1994

Genetic Algorithms for Graph Partitioning and Incremental Graph Partitioning

Harpal Maini
Syracuse University, hsmaini@top.cis.syr.edu

Kishan Mehrotra
Syracuse University, mehrotra@syr.edu

Chilukuri K. Mohan
Syracuse University, ckmohan@syr.edu

Sanjay Ranka
Syracuse University, ranka@top.cis.syr.edu

Genetic Algorithms for Graph Partitioning and Incremental Graph Partitioning*

Harpal Maini Kishan Mehrotra Chilukuri Mohan Sanjay Ranka†
School of Computer and Information Science, 4-116 CST
Syracuse University, Syracuse, NY 13244-4100

email: hsmaini/kishan/mohan/ranka@top.cis.syr.edu

September 10, 1996

Abstract

Partitioning graphs into equally large groups of nodes, minimizing the number of edges between different groups, is an extremely important problem in parallel computing. This paper presents genetic algorithms for suboptimal graph partitioning, with new crossover operators (KNUX, DKNUX) that lead to orders of magnitude improvement over traditional genetic operators in solution quality and speed. Our method can improve on good solutions previously obtained by using other algorithms or graph theoretic heuristics in minimizing the total communication cost or the worst case cost of communication for a single processor. We also extend our algorithm to Incremental Graph Partitioning *problems, in which the graph structure or system properties changes with time.*

1 Introduction

Graph partitioning is the task of dividing the nodes of a graph into groups called *parts* (or bins), in such a way that each part has roughly the same number of nodes, and minimizing the *cut-size*, i.e., the number of edges that connect nodes in different parts. This problem has important applications in parallel computing. For instance, efficiently parallelizing many scientific and engineering applications requires partitioning data or tasks among processors, such that the computational load on each node is roughly the same, while inter-processor communication is minimized.

Obtaining exact solutions for graph partitioning is computationally intractable, and several suboptimal methods have been suggested for finding good solutions to the graph partitioning problem. Important heuristics include recursive coordinate bisection, recursive graph bisection, recursive spectral bisection, mincut based methods, clustering techniques, geometry-based mapping, block-based spatial decomposition, and scattered decomposition [3, 11, 12, 15].

We present genetic algorithms for graph partitioning, using new crossover operators that utilize information available from the history of genetic search. Our work is characterized by the following features:

1. *Use of prior information to improve solutions.*
2. *Efficient partitioning of graphs to which incremental updates are made.*
3. *Parallelizability, using a distributed genetic algorithm model.*
4. *Refinement of parts obtained by other methods.*
5. *Optimization of the worst case communication cost, a non-differentiable function.*

We have obtained excellent results due to newly developed genetic recombination operators (KNUX and DKNUX) that exploit domain-specific knowledge. These give improved solutions and faster convergence rates when compared with the traditional crossover operators. Exact comparisons of the different algorithms are not available due to the unavailability of

*This is a revised version of a paper that appeared in Proc. IEEE Supercomputing Conf., 1994.

†Sanjay Ranka is currently at the University of Florida, Gainesville. His work was partially supported by NSF grant CCR-9110812 and DARPA contract #DABY63-91-C-0028. The contents of this paper do not necessarily reflect the position or policy of the United States government, and no official endorsement should be inferred.

benchmark problems and results. However, our experiments with the traditional crossover operators used by some of these researchers gave results of lower quality than using the operators presented in this paper.

The results achieved by our methods are better or comparable to the best known methods for graph partitioning, for graphs with a few hundred nodes. The quality of solutions obtained using DKNUX is competitive with recursive spectral bisection as a graph partitioning strategy, especially for incremental graph partitioning. However, genetic algorithms do require much more execution time than greedy algorithms, and are recommended in applications where the quality of solution is important enough to warrant the extra computational effort. Fortunately, GA's are readily parallelizable, with near-linear speedups. Applying a prior graph contraction step should precede the partitioning of very large graphs using GA's.

Section 2 describes the task addressed by the genetic algorithm. Section 3 describes how genetic algorithms are applied to this problem. Experimental results are given in Section 4.

