A Round-Robin Parallel Partitioning Algorithm

Doug Moore*

TR 88-916
June 1988

Department of Computer Science
Cornell University
Ithaca, NY 14853-7501

* Supported in part by the National Science Foundation under grant DCR-84-51385.

A Round-Robin Parallel Partitioning Algorithm

Doug Moore†

Computer Science Department
Cornell University
Ithaca, New York 14853

1 June 1988

Abstract

We develop parallel heuristic algorithms for partitioning the vertices of a graph into many groups of roughly equal size so that few edges connect vertices in different groups. The algorithms are intended for a message passing multiprocessor such as a hypercube, but only require the processors to be connected as a ring. They are based on the Kernighan-Lin algorithm, which finds a small edge separator that divides a graph into two pieces of equal size. Each of the algorithms features a parameter that allows for a trade-off between the potentially conflicting goals of reducing the number of edges between different groups and keeping the sizes of the groups roughly the same.

These algorithms are applied to the problem of parallel block oriented Cholesky factorization. For an efficient factorization, we need a graph partition in which few pairs of vertex groups are interconnected; that is, we desire the quotient graph induced by the partition to be sparse. We discuss how standard partitioning heuristics may fail to give sparse quotient graphs and how they can be modified to correct this.

---

† Research supported by the National Science Foundation under grant DCR-84-51385.

1 Introduction

Like many other problems, graph problems are often efficiently solved by use of the divide-and-conquer paradigm. We divide the graph into two or more parts, solve the problem on each of the parts, and combine the partial solutions to form a solution to the original, larger problem. Combining the partial solutions is easiest when the parts of the graph are somewhat independent. Therefore, finding small vertex or edge separators is an important part of many graph algorithms. For example, small separators are required in sparse numerical algorithms and in VLSI design.

The use of local-memory multiprocessors, such as the NCUBE and the Intel iPSC hypercubes, permit the solution of problems too large to fit in the memory of one processor, but rule out algorithms which require random access to the input data. Such machines consist of a number of identical processors, each with a small local memory. Two such processors may share data only by explicitly sending and receiving messages. The communication time depends on which processors are communicating and upon the size of the message, but for any message is many times slower than the amount of time one processor would need to copy the message. Therefore, we must not only be concerned with what is to be computed, but also where it is to be computed and where and when the results are needed.

Divide-and-conquer algorithms are well-suited to this parallel environment, because the separate subproblems may be solved simultaneously without excessive communication. Finding separators in parallel, then, can be a useful preprocessing step for parallel graph algorithms. In [GZ87], [Z88], Zmijewski applies the divide-and-conquer approach to the problem of finding certain kinds of edge separators. In this paper, we apply a different paradigm, the round-robin paradigm, to the problem of finding small edge separators, or, equivalently, to the problem of obtaining good graph partitions.

This paper is organized as follows. The next section defines the basic two-way graph partitioning problem, also called the min-cut problem, and reviews several of the heuristic algorithms that have been devised to solve it. Section three generalizes the 2-way partitioning problem to the multiway problem and introduces the round-robin algorithm to solve the more general problem. The fourth section concerns the relaxation of some constraints of the basic problem and how, for some applications, the less constrained problem has significantly better solutions. In the fifth section we develop techniques for comparing graph partitions into more than two pieces and in the sixth modify the round-robin algorithm to allow the number of pieces of the graph to change as the algorithm progresses. Section seven concerns a new and different criterion for measuring the quality of a graph partition and proposes a heuristic parallel partitioning algorithm to satisfy this criterion. We then present the results of experiments on the Intel hypercube

