we do the initial computation, and here the main computation is due to GreedyCompute. Hence line 3–6 has a complexity of $\mathcal{O}(kt_{GComp})$. Line 6 calls algorithm ProcessCycle, and therefore has a cost of $\mathcal{O}(k)$. Finally ComputeCost of line 7 takes $\mathcal{O}(m)$ operations because each graph-edge is considered only once. We assume that algorithm GreedyCycle terminates after r iterations. Then the computational complexity of algorithm GreedyCycle is,

$$t_{GCycl} \quad = \quad r \cdot (\mathcal{O}(kt_{GComp}) + \mathcal{O}(k) + \mathcal{O}(m))$$

$$
\begin{aligned}
&= r \cdot \Big(\mathcal{O}(n + \frac{m}{k}) + \mathcal{O}(k) + \mathcal{O}(m)\Big) \\
&= \mathcal{O}(rkn + rm). \\
\text{Hence, } t_{GCycl} &= \mathcal{O}(rkn + rm).
\end{aligned}
$$

4.3 Discussion

Our algorithms GreedyPass and GreedyCycle have some similarity with Lee's algorithm [25] in that each of these algorithms uses iterative improvement technique for the k-way GP problem.

However, the main heuristics and computations in Lee's algorithm proceed in a completely different way than ours. Some major issues where their algorithm differs from ours are as follows. In their algorithm, they use an adjacency matrix representation for the graph, and before the iterative algorithm starts, they first convert the problem graph into a completely connected graph by some pretransformation. It can start with any initial partition, and as the algorithm proceeds, the node balance among parts is achieved due to the use of the pretransformation and a carefully designed gain-function.

Some of the features which differentiate our algorithms GreedyPass and GreedyCycle from other ones are as follows.

- Algorithms GreedyPass and GreedyCycle are simple, nonhierarchical, greedy and heuristic GP algorithms.
- Since our algorithms use adjacency list format, the per-processor memory requirement is ($\mathcal{O}(2m)$) which is less compared to other methods such as Lee's where adjacency matrix format requires more memory ($\mathcal{O}(n^2)$). However in our distributed implementations, if we use p processors, the total memory requirement is ($\mathcal{O}(2pm)$). The less per-processor memory

requirement in our algorithms is important, because the memory capacity of a system usually comes from a per-processor basis. For example, in our experimental studies, we could not test Lee's algorithm for problem sizes beyond $n = 800$, whereas we did not have any problem to test any of our algorithms for n up to 12800.

- The computational complexity per pass for our algorithms is also less compared to other nongreedy methods such as Lee's. Algorithm GreedyPass has a per-pass complexity of $\mathcal{O}(n + m/k)$, and algorithm GreedyCycle has a per-pass complexity of $\mathcal{O}(kn + m)$, whereas the per-pass complexity of Lee's algorithm is $(\mathcal{O}(kn^2))$. It is worth mentioning that this per-pass complexity comparison is not very fair because there are other issues where the algorithms might perform quite differently such as the gain achieved per pass or the number of passes required to converge. To avoid this problem, the best way is to compare them experimentally. In chapter 5, we give our experimental results, where we show that our simple algorithms give excellent time performance compared to Lee's algorithm.

- The average quality of solutions is comparable to the best available algorithms on this problem.

- The major weakness of these algorithms is that, as they use greedy heuristics, the solution might be stuck at a poor local minimum on special bad instances. However, as we will show later, this does not, in general, effect the average performance of the algorithm on random graphs.

Chapter 5

Experiments and Performance Results

5.1 Implementations

In the preceding two chapters, we have discussed our basic GP algorithms AUCTION, GREEDYPASS, and GREEDYCOMPUTE in full detail including their properties and computational complexities. We also discussed some implementation issues and developed some procedures for this purpose. Now in this section, we briefly discuss the implementations of these algorithms in two kinds of platforms — sequential and distributed. We put these algorithms in Appendix A for two reasons:

- Using the already discussed procedures, derivation of these implementations are more or less simple and straightforward, and we do not need detailed elaborations on them.
- When we talk about the experiments and performance comparisons, referencing is easy if they are all in a row in the Appendix.

Before we start the discussion on individual algorithms, let us put a small note here on the naming conventions used for our implementation algorithms. All the names come from the following abbreviations: S for sequential, D for distributed, A for auction, P for greedy passing, C for greedy cycling, and W for worker. For example, the name of the worker algorithm for our distributed greedy cyclic algorithm will be DCW. Also it requires mentioning that, in a master-worker paradigm, a distributed algorithm means a collection of two algorithms — the master algorithm and the worker algorithm. In our description, we use the same name to refer to a distributed algorithm, and its master algorithm since all the user interfaces to the distributed algorithms are through the master, and this use should not create much confusion because they are used in different contexts. For example, distributed algorithm DA consists of two algorithms, algorithm DA at the master and algorithm DAW at each worker.

Algorithm AUCTION

Using the procedures already developed, implementations of the algorithm AUCTION is simple and straightforward. To port the original master-worker distributed algorithm to the sequential environment, now we have to do all the worker computations in only one process. The easiest approach for this is to do the computations for each worker one after another. Algorithm SA of figure A.1 gives our sequential implementation using this idea.

The distributed implementation DA of algorithm AUCTION is given in figures A.2 and A.3, where the first one gives the algorithm at the master and the second one gives the algorithm at the workers. These algorithms are basically the same as in figure 3.1 except now we are very specific about all the strategies we adopted for our implementation. At the master, we used a mod function for the initial seeds and algorithm DISTRIBUTENODES for distributing nodes. At each worker, we used ComputeBid2 for computing the bid.

Algorithm GreedyPass

Our sequential implementation SP of algorithm GreedyPass is given in figure A.4 where we just added the basic input/output and an initial partition strategy to the basic algorithm of figure 4.2. For the initial partition, we used a simple modular function so that the initial partition becomes balanced. Since the algorithm GreedyPass is inherently sequential, we did not implement it in the distributed environment.

Algorithm GreedyCycle

Once again, the sequential implementation SC is straight-forward, we just add the input/output function, and a simple starting partition generation strategy to algorithm GreedyCycle.

For the distributed implementation DC, we distribute the main computation loop (line 3–6, figure 4.4) of algorithm GreedyCycle to the different worker processes. The master algorithm DC of figure A.6 goes as follows. In every iteration, the master waits to get the node-transfer interest from the workers, and after receiving this information the master then detects and processes all the cycles in the parts-graph. Finally before the start of the next iteration the master broadcasts the new nodes-to-part mapping P in an UPDATE message to all the workers, so that they can compute for the next iteration. When the master detects termination, it sends a TERMINATION message to all the workers.

The worker algorithm DCW of figure A.7 also proceeds in iterations. In each iteration, the worker i computes the transfer-interest ($going$, $dest$) of part i, and sends that to the master. Then it waits for a message from the master. If the master sends an UPDATE message it continues the iteration, else it terminates the loop.

Combination of Basic Algorithms

Since GreedyCycle needs an initial partition, and Auction can directly compute a partition using the input information, it is an interesting idea to use Auction to get the initial partition for GreedyCycle. We implemented this idea in both the sequential and distributed environments. In either environment, as one can easily understand, this implementation simply consists of the union of the operations done in the implementations of Auction and GreedyCycle in that environment. The sequential implementation for the SAC is given in figure A.8, and for the distributed implementation, the master algorithm DAC is given in figure A.9 and the worker algorithm DACW is given in figure A.10.

Random Partition and Lee's Algorithm

To perform the performance studies of our algorithms we compare them with the simple random partitioning and Lee's algorithm. For the random algorithm RAN we used the simple modular function partitioning as we have used as the initial partitions for in SP or SC. For Lee's algorithm we implemented the algorithm of figure 2.1.

Test Graph Generations

Generating the input problem graphs is the first thing required before we start our experiments. For this purpose, we wrote some simple graph utilities like, graph generation in both adjacency matrix format and adjacency list format, and their inter-conversions. To test all the algorithms with the same problem graphs, the conversions were required because Lee's algorithm uses adjacency matrix format, and the rest of the algorithms use adjacency list format. For our graph generation routines, we have to supply two inputs: the number of nodes and the maximum number of edges each node can have. The second parameter allows us to experiment with graphs with different densities. For the edge-weights of the output graph we use the Unix function call *rand*.

5.2 Performance Metrics

An important metric for comparison of algorithms is their memory requirements. We have already mentioned that Lee's algorithm uses adjacency matrix format where the memory requirement is $\mathcal{O}(n^2)$, and all of our algorithms use adjacency list format where the memory requirement is $\mathcal{O}(2m)$. Therefore except for extremely dense graphs, our algorithms require a lot less memory compared to Lee's algorithm. Since this memory requirement advantage of our algorithms is obvious due to the choice of data-structure, we do not further deal with this in our experimental studies.

To compare the experimental performance of the GP algorithms discussed, we use two metrics — the quality of solution and the time of execution.

Quality of Solution

The *quality of a solution* of an algorithm means how good a partition is produced by a certain algorithm. In other words, quality of solution indicates the amount of optimization an algorithm achieves on the partition-cost or cutsize. For this reason, when we compare the quality of solutions given by different algorithms, we can just compare the partition-costs, (denoted by C), produced by them. When algorithm A produces a partition-cost less than that produced by algorithm B, we say that, quality-wise algorithm A performs better than algorithm B.

However we will later see that comparing the partition-costs directly has a technical problem. When we compare quality of solutions for varying graph sizes or densities, the partition-cost also varies with the problem parameters. Now if the partition-cost itself changes very rapidly with the problem parameters, it is very difficult to see the relative performance of the algorithms based on the absolute value of the partition-cost, especially when we try to plot them as a function of graph size or density. The easiest escape from this problem is to compare the relative performance by taking ratio, because even though the absolute value of the partition-costs change rapidly with the problem parameters, the ratio does not. For this reason, we define the *relative quality* of an algorithm as the ratio of the partition-cost produced by that particular algorithm to the partition-cost produced by the random partitioning. In notations, the relative quality of algorithm A is given by,

$$Q(\mathrm{A}) = \frac{\text{Partition-cost produced by algorithm A}}{\text{Partition-cost produced by random partitioning}}$$

Thus a lower $Q(\mathrm{A})$ means better quality.