2 Graph Partitioning as Optimization

Let V denote the set of vertices of the graph to be partitioned, and let E denote its edges. The graph-partitioning problem consists of finding an assignment scheme $M : V \longrightarrow P$ that maps vertices to n parts. We denote by $B(q)$ the set of vertices assigned to a part q, *i.e.*, $B(q) = \{v \in V : M(v) = q\}$. Edges may connect physically proximate vertices in graphs representing the computational structure of a physical domain.

Graph partitioning is a multi-objective optimization problem, since both load imbalance and communication costs must be minimized. These objectives are often achieved by minimizing either

$$\sum_q I(q) + \beta \sum_q C(q),$$

where $I(q)$ is the load imbalance attributed to part q, $C(q)$ is the communication cost attributed to part q, and β expresses the relative importance of the two objectives. This composite cost function focuses on the total communication cost; an alternative is to minimize

$$\sum_q I(q) + \beta \max_q C(q),$$

which focuses on the communication cost for the worst part. Our methods work with either formulation. For domain decomposition methods, optimizing the latter function is more desirable. The former is often used because most traditional methods require differentiability of the function being optimized.

The weight w_i corresponds to the computation cost (or weight) of a vertex $v_i \in V$. The average load of each part is $\sum_{v_i \in V} w_i/n$. We define the load imbalance attributed to the qth part as

$$I(q) \ = (\textstyle\sum_{v_i \in B(q)} w_i - \sum_{v_i \in V} w_i/n)^2,$$

where n is the number of parts into which the graph must be partitioned.

The communication cost $w_e(v_1, v_2)$ corresponding to an edge describes the amount of interaction between vertices v_1 and v_2. The cost of all the outgoing edges from a part is

$$C(q) \ = \textstyle\sum_{v_i \in B(q),\ v_j \notin B(q)} w_e(v_i, v_j)$$

Genetic algorithms attempt to maximize a "fitness" function, whose value is relatively high for candidate solutions of better quality. Our experiments were conducted using the following two fitness functions, which assume unit (equal) computation cost (w_i) for each node, unit communication cost (w_e) for each edge, and $\beta = 1$.

$$\mathsf{Fitness}_1 : \quad -\left(\sum_q (|B(q)| - \frac{|V|}{n})^2 + \sum_q C(q)\right)$$

$$\mathsf{Fitness}_2 : \quad -\left(\sum_q (|B(q)| - \frac{|V|}{n})^2 + \max_q C(q)\right)$$

3 Genetic Algorithms for Graph Partitioning

Genetic algorithms (GAs) are stochastic state-space search techniques modeled on natural evolutionary mechanisms [4]. The *population*, a set of *individuals* (potential solutions to the optimization problem) steadily changes with time due to the application of operators such as *crossover* and *mutation*. A *selection* process determines which individuals (from among parents and offspring) remain in the next generation. Genetic algorithms have been used in the past to find good suboptimal solutions to the graph-partitioning problem [1, 8, 5, 6]. This section describes the representation used to solve the graph partitioning problem, the genetic operators used, and various methods of improving the performance of the GA.

3.1 Representation

For graph partitioning, we select a vector representation for each individual (candidate solution), in which the i^{th} element of an individual is j iff the i^{th} node of the graph is allocated to the part labelled j. For instance, the string 11100011 represents the mapping that assigns nodes 1,2,3,7,8 to part (processor) 1 and nodes 4,5,6 to part (processor) 0. According to our definitions of fitness, if the graph is one in which the i^{th} node is adjacent to the $(i+1)^{st}$ node for each i, then 11100011 would be less fit than 11100001 (which is a more balanced partition), but more fit than 10101011 (which has 6 inter-part edges).

3.2 Crossover

One-point crossover [4] works by selecting a site in chromosomes $\alpha\beta$ and $\gamma\delta$ to produce $\alpha\delta$ and $\gamma\beta$. A popular generalization is 2-point crossover, in which the parents $\alpha\beta\gamma$ and $\delta\epsilon\phi$ produce offspring $\alpha\epsilon\gamma$ and $\delta\beta\phi$. This has been further generalized to 'k-point crossover'. In *uniform crossover* (UX) [14], the i^{th} component of an offspring is chosen to be the same as that of one of the two parents, with equal probability.