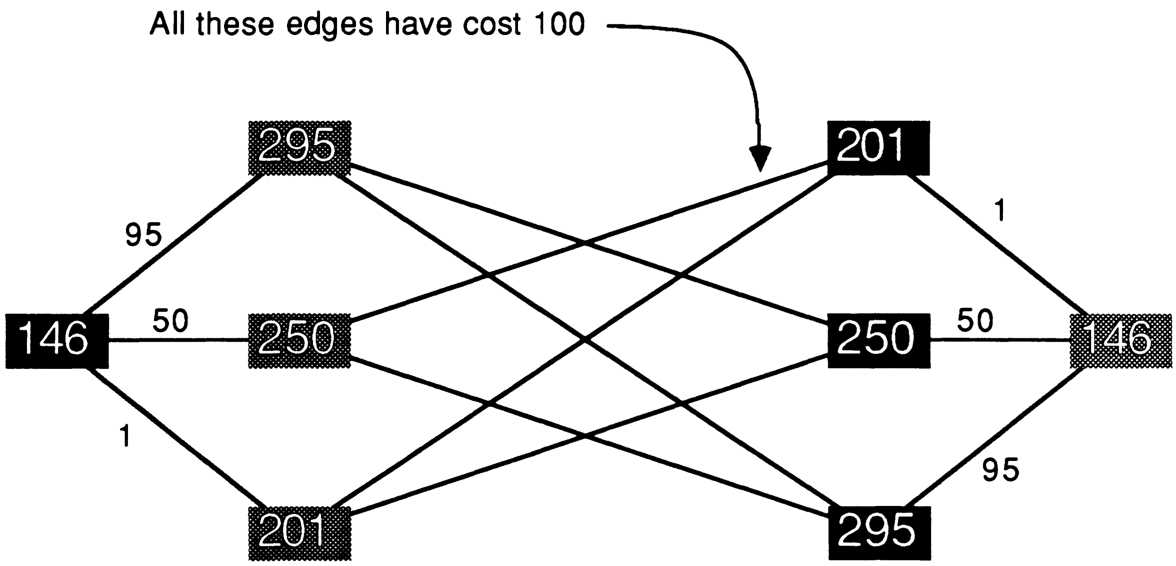
2 The Kernighan-Lin algorithm and variations

In [KL70], Kernighan and Lin propose a graph partitioning heuristic and a number of variations upon it. Here we review the basic problem and the Kernighan-Lin algorithm. We are given a graph $G = (V,E)$ with $2n$ vertices $\{1,2,...,2n\}$ and a nonnegative cost c_{ij} associated with each edge $\{i,j\}$. If $\{i,j\}$ is not in E, then $c_{ij} = 0$. An equipartition of V is a pair of sets A and B for which $|A| = |B| = n$, $A \cup B = V$ and $A \cap B = \emptyset$. The *cost* of an equipartition $\{A,B\}$ is the sum of edge costs for all edges with one endpoint in A and the other in B. The objective of the Kernighan-Lin algorithm is to find the equipartition of minimum cost. For graphs with edges of unit cost, this is equivalent to finding the equipartition with the fewest such edges. The problem is NP-hard, but in practice the Kernighan-Lin algorithm often finds very good, if not optimal, equipartitions.

For an arbitrary equipartition, we call edges of G with one endpoint in A and the other in B *external edges* and call all other edges *internal edges*. At each step of the K-L algorithm, we move a vertex from A to B and another from B to A. We exchange the vertices so that the accumulated cost of all the external edges is made as small as possible. Kernighan and Lin define $D_a = \sum_{y \in B} c_{ay} - \sum_{x \in A} c_{ax}$ for all $a \in A$ and define D_b analogously for $b \in B$. The *D-value* of a vertex is the net decrease in the cost of external edges that would result from moving the vertex from its group to the other. For a pair of vertices (a,b) in AxB, the net benefit of exchanging the vertices is $D_a + D_b - 2c_{ab}$, which Kernighan and Lin call the *gain* associated with (a,b). At each step, the K-L algorithm exchanges the pair (a,b) with the greatest associated gain.

Exchanging vertices so long as some pair of vertices has a nonnegative gain and reporting the resulting partition is a simple local search heuristic for graph partitioning. Where local search halts, however, the K-L algorithm can go on to exchange vertex pairs with negative gain. Such exchanges cause small increases in the cost of the partition at one step, but often permit large cost reductions at later steps. To prevent a pair of vertices from trading places again and again, Kernighan and Lin add the provision that in each pass of their algorithm, no vertex may move from one vertex group to the other more than once. After n exchanges, then, one pass of the algorithm terminates because all vertices have moved. The resulting partition is one in which A and B have completely changed places, resulting in no net change in cost. One pass of the algorithm reports the best of the n intermediate partitions, breaking ties by choosing the first partition of minimum cost. Another pass of the algorithm may improve further the partition. The complete K-L algorithm carries out pass after pass until some pass fails to improve the partition. Kernighan and Lin found, in their experiments on graphs of a few hundred vertices, that after two to four passes, the algorithm cannot further improve the partition, and conjecture that the number of passes required does not strongly depend on the size of the problem.