Time of Execution

The next index we use for comparing GP algorithms is the time of computation, that is how much time an algorithm needs to produce its result. For our experiments we use the Unix */bin/time* command to measure the time, and for any algorithm, we consider the *real time* between the invocation of the command to the production of output, which we denote by symbol T. We take the real time because this is the wall-clock time a user sees the algorithm to take from the outside world. Usually we want an algorithm to produce the output in as small a time as possible. We also use the *relative time* of execution as an index to compare the different algorithms. In notations, relative time of algorithm A is given by,

$$Tr(\mathrm{A}) = \frac{\text{Real time required by algorithm A}}{\text{Real time required by random partitioning}}$$

A lower $Tr(\mathrm{A})$ means faster execution.

5.3 Experimental Results

We have implemented all the above mentioned algorithms in an Ethernet local area network of 28 SUN4[1] workstations running SUN OS version 4.1.3. For our distributed algorithms we used the PVM 3 environment. We have run many experiments. All the results presented in the next few sections show the representative performances. All the experiments were repeated several times, (4 to 10 depending on the network load they produced), and the average results were accepted. For the sequential algorithms, we have generated ten random graphs for each set of parameters, and then partitioned all of them. In most

[1]SUN is a registered trademark of Sun Microsystems Incorporated.

of the cases the ten results were within 6% of the average value, and therefore we used the average of these results for our plots. However for the distributed algorithms, we used only one random graph for each set of parameters.

One small note about the size of the problem is important to mention here. The upper limit of the problem size we could experiment with an algorithm was mainly determined by its memory requirements. Since Lee's algorithm used the adjacency matrix format for the graph, the highest size of graph we experimented without any problem was limited to $n = 800$. However for all our algorithms, sequential, or distributed, we used the adjacency list format, and therefore for low degree (typically around d=10-20) of the graph nodes we could easily go up to $n = 12800$.

5.3.1 Quality of Solution

Figures 5.1–5.3 give the comparison of the quality of solutions among all our algorithms and Lee's algorithm along with the random partitioning.

Figure 5.1 gives the comparison of the algorithms' quality as a function of n, and figure 5.2 compares them as a function of k. In figure 5.3 we see the effect of change of the maximum node degree on the quality of solutions. The key features of these results are summarized in the following observations and explanations.

- Our distributed algorithms give the same partition-costs as their sequential counterparts. This is very logical because the distributed algorithms actually do the same kind of computations on the same data as their sequential counterparts. The only specialty with the distributed algorithms is that the computations may be distributed to be done at different processors for time-parallelism.

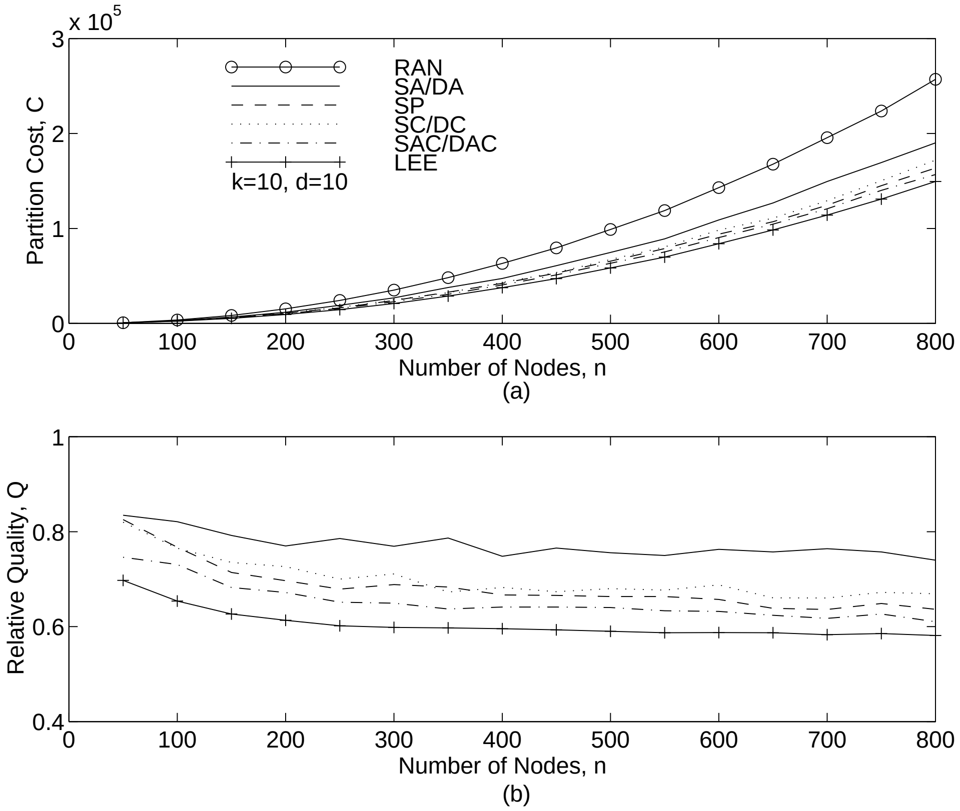

Figure 5.1: Comparison of solution qualities given by our algorithms with that of the Lee's algorithm as a function of n.

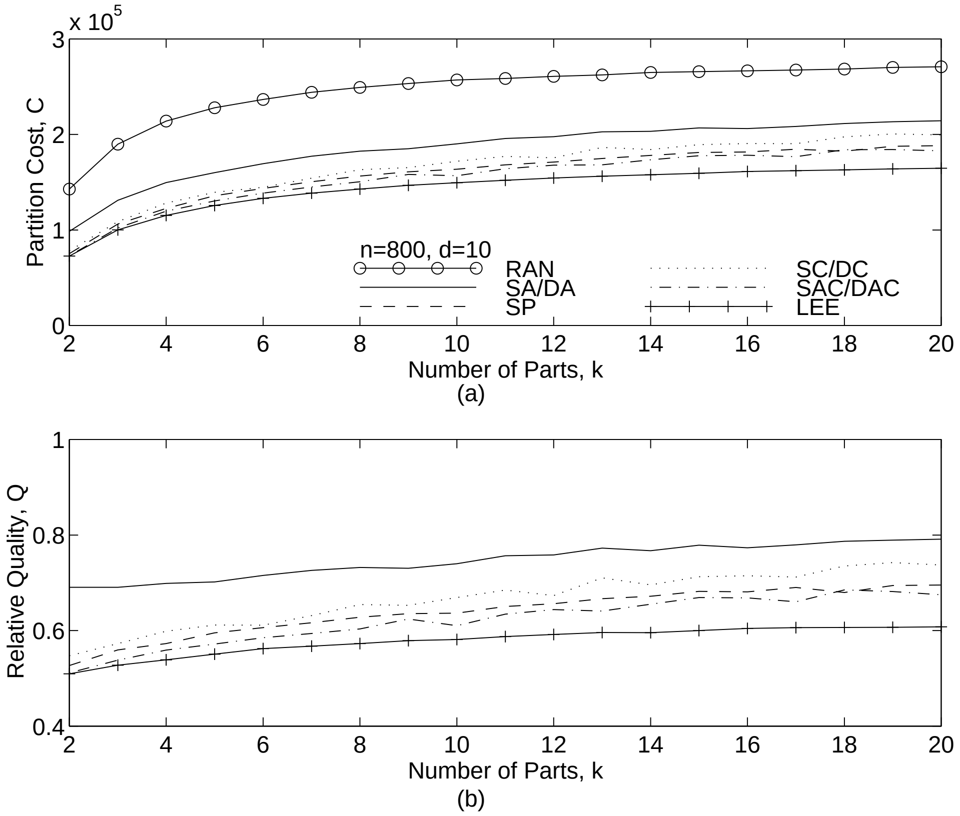

Figure 5.2: Comparison of solution qualities given by our algorithms with that of the Lee's algorithm as a function of k.

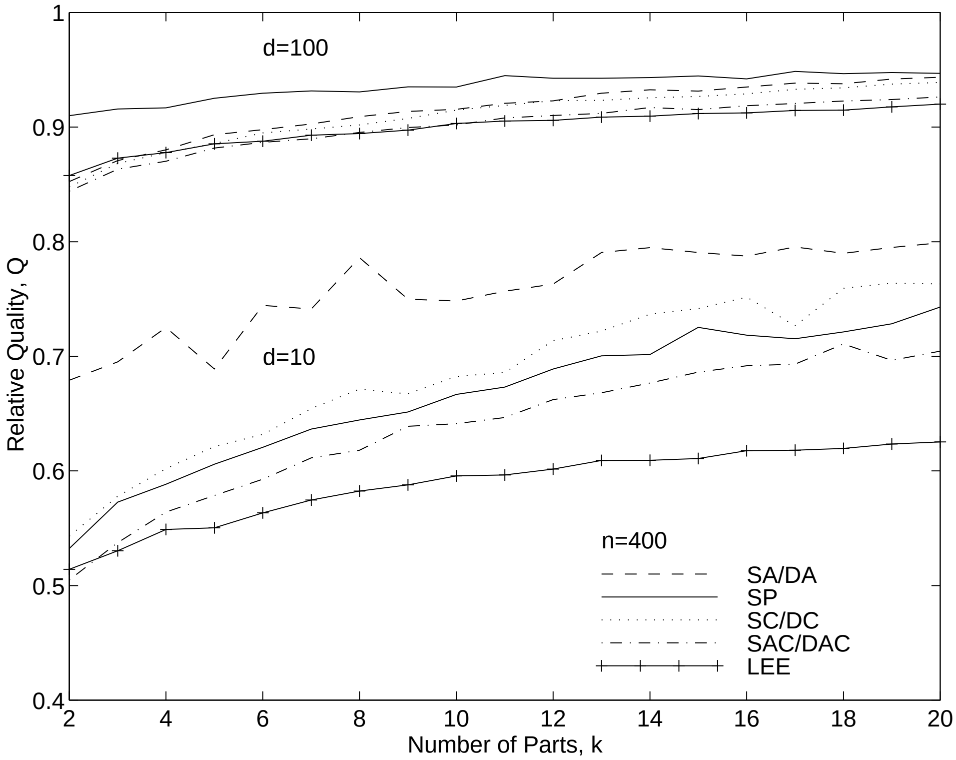

Figure 5.3: The effect of maximum node degree (d) on the quality of solution.