UX ignores the fact that one parent may have much better genetic material than another, or that one region of the search space is already known to produce individuals of higher fitness than other regions. UX can be described in terms of a bit-vector mask, each bit of which determines the parent from which an offspring inherits a value for a particular bit-position.

Our new *Knowledge-based Non-Uniform Crossover* operator (*KNUX*) generalizes this idea, using a *bias probability vector* $\boldsymbol{p} = (p_1, \ldots, p_n)$, where each p_i is a real number $\in [0, 1]$. The value of each bias probability p_i depends on i, the relative fitness of the parent strings, and on problem-specific knowledge. Given $\boldsymbol{p}$ and the two parents, $\boldsymbol{a} = (a_1, \ldots, a_n)$ and $\boldsymbol{b} = (b_1, \ldots, b_n)$, the offspring $\boldsymbol{c} = (c_1, \ldots, c_n)$ is obtained such that if $a_i = b_i$, then $c_i = a_i$, else the probability that $c_i = a_i$ is p_i.

For graph partitioning, an initial candidate solution I is first generated. Let $\nu(i)$ be the set of neighbors of node i in the graph under consideration. For any candidate solution X, let $\#(i, X, I)$ be the number of nodes in $\nu(i)$ that are allocated by I to part X_i. If $\boldsymbol{a}$ and $\boldsymbol{b}$ are the two parents, then we define

$$p_i = \begin{cases} 0.5 & \text{if } \#(i,\boldsymbol{a},I) = 0 \ \& \ \#(i,\boldsymbol{b},I) = 0 \\ \frac{\#(i,\boldsymbol{a},I)}{\#(i,\boldsymbol{a},I)+\#(i,\boldsymbol{b},I)} & \text{otherwise.} \end{cases}$$

3.3 Dynamic KNUX (DKNUX)

The quality of solutions obtained by KNUX depends on the quality of the heuristic estimate (I above) used to derive bias probabilities. It is therefore important to obtain a good, fast heuristic estimate of a solution. DKNUX utilizes information inherent in the history of the genetic search, and continually updates the estimate I to be the current best solution, using this to build the bias vector.

3.4 Distributed Population Model

We use a coarse-grained, distributed-population genetic algorithm (DPGA), where individuals are distributed into various subpopulations which may be physically located on different processors configured in some architecture (e.g., mesh). Crossovers are restricted to occur between members of the same subpopulation. Each subpopulation periodically communicates copies of its best individuals to its neighboring subpopulations (situated on neighboring processors in the parallel architecture); this is how genetic information is exchanged.

3.5 Population Initialization

The initial population can be seeded with a preestimated heuristic solution such as that obtained through an Index Based Partitioning scheme or the results of recursive spectral bisection. In the incremental case, the previous partitioning can itself be used to generate a good partitioning for the changed graph by randomly assigning new graph nodes to various nodes, while at the same time ensuring that balance is maintained.

3.6 Hill Climbing

It is possible to perform hill-climbing on offspring, to obtain the nearest local optima of the fitness function. Only the "boundary points" of each part (with neighbors in other parts) are examined to see if migrating them to the appropriate neighboring part improves fitness.

4 Experimental Results

In this section, we compare the results obtained using our approach with those of traditional heuristics (e.g., IBP or RSB) as well as with genetic algorithms

that invoke traditional crossover operators. The figures are obtained by averaging the results of 5 runs, and the tables represent the best solutions obtained in these 5 runs. All experiments were done with algorithm DPGA set with a total population size of 320. The crossover rate $p_c = 0.7$ and the mutation rate $p_m = 0.01$. Tables 1, 2 and 3 report $\sum_q C(q)/2$ values, while Tables 4, 5 and 6 report $\max_q C(q)$ values, where $C(q)$ is the number of edges that cut across part q.

Experiments were conducted with a single population as well as with 16 subpopulations configured as a four dimensional hypercube. Graphs with unit weight nodes and edges were assumed, although weighted edges and nodes can also be handled easily. For clarity, the cut-size numbers are given in the tables, instead of the actual fitness function values; for graph-partitioning, smaller cut-size numbers indicate superior performance. The results establish very clearly the excellent performance of KNUX and DKNUX in comparison with two-point crossover and also that DKNUX is competitive with recursive spectral bisection as a graph partitioning strategy.