A straightforward implementation of the K-L algorithm takes time $O(n^3)$ per pass for a graph with n vertices and m edges. Calculating the initial D-values takes time proportional to the sum of the degrees of the vertices, $O(m) \leq O(n^2)$. The total time required to update D-values after exchanges of vertices is also $O(m)$, because no edge changes from internal to external, nor from external to internal, more than once. To select the next pair of vertices to exchange can conceivably take $O(n^2)$ time per selection, however. Most implementations keep the unmoved vertices in two heap data structures, one for A and one for B, ordered by decreasing D-value, on the reasonable assumption that vertices of high D-value are most likely to be promising candidates for vertex exchange. Were there some guarantee that the vertices with highest D-value included the vertices whose interchange yield the greatest gain, the selection of vertices and maintenance of the heaps could be carried out in time $O(n \log n)$ overall. However, there is no such guarantee. For an arbitrary k, graphs exist for which the vertex pair (a,b) of maximum gain, the best pair to exchange, includes none of the k vertices of greatest D-value in either group. In the example below, the vertices with D-values of 250 yield the greatest gain when exchanged, though there are vertices with larger D-values in both vertex groups. Therefore, a single vertex selection can take $O(n^2)$ time. We know of no results that suggest that the number of such time-consuming vertex selections is small; therefore, we know no bound lower than $O(n^3)$ for the selection of pairs. Perhaps there is some amortized bound better than this. At any rate, the time the algorithm takes in practice is $O(m + n \log n)$ per pass.