- All the algorithms we have tested can be arranged in descending quality of output as follows: Lee's algorithm, then SAC/DAC, then SP and SC/DC, and finally SA/DA.

  Lee's algorithm gives the best partitioning in most of the cases. The reason for this is that it is a non-greedy iterative algorithm where in each iteration, all nodes are moved one at a time generating n possible partitions, and finally the partition with the lowest partition-cost is accepted. This requires a lot more computations than what we require in our algorithms.

  SA and DA perform the worst which fits well with the knowledge that direct algorithms in general give worse quality compared to the iterative algorithms. However for some applications even this quality can be acceptable, especially when we remember the amount of computations required for this. We justify this point in the next section.

  Among our algorithms, SAC and DAC perform the best, and on average, their quality is within 8% of that achieved by Lee's algorithm. Even, in some cases of our results, SAC/DAC gave better partition-cost than Lee's algorithm, especially at low k values for high d (figure 5.3). The reason why SAC/DAC gives the best quality among our algorithms is understandable. They combine both the ideas of direct partitioning and iterative improvement: initial partition using auction, and then cyclic iteration to improve on the partition-cost.

  Algorithm SP and SC generally produce similar partition quality, which is close to SAC/DAC, and better than SA/DA.

- In the figures 5.1–5.3, we note that, the basic pattern of the variation of partition-cost is the same for all the algorithms, and the absolute magnitude of this is mainly dominated by the problem parameters, such as graph size n, number of parts k and the maximum node degree d. Figure 5.1(a) shows how the partition-costs vary with n. The partition-cost for all the algorithms increases rapidly with n mainly because, increase of n increases the number of edges rapidly. Figure 5.2(a) shows the variations of partitions-costs with k. When n and d are kept constant, the number of edges increases with the increase of k, because now more and more edges are cut by the partitioning. This is revealed in figure 5.2. The dependence of the partition-cost on factors other than the algorithms' characteristics justify our use of relative quality index for the comparison of the actual algorithmic performances.

- One important observation in our study is that, all the GP algorithms we implemented show the common characteristic that, they give good quality relative to random partitioning only when the graph is either sparse or moderately dense. This is revealed in figure 5.3. In this graph, we see that with the same problem size ($n = 400$), and the same variation of k, all the algorithms perform significantly better when each node has a maximum node degree of $d = 10$ compared to a higher $d = 100$.

  We believe that this characteristic comes from the inherent nature of the problem, and it has little to do with the algorithm we use to solve it. Here is one possible explanation for this. When d is low, the total number of edges is small, and it is possible for an algorithm to accommodate most of the edges inside the parts and hence minimize the partition-cost. On the other hand, when d is high, the number of edges also becomes high, and the balancing requirement compels any algorithm to have a lot of cross-edges among the parts. This is because, if a node has connections to many other nodes, there is no way it can have all its neighbors in the same part. For example, let us consider the extreme case in this line, that is the fully connected graph with equal weights on all the edges. No algorithm can now have a better partition than the random partitioning, because there does not even exist any better partition.

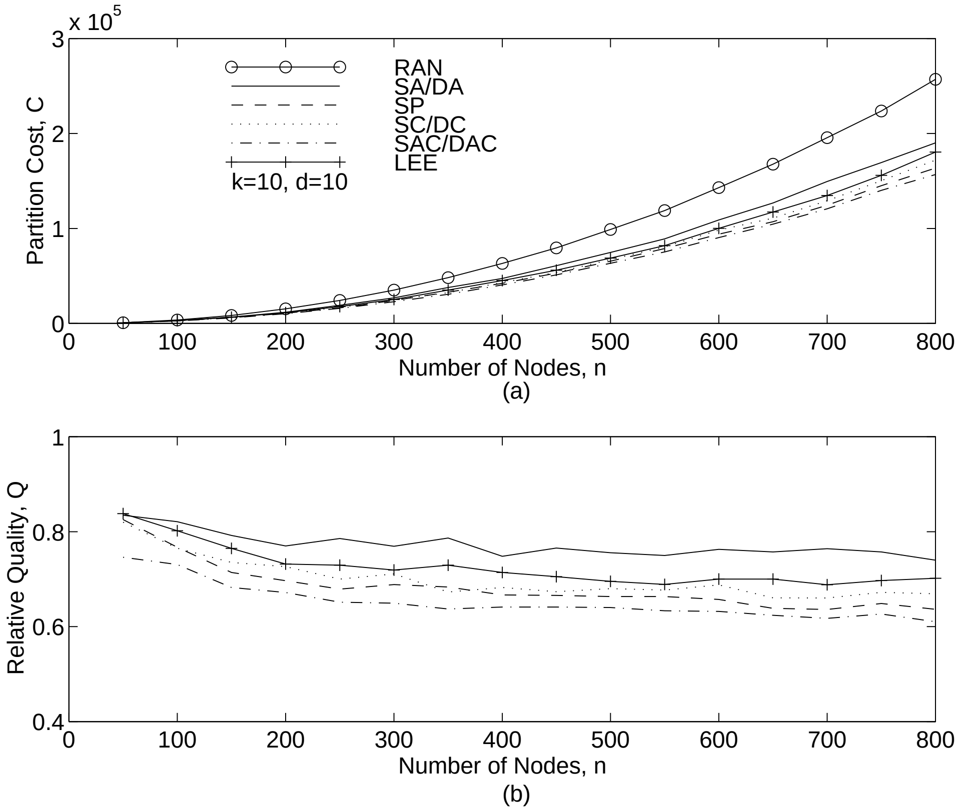

Figure 5.4: Comparison of solution qualities given by our algorithms with that of the original Lee's algorithm as a function of n.

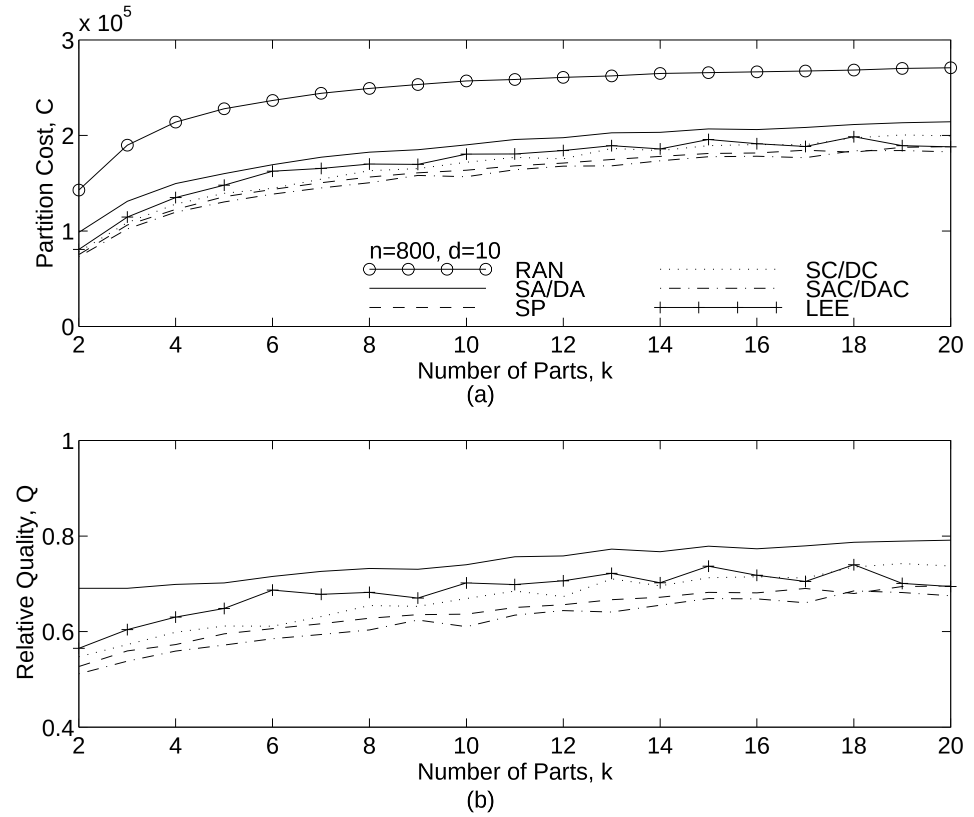

Figure 5.5: Comparison of solution qualities given by our algorithms with that of the original Lee's algorithm as a function of k.

Until now we have compared our algorithms' solution quality with that of the modified Lee's algorithm. We have also implemented the original Lee's algorithm, and figures 5.4-5.5 show the quality comparison results. We see in most of the cases all of our iterative algorithms significantly outperform the original Lee's in quality of solution, and our AUCTION algorithms also give solution quality which is close to that of Lee's algorithm. The improved solution quality of our algorithms compared to the original Lee's comes mainly from the bad initial partition of the original algorithm where at the start all the nodes are put into a single part. This justifies the importance of the small modification we made for Lee's algorithm.

5.3.2 Time of Execution

In this section, we present the experimental results on the time performance of our algorithms as well as that of Lee's algorithm and random partitioning. Since the time performance of the distributed algorithms is much different compared to their sequential counterparts, we present the results in two parts. At first, we compare our sequential algorithms with Lee's algorithm, and finally we compare our distributed algorithms with our sequential algorithms. We hereby mention that, in all our time comparisons we used the modified Lee's algorithm although some tests showed that it would not have made much difference if we used the original Lee's algorithm instead.