4.1 Improving solutions obtained using other methods

Fast heuristic algorithms can be used to obtain an initial candidate solution which is then improved by applying the genetic algorithm. Table 1 compares the results of Recursive Spectral Bisection (RSB) [11, 12, 13] with the GA initialized by a solution obtained by the Index-Based Partitioning algorithm (IBP) [10] described in the Appendix.

Number of Parts	2	4	8
167 Nodes			
Cut Using DKNUX	20	63	109
Cut Using RSB	20	59	120
144 Nodes			
Cut Using DKNUX	33	65	120
Cut Using RSB	36	78	119

Table 1: A Comparison of the Best Solutions found Using DKNUX and RSB: starting with a population initialized with an IBP solution, using Fitness Function 1. In each case, the total number of inter-part edges is reported for the best individual explored by the GA.

Number of Parts	2	4	8
139 Nodes			
Cut Using DKNUX	28	65	100
Cut Using RSB	30	69	113
213 Nodes			
Cut Using DKNUX	41	77	138
Cut Using RSB	41	82	151
243 Nodes			
Cut Using DKNUX	43	88	141
Cut Using RSB	47	95	154
279 Nodes			
Cut Using DKNUX	36	78	139
Cut Using RSB	37	88	155

Table 2: Improving the Solution found through Recursive Spectral Bisection, using Fitness Function 1. In each case, the total number of inter-part edges is reported for the best individual explored by the GA.

4.2 Incremental Graph Partitioning

For this series of experiments, we start with a graph, partition it, then modify by adding some number of nodes in a local area chosen randomly within the graph. The modified graphs are then partitioned.

Number of Parts	2	4	8
118 plus 21 Nodes			
Cut Using DKNUX	31	61	103
Cut Using RSB	30	69	113
118 plus 41 Nodes			
Cut Using DKNUX	31	66	120
Cut Using RSB	33	75	128
183 plus 30 Nodes			
Cut Using DKNUX	37	72	133
Cut Using RSB	41	82	151
183 plus 60 Nodes			
Cut Using DKNUX	44	83	160
Cut Using RSB	47	95	154

Table 3: A Comparison of the Best Solutions found Using DKNUX and RSB: Incremental Graph Partitioning, using Fitness Function 1. In each case, the total number of inter-part edges is reported for the best individual explored by the GA.

Number of Parts	4	8
78 Nodes		
Worst Cut Using DKNUX	23	23
Worst Cut Using RSB	26	25
88 Nodes		
Worst Cut Using DKNUX	28	21
Worst Cut Using RSB	33	27
98 Nodes		
Worst Cut Using DKNUX	26	23
Worst Cut Using RSB	30	30
144 Nodes		
Worst Cut Using DKNUX	53	42
Worst Cut Using RSB	44	35
167 Nodes		
Worst Cut Using DKNUX	44	39
Worst Cut Using RSB	40	41

Table 4: A Comparison of the Best Solutions found Using DKNUX and RSB: Starting with a Randomly Initialized Population and Using Fitness Function 2. "Worst Cut" refers to $max_q C(q)$, where $C(q)$ is the number of edges leading out of part q. For the GA, the maximum number of edges leading out of a part is reported, for the best individual explored by the GA.

4.3 Minimizing Worst Case Communication Cost

Unlike other methods which can be used only with a differentiable optimization function, genetic algorithms can be used directly to optimize $\sum_q I(q) + \beta \max_q C(q)$, a task that cannot be attempted with methods that require availability of the first derivative of the function to be optimized. Table 4 exhibits the effect of partitioning graphs of 78, 88, 98, 144 and 167 nodes into 4 and 8 parts, respectively. Table 4 shows the best solution found using operator DKNUX is better than that obtained using RSB in most cases. In other cases, improvements can be obtained by seeding the initial population with a heuristically obtained good solution such as the index based partitioner.