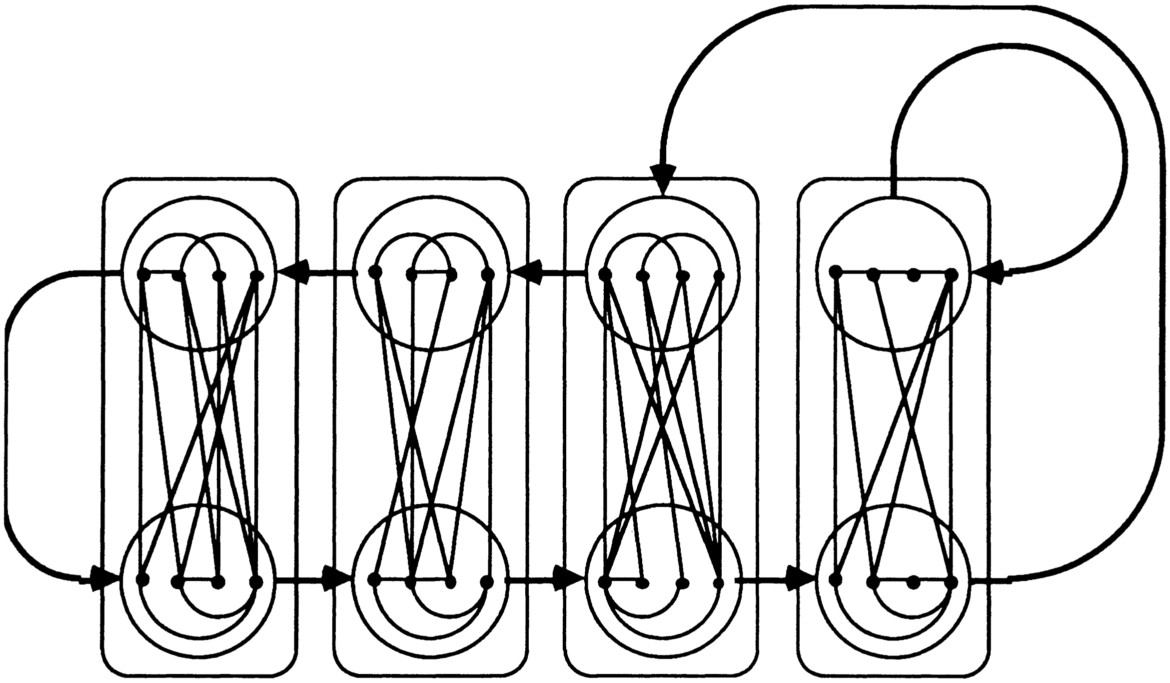


Vertices of lesser D-value can yield the greatest gain

In [FM82], Fiduccia and Mattheyses describe an algorithm similar to that of Kernighan and Lin but applied to the more general problem of reducing the number of connecting nets in a partition of cells, where a net is an entity that connects an arbitrary number of cells. Their algorithm, applied to problems in which all nets are 2-nets, solves the graph partitioning problem with unit edge costs. In the F-M algorithm, vertices are moved one at a time rather than pairwise, moving the vertex of greatest D-value at each step, subject to constraints on the ratio $|A|/|V|$. For arbitrary edge costs, this would lead to an $O(m+n \log n)$ algorithm, but Fiduccia and Mattheyses observe that with unit edge costs, the D-values of unmoved vertices are necessarily integral values in the interval $[-\text{maxD}, \text{maxD}]$, where maxD is the highest vertex degree in G. Rather than maintain a heap, then, they maintain ($2 \times \text{maxD} + 1$) linked lists of unmoved vertices of the same D-value. Maintaining these lists takes O(m) time, so the entire algorithm takes O(m) time. As with the K-L algorithm, the F-M algorithm moves all the vertices and reports the best of the intermediate partitions. Also in the style of K-L, the F-M algorithm is repeated a number of times until no further improvement occurs, and that number is conjectured to be only weakly dependent on the number of vertices of the graph. We base our algorithms on the F-M algorithm and assume unit edges costs in the remainder of this paper.

Johnson et al [JAMS88] report the results of experiments comparing the K-L and F-M algorithms, and conclude that the algorithms in practice take the same time to execute, but that the F-M algorithm fails to produce partitions as good at those of the K-L algorithm. However, there is a variant of the F-M algorithm that redefines the cost of a partition to permit the sizes of the vertex sets to differ by arbitrary amounts, but which penalizes lopsided partitions. The penalty term is proportional to the square of the difference of the vertex set sizes; that is, the variant attempts to reduce (# of external edges) + $\alpha(|A| - |B|)^2$ with each vertex movement. Moving $\frac{||A|-|B||}{2}$ vertices with large D-values from the larger to the smaller of the vertex sets converts the best partition found by this version of the algorithm to an equipartition. Johnson et al find that the variant performs as well, but no better, than K-L for an appropriate choice of α, 0.05, chosen because it works well over a range of graph sizes and densities in their annealing experiments. It performs better than the straightforward F-M algorithm because the penalty function allows the algorithm to escape from undesirable local minima better. Its worst case time, as well as its running time in practice, is linear in the size of the graph.

Other techniques for solving the graph equipartitioning problem include simulated annealing, extensively discussed in [JAMS88], network flow and matching techniques [GB83], and several variations on the Kernighan-Lin algorithm. Researchers have also considered the effect of the apparently random choices made in the execution of the K-L algorithm [K84]. For example, when several pairs of vertices have the same high gain, choosing for exchange the pair that most recently attained that high gain value may give better partitions than would be obtained by making an arbitrary choice.



Movement of vertex groups in the round-robin algorithm

3 Multiway partitions and the round-robin algorithm

In their original paper, Kernighan and Lin suggest that to achieve a good k-way equipartition, one should arbitrarily divide the graph into k pieces of equal size and perform the standard K-L algorithm on the pieces in pairs until no application of K-L to any pair results in an improvement. This is an approach that is readily parallelizable, because the k/2 pairwise applications of K-L can be performed in parallel on k/2 processors. Only the scheduling of the pairwise K-L processes needs to be specified to have a viable parallel algorithm.

The implementation of a parallel algorithm for the pairwise application of either the Kernighan-Lin algorithm is straightforward. The hypercube topology has embedded within it a ring; that is, there is a sequence of processors P_0, P_1, ..., P_{n-1}, P_p=P_0, for which P_{i-1} and P_i are directly connected by a communication link for $1 \leq i \leq p$. There are many more connections within a hypercube, but we require only these for our partitioning algorithm. We schedule the pairwise applications of the two-way partitioning algorithm on a chain of processes as we would schedule a round-robin chess tournament

upon a row of chessboards. One player is stationary; he sits on the same side of the same table, at one end of the row of tables, throughout the tournament. All the other players move, after each match, to the right, moving around to the other side of the table if necessary. If k is odd, then the stationary player is "bye", the null player. After k–1 games, each player has returned to his original seat having played every other player once. Instead of players and chess games, we have groups of vertices and applications of a pairwise partitioning algorithm. The movement of vertex groups from processor to processor is illustrated in the figure above. After each process has witnessed k–1 consecutive applications of the partitioning algorithm with no reduction in the number of external edges, the algorithm can terminate. The number of processes is independent of the number of processors; the ith process is assigned to processor
$P_{i \bmod p}$.