Lee's algorithm vs. Our Sequential Algorithms

The most important contribution of this work is the excellent time performance of our simple algorithms compared to Lee's algorithm, one of the best algorithms of its kind, without compromising much of the solution quality. The time performance of Lee's algorithm and our sequential algorithms, (SA, SP,

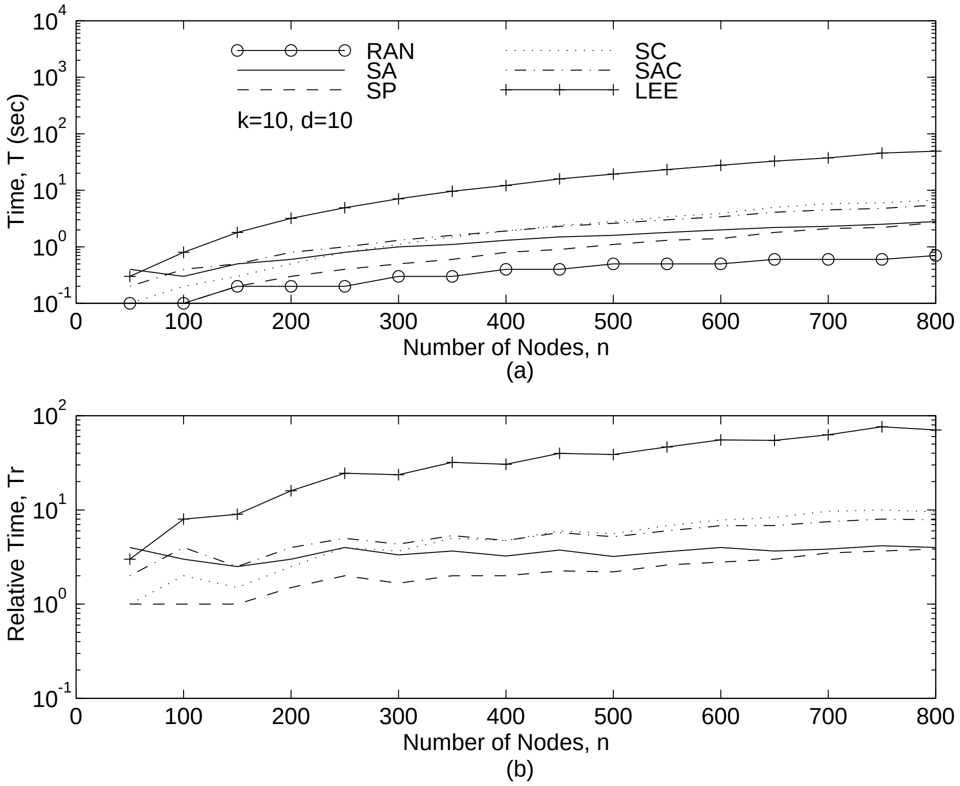

Figure 5.6: Comparison of execution times of our sequential algorithms with Lee's algorithm as a function of n.

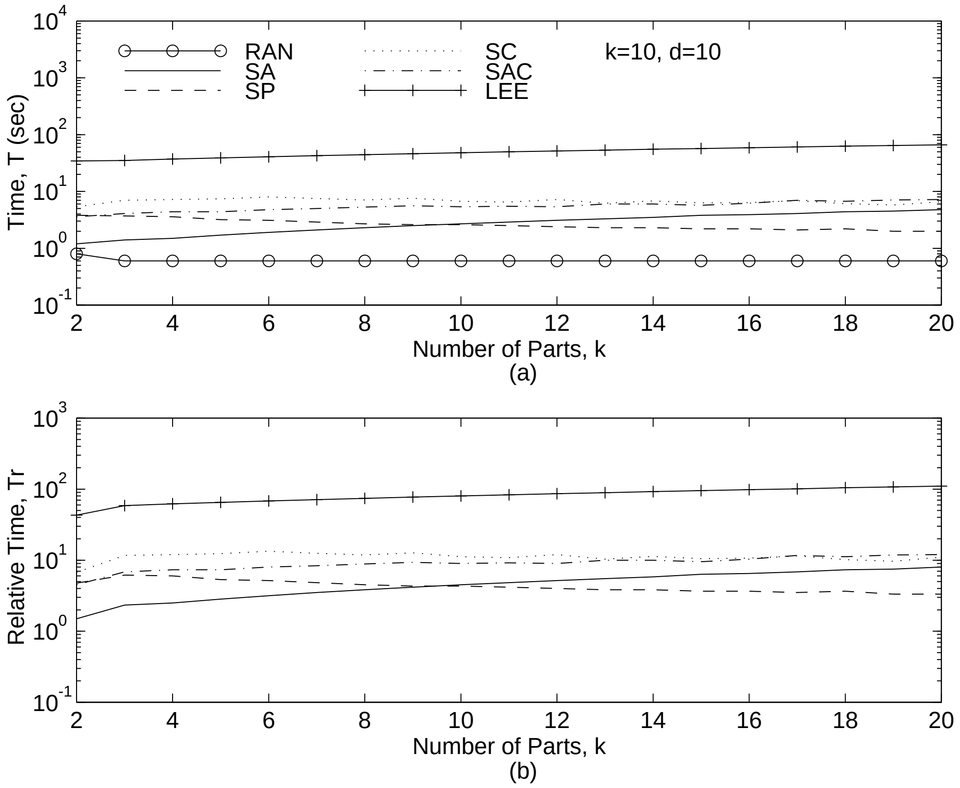

Figure 5.7: Comparison of execution times of our sequential algorithms with Lee's algorithm as a function of number of parts k.

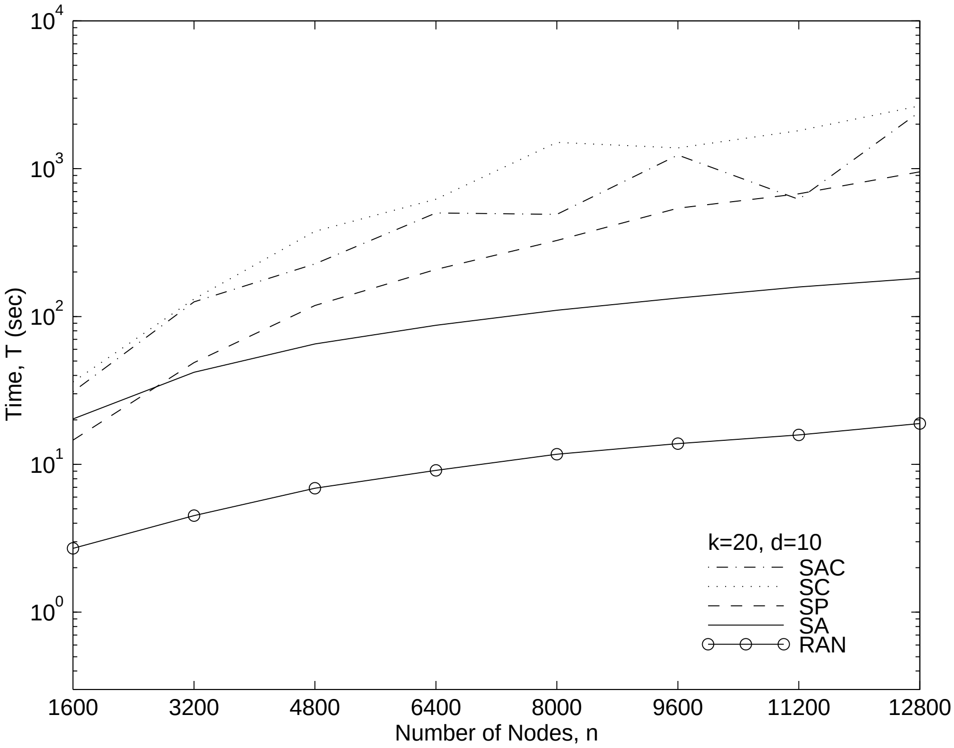

Figure 5.8: Comparison of execution times among our sequential algorithms as a function of n. Here we see variation of n up to 12800, which we could not go in figure 5.6 because Lee's algorithm could not be tested beyond $n = 800$.

SC, and SAC), are shown in figures 5.6–5.7. Figure 5.6 compares the execution times as a function of the problem size (n), and figure 5.7 compares them as a function of number of parts (k). We note that all our algorithms perform a lot faster compared to Lee's algorithm. For example, as shown in figure 5.6(a), for $n = 400$, $k = 10$ and $d = 10$, algorithm SP took 1.8 seconds, algorithm SA took 2.5 seconds, SC took 3.8 seconds, and algorithm SAC took 3.5 seconds as compared to 25.4 seconds taken by Lee's algorithm. Compared to Lee's algorithm, SP is 14 times faster, SA is 10 times faster, and SC and SAC are 7 times faster. This difference even increases with the increase of problem size. Figure 5.8 gives the comparison of execution times among our sequential algorithms as a function of n for a larger range (up to 12800) than figure 5.6. Lee's algorithm is not shown here because for memory limitation it could not be tested beyond $n = 800$.

We now explain some of the other key observations in these results.

- If we look into the results of figures 5.6-5.8 carefully, we note that the time-wise champion among our algorithms is not always the same, it rather changes with the problem parameters. In some cases, SA performs better, and at other cases SP or SC performs better. Specifically, for small n/k ratio, SP or SC performs better and for large n/k ratio (large n or small k), SA performs better. This can be explained as follows. For the same input graph (i.e., same n and d), if k is small, there are more nodes per part, and hence each iteration of SP takes more time. On the other hand, if k is large SA takes longer time because the main computation in SA, (i.e., bid computation and node distribution), is directly proportional to k. This property is clearly seen in figure 5.6. For $k < 10$, SA performs better, and for $k > 10$ SP performs better. For the same reason, for fixed k and d, as n increases SA tends to perform better than SP or SC. This is clearly seen in figure 5.8. Here up to n=2800, SP performs better, and for $n > 2800$, SA performs better.

