Fast Graph Partitioning Algorithms

M. S. Khan* and K. F. Li
Department of Electrical and Computer Engineering
University of Victoria
Victoria, BC, Canada V8W 3P6
{skhan,kinli}@ sirius.uvic.ca

Abstract – **In this work, the following k-way graph partitioning (GP) problem is considered: 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. Two simple and fast algorithms are proposed, namely, direct algorithm** Auction **and iterative algorithm** GreedyCycle**.**

In algorithm Auction**, the idea of using auction and biddings is introduced using the master-workers paradigm. Algorithm** GreedyCycle **is a greedy algorithm where the idea of cyclic node passing among parts during the iterative improvement stage is introduced. Cyclic node passing is a k-way generalization of the 2-way node exchange found in the Kernighan-Lin approach. Experimental results show that, as compared to the existing algorithms, these algorithms are extremely fast, and they produce solutions of reasonable quality.**

1 Introduction

Dividing a graph into sets of nodes with the aim of optimizing certain cost-functions constitute an important combinatorial problem with applications including VLSI circuit partitioning and layout [4], and multiprocessor task allocation [1]. For this work, we use the following formulation of the k-way graph partitioning (GP) problem.

***k*-way GP problem:**

Input:

- Undirected graph $G(V, E)$ of n nodes (in the set V), and m edges (in the set E) 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 on the part size $|V_i|$.

Constraint:

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

Objective function:

- The partition-cost $c(\Pi) = \frac{1}{2} \sum_{i=1}^{k} \sum_{e \in E_{ext,i}} c(e)$, 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.

An exact solution of the GP problem is known to be NP-Hard even for $k = 2$. Two approaches have been taken for the GP problem: *direct* or *constructive* methods and *iterative improvement* techniques. For $k = 2$ (also called bipartitioning), direct algorithm based on max-flow min-cut technique is given by Stone and Bokhari [7], local search based iterative algorithm is given by Kernighan and Lin (KL-method) [2], and simulated annealing (SA-method) is given by Kirkpatrick *et al.* [4]. For general k-way GP problem, hierarchical methods using repeated bipartitioning are given in [2], and non-hierarchical methods are given in [6]. Recently Lee *et al.* [5] described a simple and efficient nonhierarchical algorithm for general k-way partitioning, and have shown that their algorithm outperforms the KL-method in both partition-cost, and computation time.

The rest of the paper is organized as follows. Section 2 describes algorithm Auction, and section 3 describes algorithm GreedyCycle. In section 4, we compare our algorithms with Lee's algorithm. Finally, section 5 gives the concluding remarks and future work.

*M. S. Khan was partially supported by a Canadian Commonwealth Scholarship during this project.

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



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

Figure 1: Distributed auction algorithm for graph partitioning: Algorithm at the master AUCTION.

```
int me, master;  // my pid and master's pid
int s;           // my initial seed
int M[n];        // my array of bids for n nodes



procedure AuctionW()
{
    receive n, k, D, GL, W
       and S from the master;        // initialization
    s = S[my position in W[1:k]];
    calculate M[1:n];                // bid computation
    send M to the master;            // sending
}
```

Figure 2: Distributed auction algorithm for graph partitioning: Algorithm at each worker processor AUCTIONW.

2 Direct algorithm – AUCTION

The AUCTION algorithm 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 chooses initial nodes, called seeds, one for each worker, 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 sends 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.

[1]Here *bid* means a one-to-one mapping of graph node indices to priority numbers 1 to n.

Figure 1 and 2 give our basic AUCTION algorithm in its generic form where we have basically two algorithms: algorithm AUCTION for the master, and algorithm AUCTIONW for each worker. We note that, algorithm AUCTION is inherently distributed in the sense that, once the workers are initiated, all the workers can compute their bid, which is the main computation part in this algorithm, independent of one another. However the results presented later in this paper uses a sequential implementation, which is detailed in [3]. 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.