To start the round-robin algorithm, the host processor sends to each process a pair of vertex groups and other assorted information, then awaits the results of the algorithm. A vertex group consists of the identifying labels of its vertices and a list, for each vertex in the group, of adjacent vertices in the graph. No process ever stores a copy of the entire graph, so the size of problems that may be solved is not limited by the memory size of any one processor. Besides the vertex groups, the most critical datum passed from process to process is information about when the meeting of a pair of vertex groups elsewhere has produced a profitable vertex exchange. Every k–1 rounds, each vertex group encounters every other, so after k–1 rounds without an improvement, no more improvement is possible and the round-robin algorithm can terminate. However, to accurately determine the time of the most recent vertex exchange after every round requires a series of broadcasts of small integer messages that would significantly increase message traffic. Instead, each vertex group carries with it news of a recent vertex exchange, and when two vertex groups meet, the one with the more recent news shares it with its less up-to-date counterpart. A process that performs a vertex exchange increases this value before passing it on to adjacent processes. Since it can take p–1 rounds for news of an exchange to propagate from one end of the row of processors to the other, the round-robin algorithm ends when each process knows that the last vertex exchange was k+p–2 rounds ago.

K-way partitions are useful in the same circuit layout, sparse matrix and other problems that make 2-way partitions worthy of study. Even if they weren't interesting in their own right, they would be useful because they can be used to generate good 2-way partitions. Kernighan and Lin suggest that when their basic algorithm is least likely to find a good partition it is because the algorithm must exchange roughly half the vertices to get the best partition. The algorithm does not look far enough ahead to find the appropriate vertex exchanges. In this case, they say, it is worthwhile to find a 4-way partition, combine the pairs of parts in each of the three possible ways, and return the best of the three 2-way partitions so constructed. We suggest that starting with k-way partitions for larger k allows us to construct even better 2-way partitions.

The complexity analysis of a single round of the algorithm is straightforward. The communications complexity of a round is linear in the size of the graph, since the union of all the messages passed in a round essentially is the graph itself. The time complexity of one round of a single process depends on the 2-way partitioning algorithm that process applies and on whether memory is scarce. One step of the algorithm on a particular process begins when two lists of vertices and adjacency lists become available. From these two pieces of the original graph, we construct the subgraph induced by the vertices of the two pieces, partition that subgraph and rebuild the vertex groups to reflect any exchanges that have occurred. The time complexity of constructing the subgraph depends on what data structure we use for implementing a membership test; when scanning the adjacency list of vertex v, the algorithm must determine for each vertex adjacent to v in the graph whether it is in one of the vertex groups present at this process. At the cost of O(N) space, where N is the number of vertices in the original graph, the membership test takes constant time and the constructing the subgraph takes O(m+n) time, where n is the number of vertices in the two vertex groups and m is the size of the adjacency lists. To avoid using more than O(m+n) space requires a membership test based on binary trees that takes O(log n) time. The time to construct the subgraph is then O(m + n log n). We use variations of the Fiduccia-Mattheyses algorithm for generation 2-way partitions. The partitioning algorithms take O(m' + n) time [FM82], where $m' \leq m$ is the number of edges in the subgraph, if we are willing to use O(maxD) space, where maxD is the highest vertex degree in the graph. When space is scarce, O(m' + n log n) time, O(m' + n) space implementations are possible. Finally, rebuilding the vertex groups to send on to adjacent processors requires little more than a simple linear time pass. So, if O(N) space is not a problem, the time for one round of the algorithm for one process is linear in the size of the messages sent to the process.

This complexity analysis is incomplete in several ways. Although we have shown that the time a process spends in a round is linear in the problem size, we have not bounded the problem size. If the problems vary widely in size, large problems will create bottlenecks and severely reduce parallelism. More importantly, we have not demonstrated that our round-robin algorithm terminates. Indeed, for inappropriate 2-way partitioning algorithms the algorithm may not terminate. We cannot address these issues until we describe in more detail the 2-way partitioning algorithms we apply with each process.

4 Relaxing problem constraints

Most applications do not require the two sets of vertices to be of exactly equal size. It is not difficult to construct graphs for which an equipartition requires considerably more external edges than does a slightly lopsided partition. For example, consider the graph with two connected components, one an n+1-clique and the other an n–1-clique. Such a graph has a partition into two pieces of almost equal size and no connecting edges, but the best equipartition has n edges connecting the pieces. For many applications, the partition into pieces of not-quite-equal size is the better one.