- When we compare the execution times of SC and SAC, we see a similar relation. For small n/k ratio, SC does better, and for large n/k ratio, SAC does better. This is because in SAC, where we use AUCTION followed by GREEDYCYCLE, the computation time of the first stage dominates the overall time. This characteristic is clearly observed in figures 5.6-5.8.

Sequential vs. Distributed Algorithms

Let us now show the performance of our distributed algorithms compared to the sequential algorithms. The main motivation in our distributed algorithms is to distribute the total computation onto different processors to achieve parallelism, and thereby to reduce the overall execution time. However a distributed algorithm has its own extra cost. To coordinate the works done by different processors, distributed computation requires two types of overheads:

- communication cost to exchange information among different parts of the computation, and
- synchronization and consistency cost where some parts of computation wait for other parts' results or when some computation has to roll back to return to some consistent state.

Thus to get some real performance improvement using distributed algorithms the ratio of overhead to actual computation must be kept within a certain limit so that the overhead does not surpass the computation gain expected from computational parallelism. Some general ideas for getting good performance in distributed algorithms are given below:

1. Distribute the works onto the processors according to their computational capacity.

2. Minimize the per message communication costs, and utilize the capabilities of the communication network.

3. Minimize the number of messages required.

4. Minimize the synchronization points.

5. Try to avoid consistency problems.

Among the above issues, generally the first two are taken care of by the distributed system designers, and the final three are the main concern of the application designers.

From the preceding discussions, we understand that the performance of a particular distributed algorithm is highly dependent on its implementation environment. The point we want to make here is that once we have accepted PVM as the environment for our distributed algorithms, the practical performance is now very much dependent on the characteristics and performance of PVM.

We have done some experiments to compare the time performance of our distributed implementations with the corresponding sequential implementations. Our experimental results in a 21 workstation ($p = 21$) configuration are presented in figure 5.9 in a speedup vs. number of parts coordinate. We note that only the algorithms DA and DAC gave some positive speedup over the sequential algorithms. With 21 processors our DA algorithm achieved a maximum speedup of up to 4 which is very encouraging. Algorithm DAC gave a speedup of up to 2.2. We also note that in this range of study algorithm DC did not give any positive speedup.

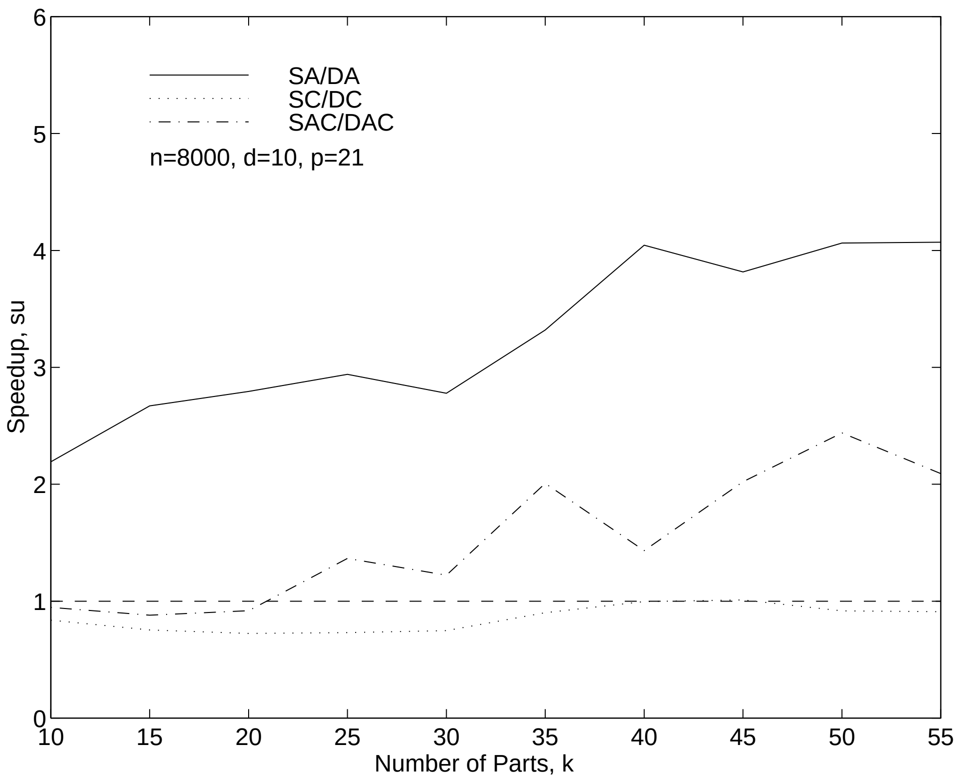

Figure 5.9: The experimental speedups of our distributed algorithms compared to their sequential counterparts as a function of number of parts k for 21 processors ($p = 21$).

In the problem domain we performed our studies, given by ranges of n, k, and d, communication in PVM proved very costly compared to the computation cost. For this reason communication requirement of an algorithm came out to be the dominant performance issue. This explains the speedup curve of figure 5.9 as follows.

- Algorithm DA requires only two stages of communication: first the master broadcasts the problem to the workers, and finally the workers send the bids to the master. This is why algorithm DA performs the best in the PVM environment.
- Algorithm DC requires a broadcast from the master and a message from each worker in every pass. This is to be too much communication in PVM to get any positive speedup.
- Communication-wise algorithm DAC is a good compromise between DA and DC. We still need a broadcast from the master and sending from each worker in every pass, but the number of passes that we require in DAC is expected to be a lot less than that of DC, because DAC starts with a better initial partition given by AUCTION.

Since we have got some significant speedups with algorithm DA, we now concentrate on some of its performance characteristics in greater detail. Figure 5.10 gives a representative pattern of time comparison of DA and SA as a function of k for 4 processors. We note the following observations:

- For k up to 6, SA performs better and after that DA performs better. For SA computation time is proportional to k, since a single process does all the works of k workers. On the other hand, execution time of DA

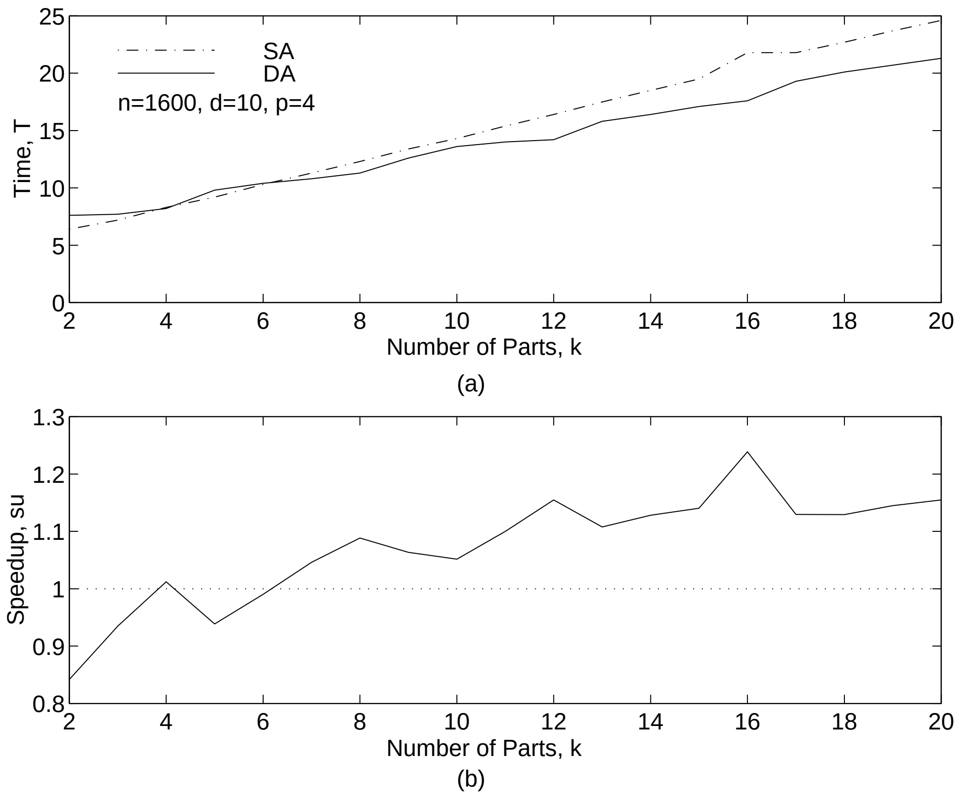

Figure 5.10: Performance of algorithm DA and SA: (a) comparison of execution times and (b) speedup as a function of number of parts k using 4 workstations ($p = 4$).

increases mainly because the communication costs increase with the number of worker processes. Initially at low k, communication overhead for DA is such that it is outperformed by SA. But since the growth rate of communication cost of DA is less than that of the computation cost of SA, soon DA takes over, and starts performing better than SA after k=6. For this reason we see a break-even point at $k = 6$, where both the algorithms take the same execution time.

- The time complexity of SA grows linearly with k which is clear in figure 5.10. The small deviations from linearity in the performance graph comes from the test graph structure. We remember that these test graphs were generated using random numbers for node degrees.

- We see some stair-case kind of jumps in the time graph for DA in figure 5.10(a), and its effect is even more clear in the speedup curve of figure 5.10(b). We note that these jumps in time curve are seen at the k-axis transitions of 4-5, 8-9, 12-13, and 16-17. To explain this let us consider one instance, that is the transition 12-13. Now when $k = 9 - 12$, there are k workers, and when the load is distributed evenly among the 4 processors each processor gets at most 3 workers each. So we do not expect any jump within $k = 9 - 12$. But now if we increase k to 13, we need at least one processor who will get 4 workers, and resulting in a 25% more computation time which explains the jump at $k = 12 - 13$. Another interesting observation here is that each of these jumps comes mainly from the computation time of an additional worker. If we have $(k + 1)$-processors for the computation, then this jump-time would have been the main computation time for each worker; and now if there were no communication and synchronization overhead, this would have been the dominant part of the time required for the whole algorithm DA!