3 Iterative algorithm GREEDYCYCLE

Algorithm GREEDYCYCLE is a heuristic iterative algorithm, and starts with an initial partition. The initial partition can be generated randomly, or may be obtained using some other algorithm. For each part in the current partition, we compute the most profitable node to go, and the corresponding destination, if any. Then the algorithm finds 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.

```
int Gv[k];      // gain-vector
int maxg = 0;   // maximum gain achievable in current pass


procedure GreedyCompute(s, g, t)()
int s, g, t;
{
    Let current part s = {s_1, s_2, s_3, ..., s_l};
    t = g = -1;
    for (i = 1; i ≤ l; i++){
        ∀j, 1 ≤ j ≤ k : Gv[j] = 0;
        for (j = 1; j ≤ D[s_i]; j++){
            u = GL[s_i][2 * j];
            Gv[P[u]] = Gv[P[u]] + GL[s_i][2 * j + 1];
        }
        Find x such that Gv[x] = max_{1≤j≤k} Gv[j];
        if ((Gv[x] - Gv[s] > maxg)
            t = x, g = s_i, maxg = Gv[x] - Gv[s];
    }
}
```

Figure 3: Algorithm GreedyCompute for a part to compute which node to go and where.

```
int G[k];       // part i gives g = G[i]
int T[k];       // part i gives t = T[i]
int C[n];       // node color vector, initially all 0's

 procedure ProcessCycle()
{
    for (i = 1; i ≤ k; i++, j = i){
        if (C[i] ≠ 0 || T[j] < 0) continue;
        while (C[j] == 0 && T[j] ≥ 0)
            C[j] = i, j = T[j];
        if (C[j] == i && T[j] ≥ 0){
            do {
                P[G[j]] = T[j];
                j = T[j];
            }while (C[j] == i);
        }
    }
}
```

Figure 4: Algorithm ProcessCycle for cycle detection and processing in the parts-graph.

To show the rationale of the above idea, 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. 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. The 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 interest 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 to overcome this basic problem.

Algorithm GreedyCycle appears 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. 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 might miss.
- If there are 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: algorithm GreedyCompute used to find the node passing interests, ProcessCycle used to find and process the cycles in node passing interests, and finally GreedyCycle which binds the pieces together.

Algorithm GreedyCompute

Algorithm GreedyCompute of Figure 3 does the main computation in the algorithm GreedyCycle. It takes a part number s as input parameter, and computes two output parameters g and t using the input graph and current partition information, where g gives the node which obtains the maximum achievable gain under current circumstances, and this gain is obtained if the node is sent to part t. If there is no effective gain possible, it gives the value -1 for both g and t.

Algorithm ProcessCycle

Let us define *parts-graph* as the directed graph $G_p = (V_p, A_p)$, where V_p is the node set, and A_p is the arc set, such that each node in V_p corresponds to a part of the original graph G. G_p can be obtained by applying GreedyCompute on all the current parts as follows: for any part s, $s \in V_p$, and if GreedyCompute(s, g, t) returns $t \neq -1$, then $(s, t) \in A_p$, otherwise node s does

```
int G[k];          // part i gives g = G[i]
int T[k];          // part i gives t = T[i]
int C[n];          // node color vector
int cost, oldcost = ∞;
int terminated = 0;


procedure GreedyCycle()
{
      do {
         for (i = 1; i ≤ k; i++){
            GreedyCompute(i, G[i], T[i]);
            color[i] = 0;
         }
         ProcessCycle();
         Compute new cost;
         if (cost ≥ oldost)
            terminated = 1;
         oldcost = cost;
      }while (!terminated);
}
```

Figure 5: Algorithm GreedyCycle for k-way graph partitioning.

not have any outgoing arc. Thus $|V_p| = k$ and $|A_p| \leq k$. Such a graph will have the following properties.

- Any node in a cycle cannot have an outgoing arc to a node outside the cycle. Since each node has only one outgoing arc, all the outgoing arcs 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 arc.

These two properties give us the algorithm ProcessCycle of figure 4 to find the cycles in parts-graph, and process them accordingly.

Algorithm GreedyCycle

Figure 5 gives the iterative GP algorithm GreedyCycle. At the onset of each iteration, we first compute the most promising node offer from each part to get the parts-graph using GreedyCompute, and then we call ProcessCycle to detect all the cycles in the parts-graph, and pass the nodes accordingly. If we improve the partition-cost in this process, the algorithm starts a fresh iteration; otherwise it terminates.