Cost functions with penalty terms, like the one Johnson et al use in their implementation of the F-M algorithm, provide a natural way to quantify the trade-off between balance in size of vertex groups and reduction in the number of connecting edges. In their cost function, α is a parameter that may be increased to emphasize the importance of keeping the sizes of the vertex sets the same, or decreased to emphasize the importance of reducing the number of external edges. At one extreme, $\alpha=0$ results in a partition in which balance is irrelevant; the best partition is the one with one of the vertex sets empty. At the other, extremely large values of α prevent any intermediate partition from getting more than one vertex out of balance.

Another way to combine the goals of balanced size and few external edges is to modify the F-M algorithm to try to minimize the *density* of the edges connecting the two pieces, where density(A,B) = $\frac{\text{(\# of external edges)}}{|A|\ |B|}$. As the cost function of Johnson et al is parameterized by a scalar α, the density may also be parameterized. If we raise the denominator of the density function to the power α, then large values of α inhibit imbalance and small values permit it. At one extreme, $\alpha = 0$ allows complete imbalance, with all vertices in one of the two vertex groups. At the other, as $\alpha \to \infty$, the importance of keeping the denominator large, i. e. the sizes balanced, grows more dominant. Lower bounds on set size may also be incorporated into the density function by using $(|A|-\min)^{\alpha}(|B|-\min)^{\alpha}$ as the denominator, thereby making partitions with set sizes near the lower bound very costly. For any of these cost or density functions, the only candidates for exchange at any step are the two vertices, one in each vertex group, of highest D-value. Thus, our algorithms need not be changed significantly to use different cost functions.

Even the stricter equipartitioning problem may be solved by the use of these balance-conscious cost functions. Starting with a small value of the parameter α allows for better escape from bad local minima, as Johnson et al observed. Increasing α from one iteration of the algorithm to the next, much as the temperature is reduced from one iteration of simulated annealing to the next, increases the importance of balance in the partition until the partition is completely balanced. In fact, Johnson's greedy algorithm for restoring balance to an unbalanced partition can be viewed as increasing the value of α very slowly just until complete balance is achieved.

5 Comparing multiway partitions

When an multiway equipartition is required, it is clear that locally reducing the number of external edges between a pair of pieces of the graph and globally reducing the number of external edges between different pieces of the graph are compatible goals. For suppose that in the process of carrying out the K-L algorithm on vertex sets A and B, vertices $v_A \in A$ and $v_B \in B$ are exchanged. It is possible that the number of edges connecting vertex set A to some vertex set C increases as a result of the exchange, because v_B is adjacent to some vertex in set C and v_A is not. In that case, however, the number of edges connecting vertex sets B and C decreases by the same amount. The net change in the number of external edges with an endpoint not in $A \cup B$ is zero, so the exchange of vertices to reduce the number of pairwise external edges can only reduce the total number of external edges.

To discuss pairwise and multiway partitions, we develop some notation. Let $c_k : V \rightarrow \{0,...,k-1\}$ denote the partition function; that is , $v_1 \in V$ and $v_2 \in V$ are in the same piece of the graph iff $c_k(v_1) = c_k(v_2)$. Let V_i denote $\{v \in V \ni c_k(v) = i\}$. We let $cost(V_i,V_j)$ denote real-valued function of a pair of disjoint vertex sets V_i and V_j and let $COST(c_k)$ denote a real-valued function of a vertex partition that corresponds, in a reasonable way, to cost. To write $cost(V_i,V_j,G)$ and $COST(c_k,G)$ would be more accurate, but we shall leave implicit the reference to the graph G. For example, if $xe(V_i,V_j)$ denotes $|\{\{a,b\} \in E \ni a \in V_i,\ b \in V_j\}|$, then $XE(c_k)$ is the number of external edges in the k-way partition c_k. The conclusion of the previous paragraph, then, is that exchanging vertices of V_i and V_j to reduce $xe(V_i,V_j)$ reduces $XE(c_k)$ as well. We call xe and XE *compatible* functions because exchanging vertices to reduce the former for any pair of vertex sets reduces the latter for the vertex partition.