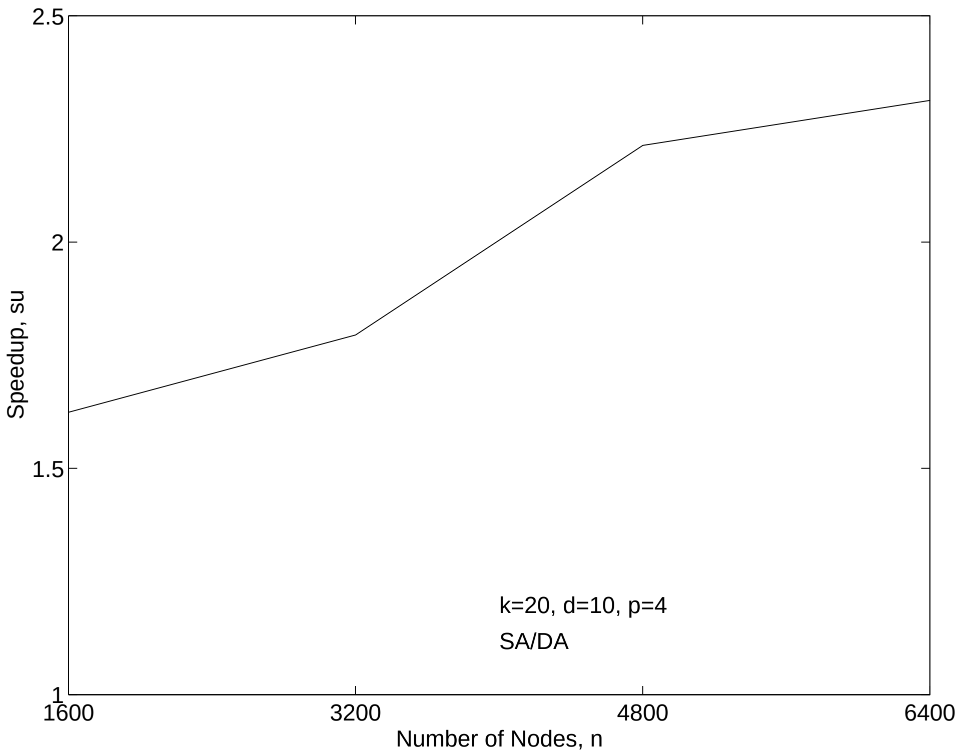

Figure 5.11: The experimental speedup of algorithm DA compared to algorithm SA as a function of number of nodes n.

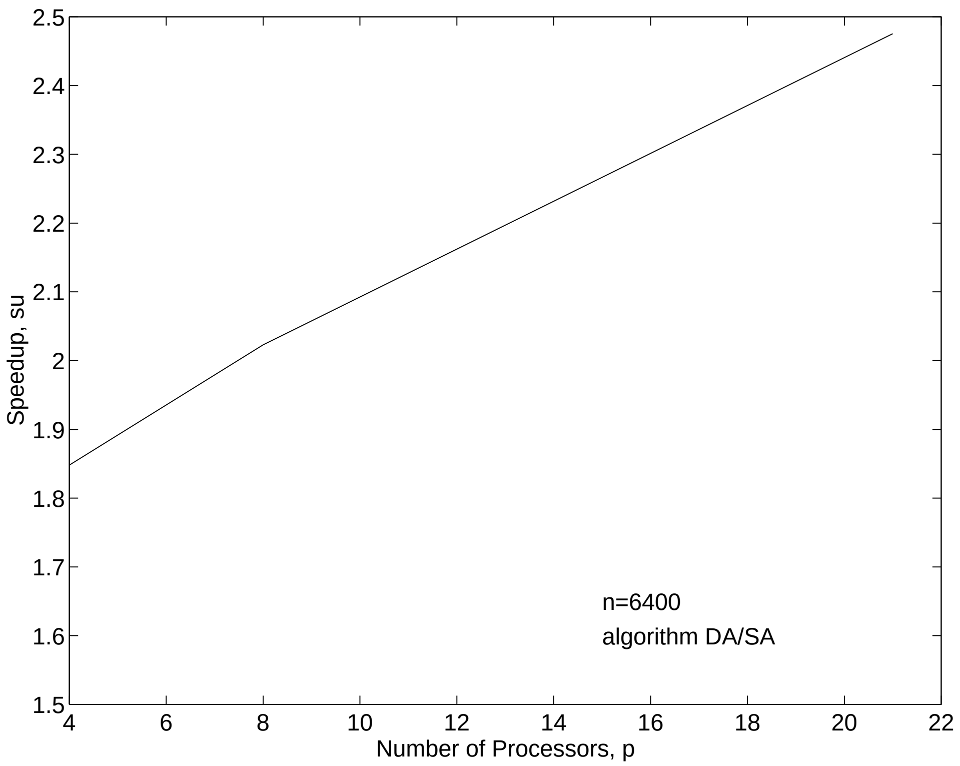

Figure 5.12: Variation of speedup of algorithm DA compared to algorithm SA as a function of number of processors p.

Figure 5.11 gives the variation of speedup obtained in algorithm DA as it varies with the number of nodes n. When n increases, both computation time and communication time increase. Initially, up to a certain value of n, computation time increases much faster than the communication overhead, and therefore speedup increases with n. However after a certain point, communication overhead increases exponentially for the following reason: as the message size increases the number of packets increases; these packets compete among themselves to access the communication medium, and as such the number of transmission failures increases. For this reason, after a certain value of n we get a speedup saturation characteristic as revealed in figure 5.11. This saturation seems very natural because even if there was no distribution overhead we cannot expect a speedup of more than 4 using 4 processors.

In figure 5.12, we plotted the variation of maximum speedup against the number of processors. To get the maximum speedup, for each value of p, we have found speedups for different values of k, and accepted the maximum of them. We see the expected pattern that maximum speedup increases almost linearly with the number of processors. However, since we could not do much to reduce the distribution overhead the slope of this curve is much less than unity. We believe that if the communication support used for the distributed algorithms could have been re-programmed to utilize the capability of the network efficiently, we could get a steeper slope in the speedup line. However due to time limitation, we leave this work for a future project.

5.4 Discussion

Table 5.1 compares some theoretical and experimental features of our algorithms with Lee's algorithm.

Table 5.1: Comparison of our algorithms with Lee's algorithm

Features	LEE	AUCTION	GREEDYPASS	GREEDYCYCLE
Theoretical issues				
Algorithm type	Iterative, Nongreedy	Direct	Iterative, Greedy	Iterative, Greedy
Part size balancing	± 1	± 1	± 2	± 1
Memory	$\mathcal{O}(n^2)^\bullet$	$\mathcal{O}(2m)$ $[\mathcal{O}(2pm)]^*$	$\mathcal{O}(2m)$	$\mathcal{O}(2m)$ $[\mathcal{O}(2pm)]$
Comp. cost	$\mathcal{O}(r_1 kn^2)$	$\mathcal{O}(kn \lg n + km)$ $[\mathcal{O}(n \lg n + m + kn)]$	$\mathcal{O}(r_2(n + \frac{m}{k}))$	$\mathcal{O}(r_3(kn + m))$ $[\mathcal{O}(r_3(k + n + m))]$
Typical experimental performance against Lee's				
Partition quality	—	Within 15%	Within 8%	Within 10%$^\diamond$
Overal speed	—	10 times faster	7 times faster	7 times faster

$^\bullet$For high memory requirement, Lee's algorithm could not be tested for $n > 800$.
*Values in the braket [] relate to distributed implementations.
$^\diamond$AUCTION+GREEDYCYCLE gives quality within 4%.

For practical applications, the above algorithms can be characterized by two main features: solution quality given by partition-cost and time of execution. For practical applications both features are important, however their relative significance varies depending on the application requirements. Our results show that our algorithms give excellent time performance compared to Lee's algorithm, and give reasonable quality of solutions. On the other hand, Lee's algorithm gives better quality of solutions but requiring about an order more execution time of execution. Hence there is a quality-time trade-off for an application designer to choose from these algorithms. For some applications quality of solution may be more important, for some others time of execution may be more critical.

As an example, we consider the multiprocessor task assignment problem. In its simplest form of this problem, the problem graph models a parallel or distributed program execution where the nodes represent the tasks, and the edges represent the amount of communications between the tasks. The problem is to have a mapping of the tasks to k processors such that the amount of inter-processor communication is minimized. For this task assignment problem, both execution time and solution quality are important. The execution time is an overhead which we want to minimize, and solution quality directly affects the amount of communication required, which indirectly affects the communication load and delays. In this trade-off there are two extremes. On one hand, a very good quality with much longer execution time using algorithms such as Lee's will cause too much overhead for the assignment part. On the other hand, random partitioning gives very poor quality but in a very short time, which means much more network load and communication delay. However, since our algorithms give reasonably good quality within a reasonably short time, they should provide a good quality-time trade-off for the designer of such an application. This discussion is also applicable for real-time scheduling or any other time critical application.

On the other hand, for some other applications, like VLSI circuit partitioning and placement, waiting for a long time for better quality of solutions might be worthwhile because in this case partition-quality might directly affect the per unit fabrication cost for a chip. For these applications Lee's algorithm is more suitable. However, even for these applications, during the early design-and-modify stage a fast partitioning algorithm might save a lot of person-hours, and then during the final few stages, one can always obtain a good quality solution using an algorithm like Lee's.

Chapter 6

Conclusion

In this thesis, we have proposed three simple algorithms, AUCTION, GREEDY-PASS, and GREEDYCYCLE, for the k-way graph partitioning problem. Among these the first one is a direct algorithm, and the last two are iterative algorithms. We have described the different features of these algorithms including the implementation issues. We implemented all the algorithms in the sequential Unix environment, and the first and third algorithms were also implemented in the distributed PVM environment. We have also implemented Lee's algorithm for this problem, and executed detailed performance studies comparing the partition qualities and execution times of the different implemented algorithms. One of the main results of this study is that sophisticated algorithms are not always the best choice for the k-way GP problem, even simple algorithms like ours can give very fast performance without compromising much in the solution quality.