4 Comparisons

Table 1 compares some theoretical and experimental features of our algorithms with Lee's algorithm, one of the best algorithms available in the literature for the k-way GP problem. Since our algorithms use adjacency lists for the graph our algorithms need less memory compared to Lee's algorithm where they use adjacency matrix. On time comparison, algorithm Auction is the fastest since it does not require any iteration. Although the execution time of iterative algorithms GreedyCycle and Lee depends on the number of iterations r_1 and r_2, we note that the per iteration complexity of algorithm GreedyCycle is significantly less than that of algorithm Lee.

Table 1: Comparison of our algorithms with algorithm Lee

Features	Auction	GreedyCycle	Lee
Theoretical issues			
Type	Direct	Iterative, Greedy	Iterative, Nongreedy
Memory	$\mathcal{O}(n^2)^\bullet$	$\mathcal{O}(2m)$	$\mathcal{O}(2m)$
Time	$\mathcal{O}(kn \lg n + km)$	$\mathcal{O}(r_1(kn+m))$	$\mathcal{O}(r_2 kn^2)$
Typical experimental performance against Lee			
Quality	Within 15%	Within 10%$^\diamond$	—
Speed	10× faster	7× faster	—

$^\bullet$Due to high memory requirement, Lee's algorithm could not be tested for $n > 800$.

$^\diamond$Auction+GreedyCycle gives quality within 4%.

For field comparison of the algorithms, we have experimented with the following algorithms: random partitioning (RAN), Lee's algorithm (LEE), algorithm Auction (AUC), algorithm GreedyCycle (GC) and algorithm Auction+GreedyCycle (A+GC) (uses GreedyCycle on an initial partition generated by Auction). We have used many test graphs which were randomly generated with number of nodes n and maximum degree d as parameters. To compare these implementations we have used the following two metrics:

1. *Quality of Solution:* The quality of a solution of an algorithm means how good a partition is given by a certain algorithm. Quantitatively, we define the *relative quality* $Q(A)$ of an algorithm A as the ratio of the partition-cost given by algorithm A to the partition-cost given by the random partitioning. Thus a lower Q(A) means better quality.

2. *Time of execution:* For our experiments we use the Unix */bin/time* command to measure the *real time* required by an algorithm A between the invocation of the command to the production of output, which we denote by symbol T. Then relative time $Tr(A)$ of an

algorithm A is given by the ratio of real time required by algorithm A to real time required by RAN. A lower Tr(A) means faster execution.

Quality of solution comparisons

Figure 6 gives the comparison of the quality of solution, as a function of problem size n, among all our algorithms and algorithm LEE along with the random partitioning. The tested algorithms arranged in descending quality of output are as follows: LEE, A+GC, GC, and AUC. Quantitatively, A+GC gave a solution quality of within 4% of that given by LEE, and GC and AUC gave a solution quality within 10% and 15% respectively.

Time of execution comparisons

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 algorithms, (AUC, GC, and A+GC) is shown in Figure 7 as a function of problem size n. We note that all our algorithms perform a lot faster compared to Lee's algorithm. For example, for $n = 400$, $k = 10$ and $d = 10$, algorithm AUC took 2.5 seconds, GC took 3.8 seconds, and algorithm A+GC took 3.5 seconds as compared to 25.4 seconds taken by Lee's algorithm. Compared to Lee's algorithm, AUC is 10 times faster, and GC and A+GC is 7 times faster.

5 Conclusion and future work

In this work, we have given two fast algorithms AUCTION, and GREEDYCYCLE, for the k-way graph partitioning problem. Our algorithm AUCTION is a very fast algorithm using the new idea of master-workers auction. 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. One of the main findings 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.

We are currently working on further performance comparisons with existing algorithms, including the simulated annealing method. We have also implemented these algorithms and some of their variations in the distributed PVM [8] environment. The initial results are encouraging [3], and we believe that implementation with more efficient communications support will improve the distributed performance significantly. As a future project, the field-performance of the algorithms may be tested by some real-world application programming. Distributed task allocation and VLSI circuit partitioning would be two interesting applications.