5 Conclusions

We have solved the graph partitioning problem using GA's with new knowledge-based crossover operators; problem-specific knowledge is used to generate bias probabilities, and the "environment" and current population play roles in controlling genetic expression. The trajectory that the population takes in search

Number of Parts	4	8
78 Nodes		
Worst Cut Using DKNUX	23	20
Worst Cut Using RSB	26	25
88 Nodes		
Worst Cut Using DKNUX	24	22
Worst Cut Using RSB	33	27
98 Nodes		
Worst Cut Using DKNUX	24	22
Worst Cut Using RSB	30	30
213 Nodes		
Worst Cut Using DKNUX	40	41
Worst Cut Using RSB	46	45
243 Nodes		
Worst Cut Using DKNUX	45	41
Worst Cut Using RSB	51	47
279 Nodes		
Worst Cut Using DKNUX	42	42
Worst Cut Using RSB	46	47
309 Nodes		
Worst Cut Using DKNUX	44	47
Worst Cut Using RSB	46	52

Table 5: A Comparison of the Best Solutions found Using DKNUX: Improving Upon RSB Solutions Using Fitness Function 2. "Worst Cut" refers to $max_q C(q)$, where $C(q)$ is the number of edges leading out of part q. For the GA, the maximum number of edges leading out of a part is reported, for the best individual explored by the GA.

Number of Parts	4	8
78 plus 10 nodes		
Worst Cut Using DKNUX	27	25
Worst Cut Using RSB	33	27
78 plus 20 nodes		
Worst Cut Using DKNUX	29	27
118 plus 21 Nodes		
Worst Cut Using DKNUX	33	29
Worst Cut Using RSB	38	34
118 plus 41 Nodes		
Worst Cut Using DKNUX	34	35
Worst Cut Using RSB	40	39
183 plus 30 Nodes		
Worst Cut Using DKNUX	41	40
Worst Cut Using RSB	46	45
183 plus 60 Nodes		
Worst Cut Using DKNUX	46	45
Worst Cut Using RSB	51	47
249 plus 30 Nodes		
Worst Cut Using DKNUX	42	44
Worst Cut Using RSB	51	47
249 plus 60 Nodes		
Worst Cut Using DKNUX	46	56
Worst Cut Using RSB	46	52

Table 6: A Comparison of the Best Solutions found Using DKNUX and RSB: Incremental Partitioning with Fitness Function 2. "Worst Cut" refers to $max_q C(q)$, where $C(q)$ is the number of edges leading out of part q. For the GA, the maximum number of edges leading out of a part is reported, for the best individual explored by the GA.

space is constrained, driving evolution in certain preferred directions.

We have introduced novel operators that exploit the locality information inherent in most computational graphs. We have shown this enhances the speed and performance of genetic search by orders of magnitude. We have demonstrated that genetic algorithms can be used to greatly refine previously estimated parts with the help of KNUX and DKNUX. We show how the strategies discussed in this paper extend naturally to incremental graph partitioning. The incremental partitioning results obtained using DKNUX could not be obtained by a simple deterministic algorithm that assigns new nodes to the part to which most of its nearest neighbors belong. Performance can further be improved by incorporating a hill-climbing step.

We have presented preliminary results showing the feasibility of this approach and the gains obtainable by examining the history of the search process; unfortunately, partitioning very large graphs does require high amounts of computation by the genetic algorithm. A prior graph contraction step would allow these techniques to be applied to graphs much larger than those explored in this paper [13]. Some gains can be expected from executing the GA on parallel computers, since DPGA is an inherently parallel algorithm from which we can expect near-linear speedups. We are currently parallelizing the algorithm to run on distributed memory machines such as the CM-5 and the Intel Paragon.

References

[1] J. P. Cohoon, W. N. Martin, and D. S. Richards, "A multi-population genetic algorithm for solving the k-partition problem on hypercubes," *Proc. 4th ICGA*, 1991, pp. 244–248.

[2] R. Collins and D. Jefferson, "Selection in massively parallel genetic algorithms," *Proc. 4th ICGA*, 1991, pp. 249–256.

[3] F. Ercal, "Heuristic Approaches to Task Allocation for Parallel Computing", *Ph.D. Thesis, Ohio State University* , 1988.