6.1 Contributions

We have made the following contributions in this thesis:

- Our algorithm AUCTION introduces a completely new idea for direct graph partitioning using master-worker auction. The main computation here is inherently distributed which can also have efficient implementations in sequential systems. It may be a very good choice for time-constrained applications like multiprocessor task allocations or real-time scheduling.

- Algorithm GREEDYCYCLE introduces the idea of cyclic node exchange among the parts for k-way iterative graph partitioning. Cyclic node exchange is an intuitive k-way generalization of the profitable node exchange between two parts found in KL-method.

- Algorithm GREEDYPASS is a simple and fast greedy algorithm for k-way graph partitioning. It is an inherently sequential algorithm, and therefore is not suitable for distributed implementation.

- Our distributed algorithms (DA, DC, and DAC) comprise the first set of distributed algorithms for the GP problem. We have also given implementations and experimental performances of these distributed algorithms.

- Our experiments show that our simple GP algorithms (both sequential and distributed) give excellent time performance and reasonable solution quality compared to Lee's algorithm, one of the best algorithms for this problem available in the literature.

6.2 Future Work

In this project, we have worked with some simple and fast graph partitioning algorithms for both sequential and distributed environments. There are still many theoretical and practical issues which will be interesting to explore in some future projects. Some of these issues are mentioned below.

Incremental AUCTION. In algorithm AUCTION, the workers continue putting bid numbers on the nodes using an expanding core heuristic with the belief that it will get the low numbered nodes. Now let us consider the scenario, where an worker has put a priority on node x with the belief that it will get node y (that is y is inside the core when it puts priority to node x), and due to some race with other workers this worker does not get node y. In this circumstance, the priority this worker has put on node x is weakened in justification, which is a weakness in our current implementation.

One idea to avoid this problem may be as follows. The algorithm will run in passes. At each pass, each worker will bid for only one node, and the master will distribute only those nodes for which one or more workers have put a bid. This is continued until there is no nodes left undistributed. This algorithm should give partition cost with better quality at the cost of increased communication cost. This seems to be a very interesting extension of work for this algorithm. It is also possible that a compromise of the incremental AUCTION and the one we implemented might be even more promising, because the formal one should give better quality, and the latter one should give faster execution.

Better seed finding strategy. In our implementations, the k seeds were chosen randomly from the graph. Further investigations can be done to find some heuristics which will give 'better' seeds that will lead to lower partition cost.

Any profit greedy cycle. Algorithm GREEDYCOMPUTE now gives the most profitable *going* node. This can be modified as follows. Algorithm GREEDYCOMPUTE returns all the profitable nodes and the corresponding destinations of a part. Then algorithm PROCESSCYCLE finds all the cycles in the parts-graph and pass the nodes along these cycles. We expect that this modification will make the per-iteration time complexity in order to increase partition-cost improvement per-iteration. The actual characteristics of these modifications can only be determined by further investigations in this direction.

Implement with efficient communication. It will be an interesting project to implement the distributed algorithms using more direct and efficient communications. This should significantly improve the performance of the distributed algorithms compared to the sequential algorithms.

Applications. The field-performance of the algorithms may be tested by some real-world application programming. Distributed systems task allocation and VLSI circuit partitioning can be two interesting applications. We are also planning to use these algorithms in the design of some distributed graph algorithms using the divide-and-conquer strategy.

Bibliography

[1] Thomas Lengauer. *Combinatorial algorithms for Integrated Circuit Layout.* B.G. Teubner and John Wiley & Sons, 1990.

[2] Brian W. Kernighan and S. Lin. An efficient heuristic procedure for partitioning graphs. *AT&T Bell Labs. Tech. J.*, 49:291–307, February 1970.

[3] S. Kirkpatrick, C.D. Gelatt, and M.P. Vecchi. Optimization by simulated annealing. *Science*, 220(4598):671–680, May 1983.

[4] C.M. Fiduccia and R.M. Mattheyses. A linear-time heuristics for improving network partitions. In *Proc. 19th IEEE Design Automation Conference*, pages 175–181, 1982.

[5] T. Bui, C. Heigham, C. Jones, and F. Leighton. Improving the performance of the Kernighan-Lin and simulated annealing graph bisection algorithms. In *Proc. 26th ACM/IEEE Design Automation Conference*, pages 775–778, 1989.

[6] Vivek Sarkar. *Partitioning and Scheduling Parallel Programs for Multiprocessors.* The MIT Press, 1989.

[7] Shahid H. Bokhari. *Assignment Problem in Parallel and Distributed Computing.* Kluwer Academic Publishers, Norwell, Massachusetts, 1987.

[8] V. M. Lo. Heuristic algorithms for task assignment in distributed systems. *IEEE Transactions on Computers*, C-37(11):1384–1397, November 1988.

[9] S. H. Bokhari. Partitioning problems in parallel, pipelined and distributed computing. *IEEE Transactions on Computers*, C-37(1):48–57, January 1988.

[10] Harold S. Stone. Multiprocessor scheduling with the aid of network flow algorithms. *IEEE Transactions on Software Engineering*, SE-3(1):85–93, January 1977.

[11] Harold S. Stone and S. H. Bokhari. Control of distributed processes. *IEEE Computer*, 11(7):97–106, July 1978.

[12] Kai Hwang and Jian Xu. Efficient allocation of partitioned program modules in a message-passing multicomputer. In *Proc. ISSM International Conf. on Paral. and Distr. Compu. and Syste.*, pages 226–239, 1990.

[13] W. Hohberg and R. Reischuk. Decomposition of graphs – an approach for the design of fast sequential and parallel algorithms on graphs. Technical report, Fb. Informatik, Technische Hochschule, Darmstadt, 1989.

[14] J.R. Gilbert and E. Zmijewski. A parallel graph partitioning for a message-passing multiprocessor. Technical Report TR 87-803, Computer Science Department, Cornell University, Ithaca, N.Y., 1987.

[15] T.A Feo and M. Khellaf. A class of bounded approximation algorithms for graph partitioning. *Networks*, 20:181–195, 1990.

[16] T.H. Cormen, C.E. Leiserson, and R.L. Rivest. *Introductions to Algorithms.* The MIT Press and McGraw-Hill, 1990.

[17] R. Gould. *Graph Theory.* Benjamin/Cummings, Menlo Park, CA, 1988.

[18] Eugene L. Lawler. *Combinatorial Optimization: Networks and Matroids.* Holt, Rinehart & Winston, N.Y., 1976.

[19] A.S. Tanenbaum. *Modern Operating Systems.* Prentice-Hall, Englewood Cliffs, NJ, 1992.

[20] M.R. Garey and D.S. Johnson. *Computers and Intractability: A Guide to the Theory of NP-Completeness.* W.H. Freeman, San Fransisco, 1979.

[21] Edith Cohen and Lichael Tarsi. *np*-completeness of graph decomposition problems. *Journal of Complexity*, 7(2):200–, June 1991.

[22] B. Krisnamurthy. An improved min-cut algorithm for partitioning VLSI networks. *IEEE Transactions on Computers*, C-33(5), 1984.

[23] D.S. Johnson, C.R. Aragon, L.A. McGeoch, and C. Schevon. Optimization by simulated annealing: An experimental evaluation; part i, Graph partitioning. *Operations Research*, 37(6):865–92, November–December 1989.

[24] L.A. Sanchis. Multi-way network partitioning. *IEEE Transactions on Computers*, C-38(1):1384–1397, January 1989.

[25] Cheol-Hoon Lee, Myunghwan Kim, and Chan-Ik Park. An efficient *k*-way graph partitioning algorithm for task allocation in parallel computing systems. In *Proc. 1st Int. Conf. in Syst. Integr.*, pages 748–751, 1990.

[26] A. Casotto and A. Sangiovanni-Vincentilli. Placement of standard cells using simulated annealing on the connection machine. In *ICCAD*, pages 350–453, November 1987.

[27] C. Wong and R. Fiebrich. Simulated annealing-based circuit placement on the connection machine. In *Proceedings of the International Conference on Computer Design*, pages 78–82, 1987.

[28] P. Banerjee, M.H. Jones, and J.S. Sargent. Parallel simulated annealing algorithms for cell placement on hypercube multiprocessors. *IEEE Transactions on Parallel and Distributed Systems*, PDS-1(1):91–106, January 1990.

[29] John E. Savage and Markus G. Wloka. Parallelism in graph partitioning. *Journal of Parallel and Distributed Computing*, 13:257–272, November 1991.

[30] Kuzunori Isomoto, Noriyoshi Yoshida, and Jun'ichi Miyao. A parallel algorithm for k-way graph partitioning. *Electronics and Communications in Japan. Part 3*, 76(3):23–33, March 1993.

[31] T.L. Casavant and M. Singhal, editors. *Readings in Distributed Computing Systems*. IEEE Computer Society Press, Los Alamitos, CA, 1994.

[32] S. Mullender, editor. *Distributed Systems*. ACM Press, second edition, 1993.

[33] A.L. Ananda and B. Srinivasan, editors. *Distributed Computing Systems: Concepts & Structures*. IEEE Computer Society Press, Los Alamitos, CA, 1991.

[34] H.E. Bal. *Programming distributed systems*. Silicon press, Summit, NJ, 1990.

[35] V.S. Sunderam. PVM: A framework for parallel distributed computing. *Concurrency: Practice and experience*, 2(4):315–39, December 1990.

[36] M. Raynal. Distributed algorithms: their nature and the problems encountered. In *Proc. Int. Worksop on Par. and Dist. Algorithms*, Gers, France, 1989.

[37] M. Raynal. *Distributed Algorithms and Protocols.* John Wiley and Sons, 1988.

[38] D.P. Bertsekas and J.N. Tsitsiklis. Some aspects of parallel and distributed iterative algorithms - a survey. *Automatica*, 27:3–21, January 1991.

[39] D.P. Bertsekas and J.N. Tsitsiklis. *Parallel and distributed computation: Numerical methods.* Prentice-Hall, Englewood Cliffs, NJ, 1989.