References

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

[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] M. S. Khan. Sequential and distributed algorithms for fast graph partitioning. Master's thesis, University of Victoria, Victoria, B.C., Canada, August 1994.

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

[5] 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.

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

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

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

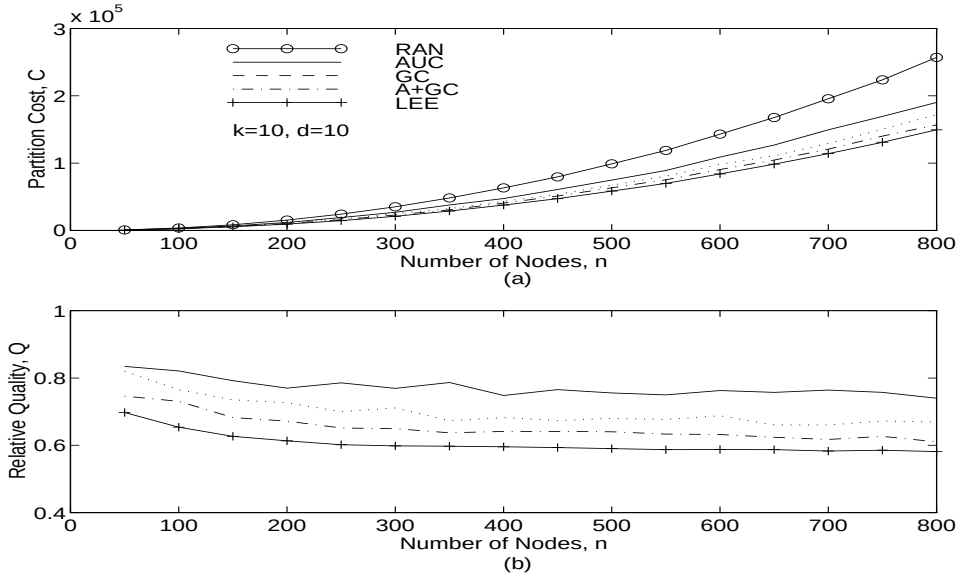

Figure 6: Comparison of solution qualities given by our algorithms with algorithm LEE as a function of n.

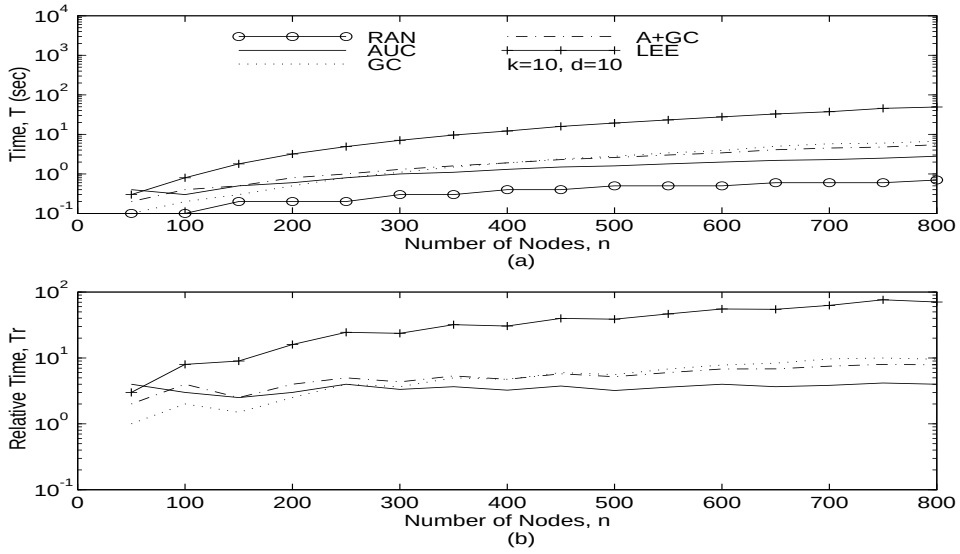

Figure 7: Comparison of execution times of our algorithms with algorithm LEE as a function of n.