[4] J. H. Holland, *Adaptation in Natural and Artificial Systems*, University of Michigan Press, Ann Arbor, 1975.

[5] D. R. Jones and M. A. Beltramo, "Solving partitioning problems with genetic algo-

rithms," *Proc. of the 4th ICGA*, 1991, pp. 442–450.

[6] G. von Laszewski, "Intelligent structural operators for the k-way graph partitioning problem," *Proc. of the 4th ICGA*, 1991, pp. 45–52.

[7] H. S. Maini, "Incorporation of Knowledge in Genetic Recombination", Ph.D. Thesis, School of Computer & Information Science, Syracuse University, August 1994.

[8] N. Mansour, *Physical Optimization Algorithms for Mapping Data to Distributed-Memory Multiprocessors*, Ph.D. Thesis, School of Computer and Information Science, Syracuse University, 1992.

[9] H. Muhlenbein, "Parallel genetic algorithms, population genetics and combinatorial optimization," *Proc. 3rd ICGA*, 1989, pp. 416–422.

[10] C.-W. Ou, S. Ranka, and G. Fox, "Fast mapping and remapping algorithm for irregular and adaptive problems," *Proc. of the International Conference on Parallel and Distributed Computing*, December 1993.

[11] A. Pothen, H. Simon, and K-P. Liou, "Partitioning sparse matrices with eigenvectors of graphs," *SIAM J. Matrix Anal. Appl.*, 11, 3 (July), 1990, pp. 430–452.

[12] H. Simon, "Partitioning of unstructured mesh problems for parallel processing," *Proc. Conf. Parallel Methods on Large Scale Structural Analysis and Physics Applications*, Pergamon Press, 1991.

[13] S. T. Barnard, H. D. Simon, "A fast multilevel implementation of recursive spectral bisection for partitioning unstructured problems," Technical Report RNR-92-033, November 1992.

[14] G. Syswerda, "Uniform crossover in genetic algorithms," *Proc. of the 3rd ICGA*, 1989, pp. 2–9.

[15] R. D. Williams, "Performance of dynamic load balancing algorithms for unstructured mesh calculations," *Concurrency: Practice and Experience*, 3(5), 1991, pp. 457–481.

Appendix: Index-Based Partition (IBP) Algorithm

Index-based algorithms to partition graphs have been described in [10]. An IBP algorithm includes three phases— indexing, sorting, and coloring. The indexing scheme is based on converting an N-dimensional co-ordinate into a one-dimensional index such that proximity in the multi-dimensional space is maintained. Row-major indexing and shuffled row-major indexing are two of the several ways of indexing pixels in a two-dimensional grid. These two indexing schemes are shown in Figure 1 for a graph in which the set of vertices are arranged in a grid of size 8×8.

```
00 01 02 03 04 05 06 07    00 01 04 05 16 17 20 21
08 09 10 11 12 13 14 15    02 03 06 07 18 19 22 23
16 17 18 19 20 21 22 23    08 09 12 13 24 25 28 29
24 25 26 27 28 29 30 31    10 11 14 15 26 27 30 31
32 33 34 35 36 37 38 39    32 33 36 37 48 49 52 53
40 41 42 43 44 45 46 47    34 35 38 39 50 51 54 55
48 49 50 51 52 53 54 55    40 41 44 45 56 57 60 61
56 57 58 59 60 61 62 63    42 43 46 47 58 59 62 63

        (a)                        (b)
```

Figure 1: (a) Row-Major and (b) Shuffled Row-Major Indexing for an 8×8 image

A simple example of interleaving indices is as follows. Suppose $index_1 = 001$, $index_2 = 010$, and $index_3 = 110$. Then the interleaved index would be 001011100. In the above case the number of bits in each dimension are equal. This could easily be generalized to cases when the sizes are different. For example if $index_1 = 101$, $index_2 = 01$, and $index_3 = 0$, then the interleaved index would be 100110. This is done by choosing bits (right to left) of each of the dimensions one by one, starting from dimension 3. When the bits of a particular dimension are no longer available, that dimension is not considered.

After indexing is done, an efficient sorting algorithm can be applied to sort these vertices according to their indices. Finally, this sorted list is divided into P equal sublists.