[40] E.W. Dijkstra and S.C. Scholten. Termination detection for diffusing computation. *Information Processing Letters*, 11(1):1–4, August 1980.

[41] B. Awerbuch. Distributed shortest path algorithms. In *Proc. of the 21st Annual Symp. on Theory of Computing*, pages 490–500, Seattle, WA, 1989.

[42] K.M. Chandy and J. Misra. Distributed computations on graphs: Shortest path algorithms. *Communications of the ACM*, 11:833–837, 1982.

[43] R. Janardan and S.W. Cheng. Efficient distributed algorithms for single-source shortest paths and related problems on plane networks. *Mathematical Systems Theory*, 25(2):93–122, 1992.

[44] J. Park, N. Tokura, T. Masuzawa, and K. Hagihara. An efficient distributed algorithm for constructing a breadth-first search tree. *Systems and Computers in Japan*, 20(10):15–30, October 1989.

[45] B. Awerbuch and R.G. Gallager. A new distributed algorithm to find breadth first search trees. *IEEE Transactions on Information Theory*, 33(3):315–22, 1987.

[46] R.G. Gallager, P.A. Humblet, and P.M. Spira. A distributed algorithm for minimum-weight spanning trees. *ACM Trans. Prog. Lang. Syst.*, 5(1):66–77, January 1983.

[47] K.E. Johansen, U.L. Jorgansen, S.H. Nielson, and S.E. Skyum. A distributed spanning tree algorithm. In *Proc. 2nd Intl. Workshop on Distributed Algorithms*, pages 1–12, Amsterdam, 1988.

[48] S.A. Zenios and J.M. Mulvey. A distributed algorithm for convex network optimization problems. *Parallel Computing*, 6(1):45–56, January 1988.

[49] S.J. Mullender, G. van Rossum, A.S. Tanenbaum, R. van Renesse, and H. van Staveren. Amoeba – a distributed operating system for the 1990s. *IEEE Computer*, pages 44–53, May 1990.

[50] S. Ahuja, N. Carriero, and D. Gelernter. Linda and friends. *IEEE Computer*, 19(8):26–34, August 1986.

[51] Al Geist, Adam Beguin, Jack Dongarra, Weicheng Jiang, Robert Manchek, and Vaidy Sunderam. PVM 3 user's guide and reference manual. Technical Report ORNL/TM-12187, Oak Ridge National Laboratory, TN 37831, 1989.

Appendix A

Implementation Algorithms

A.1 Algorithm SA

```
int n, k, *GList[n], D[n], P[n], Seed[k], Bid[k][n];

procedure SA()
{
    read the inputs: G = (V, E) and k;        // initialization
    j=1+random()%k;
    for (i = 1; i ≤ k; i++)                   // seed computation
        Seed[i] = j + 1, j = 1 + (j + n/k)%n;
    for (i = 1; i ≤ k; i++)                   // bids computation
        ComputeBid(Seed[i], Bid[i]);
    DistributeNodes();                        // node distribution
    output the partition in P;
}
```

Figure A.1: A sequential implementation of the Auction algorithm.

A.2 Algorithm DA

```
int n, k, D[n], *GList[n], P[n], Worker[k], Seed[k], Bid[k][n];

procedure DA()
{
    read the inputs: G = (V, E) and k;          // initialization
    for (i = 1; i ≤ k; i++)
        spawn algorithm DAW as Worker[i];
    j=1+random()%k;                             // seed computation
    for (i = 1; i ≤ k; i++)
        Seed[i] = j + 1, j = 1 + (j + n/k)%n;
    broadcast n, k, D, GList,                   // data broadcasting
        Worker and Seed to all the workers;
    for (i = 1; i ≤ k; i++) {                   // bid receiving
        wait to receive a bidding response;
        answer from Worker[j] is kept in array Bid[j];
    }
    DistributeNodes();                          // node distribution
    output the partition in P;
}
```

Figure A.2: An algorithm at the master for distributed implementation of the algorithm Auction.

```
int n, k, D[n], *GList[n], P[n], Worker[k], Seed[k], Mybid[n];

procedure DAW()
{
    int me, myseed;
    receive n, k, D, GList,                          // initialization
        Worker and Seed from the master;
    find me such that, my identification (ppid) = Worker[me];
    myseed = Seed[me];
    ComputeBid2(myseed, Mybid);                       // bid computation and sending
    send Mybid to master;
}
```

Figure A.3: An algorithm at the workers for distributed implementation of the algorithm Auction.

A.3 Algorithm SP

```
int n, k, *GList[n], D[n], P[n], Seed[k], Bid[k][n];

procedure SP()
{
    read the inputs: G = (V, E) and k;
    for (i = 1; i ≤ n; i++) P[i] = 1 + i%n;    // starting partition
    GreedyPass();                               // iterative improvement
    output the partition in P;
}
```

Figure A.4: Sequential implementation of algorithm GreedyPass.

A.4 Algorithm SC

```
int n, k, *GList[n], D[n], P[n], Seed[k], Bid[k][n];

procedure SC()
{
    read the inputs: G = (V, E) and k;
    for (i = 1; i ≤ n; i++) P[i] = 1 + i%n;    // starting partition
    GreedyCycle();                              // iterative improvement
    output the partition in P;
}
```

Figure A.5: Sequential implementation of algorithm GreedyCycle.

A.5 Algorithm DC

```
int n, k, D[n], *GList[n], P[n], Worker[k], Seed[k], Bid[k][n];

procedure DC()
{
    int cost, oldcost = ∞, terminated = 0, color[k], xlst[2 * k];
    read the inputs: G = (V, E) and k;
    for (i = 1; i ≤ k; i++)
        spawn algorithm DCW as Worker[i];
    for (i = 1; i ≤ n; i++) P[i] = 1 + i%n;    // starting partition
    broadcast n, k, D, GList, Worker and P to all the workers;
    do {
        for (i = 1; i ≤ k; i++) {                // proposal receiving
            wait to receive proposal from a worker (suppose Worker[j]);
            proposed sending node is kept at xlst[2 * j];
            proposed destination is kept at xlst[2 * j + 1];
            color[j] = WHITE;
        }
        ProcessCycle();                          // cycle detection and processing
        cost = ComputeCost();
        if (cost ≥ oldost){
            send TERMINATE message to all the workers;
            terminated = 1;
        }
        else {
            send UPDATE message wih new P to all the workers;
        }
        oldcost = cost;
    }while (!terminated);
    output the partition in P;
}
```

Figure A.6: The master algorithm for distributed implementation of algorithm GreedyCycle.

```
int n, k, D[n], *GList[n], P[n], Worker[k], Seed[k], Mybid[n];

procedure DCW()
{
    int me, going, dest, terminated = 0;
    receive n, k, D, GList, Worker and P from the master;
    find me such that, my identification (ppid) = Worker[me];
    do {
        GreedyCompute(me, going, dest);
        send going and dest to the master;
        wait to receive message from the master;
        if (UPDATE)
            update P;
        else
            terminated = 1;
    }while (!terminated);
}
```

Figure A.7: The worker algorithm for distributed implementation of algorithm GreedyCycle.

A.6 Algorithm SAC

int $n,\ k,\ *GList\,[n],\ D\,[n],\ P\,[n],\ Seed\,[k],\ Bid\,[k]\,[n]$;

procedure SAC()
{
read the inputs: $G = (V, E)$ and k;
j=1+random()%k; // auction
for $(i = 1; i \leq k; i++)$
  $Seed\,[i] = j + 1, j = (j + n/k)\%n$;
for $(i = 1; i \leq k; i++)$
  ComputeBid($Seed[i]$, $Bid[i]$);
DistributeNodes();
GreedyCycle(); // iterative improvement
output the partition in P;
}

Figure A.8: Sequential implementation of the GreedyCycle with initial partition given by algorithm Auction.

A.7 Algorithm DAC

```
int n, k, D[n], *GList[n], P[n], Worker[k], Seed[k], Bid[k][n];
procedure DAC()
{
  int cost, oldcost = ∞, terminated = 0, color[k], xlst[2 * k];
  read the inputs: G = (V, E) and k;
  for (i = 1; i ≤ k; i++)
     spawn algorithm DCW as Worker[i];
  j=1+random()%k;                                  // auction
  for (i = 1; i ≤ k; i++)
     Seed[i] = j + 1, j = 1 + (j + n/k)%n;
  broadcast n, k, D, GList, Worker and Seed to all the workers;
  for (i = 1; i ≤ k; i++) {
     wait to receive a bidding response;
     answer from Worker[j] is kept in array Bid[j];
  }
  DistributeNodes();
  broadcast P to all the workers;
  do {                                             // cyclic iteration
     for (i = 1; i ≤ k; i++) {
        wait to receive proposal from a worker (suppose Worker[j]);
        proposed sending node is kept at xlst[2 * j];
        proposed destination is kept at xlst[2 * j + 1];
        color[j] = WHITE;
     }
     ProcessCycle();
     cost = ComputeCost();
     if (cost ≥ oldost){
        send TERMINATE message to all the workers;
        terminated = 1;
     }
     else
        send UPDATE message wih new P to all the workers;
     oldcost = cost;
  }while (!terminated);
  output the partition in P;
}
```

Figure A.9: The master algorithm for distributed implementation of Algorithm GreedyCycle with initial partition given by Auction.

```
int n, k, D[n], *GList[n], P[n], Worker[k], Seed[k], Mybid[n];

procedure DACW()
{
    int me, myseed, going, dest, terminated = 0;
    receive n, k, D, GList, Worker and Seed from the master;
    find me such that, my identification (ppid) = Worker[me];
    myseed = Seed[me];
    ComputeBid2(myseed, Mybid);
    send Mybid to master;
    receive P from the master;
    do {
        GreedyCompute(me, going, dest);
        send going and dest to the master;
        wait to receive message from the master;
        if (UPDATE)
            update P;
        else
            terminated = 1;
    }while (!terminated);
}
```

Figure A.10: The worker algorithm for distributed implementation of algorithm GreedyCycle with initial partition given by Auction.