A multiway partition in which the size constraints are relaxed requires careful consideration. It is easy to devise cost functions f for pairwise optimization for which there are no compatible functions F of all vertex sets. One such function is the density function defined above, as

$$density(V_i,V_j) = \frac{xe(V_i,V_j)}{|V_i||V_j|}.$$

A reasonable generalization of the pairwise density might be

$$DENSITY(c_k) = \frac{XE(c_k)}{\sum_{i \neq j} |V_i||V_j|},$$

but these functions are not compatible. Reducing the imbalance between a pair of vertex sets at the cost of an extra external edge may reduce $density(V_i,V_j)$ if the relative change in $|V_i||V_j|$ is large enough. The relative change in $\sum_{i \neq j} |V_i||V_j|$ is not so large, however, so the extra edge increases $DENSITY(c_k)$. This incompatibility is more than inconvenient. Indeed, it is possible for there to exist vertex sets A, B and C and a vertex v such that $density(A \cup \{v\},B) > density(A,B \cup \{v\})$, $density(B \cup \{v\},C) > density(B,C \cup \{v\})$ and $density(C \cup \{v\},A) > density(C,A \cup \{v\})$. Therefore an algorithm that waits for vertex exchanges to stop can fail to terminate, as vertex v is passed from one vertex set to another forever. Thus, there is no function of a vertex partition compatible with the density function. Workarounds are possible; for example, one may use density to choose the next vertex to move at each step, but only select, finally, from the intermediate partitions for which the value of xe has not increased. Nevertheless, a local cost function with a compatible global counterpart is more convenient, because only such cost functions allow us to guarantee termination of the round-robin algorithm.

The cost function of Johnson et al, $f_\alpha(V_i,V_j) = xe(V_i,V_j) + \alpha(|V_i| - |V_j|)^2$, is not so badly behaved. First, we observe that we do not have a family of distinct cost functions

$$f_{\alpha,\beta}(V_i,V_j) = xe(V_i,V_j) + \alpha(|V_i|^2 - 2\beta|V_i||V_j| + |V_j|^2),$$

for each pair (α,β), as it might seem. The polynomial

$$s_{\alpha\beta}(V_i,V_j) = \alpha(|V_i|^2 - 2\beta|V_i||V_j| + |V_j|^2)$$

is in fact a function of one variable because $|V_i| + |V_j|$ is a constant for any pair of vertex sets V_i and V_j. Therefore, letting x denote $|V_i|$ and $(n-x)$ denote $|V_j|$, we have

$$s_{\alpha\beta}(x) = \alpha(x^2 - 2\beta x(n-x) + (n-x)^2)$$

which, after rearranging terms, is

$$s_{\alpha\beta}(x) = 2\alpha(\beta+1)(x^2 - nx) + \alpha n^2.$$

Thus, if $\alpha(\beta+1) = \chi(\delta+1)$, then $s_{\alpha\beta}(x)$ and $s_{\chi\delta}(x)$ differ by a constant and the functions are equivalent as penalty functions for measuring the balance of a partition. Although the formulation given by Johnson et al is computationally efficient, requiring only two multiplications, in the analysis below we shall find it more convenient to use forms for which $\beta=0$. Thus, we shall use

$$f_\alpha(V_i,V_j) = xe(V_i,V_j) + s_\alpha(V_i,V_j),$$

where

$$s_\alpha(V_i,V_j) = \alpha(|V_i|^2 + |V_j|^2),$$

as the cost function in the discussion that follows.

If we let

$$F_\alpha(c_k) = XE(c_k) + S_\alpha(c_k)$$

where

$$S_\alpha(c_k) = \alpha \sum_{0 \le i < k} |V_i|^2,$$

we find that f_α and F_α are compatible. Consider exchanging vertices in a specific V_i and V_j to reduce $f_\alpha(V_i,V_j)$. We may separate the terms that involve V_i and V_j

$$F_\alpha(c_k) = xe(V_i,V_j) + \alpha(|V_i|^2 + |V_j|^2) + \sum_{(k,l)\neq(i,j)} xe(V_k,V_l) + \alpha \sum_{k\neq i,j} |V_k|^2$$

and see that the first two terms are exactly $f_\alpha(V_i,V_j)$, the third term does not change as vertices move between V_i and V_j, as discussed previously, and the last term does not depend at all on V_i or V_j. Therefore, any vertex exchange between V_i and V_j that reduces $f_\alpha(V_i,V_j)$ inevitably reduces $F_\alpha(c_k)$ as well. Thus, F_α is compatible with f_α.

To this point, we have developed techniques for comparing the quality of k-partitions for a fixed k. We also need tools to compare two partitions when the number of vertex groups can differ among them. For some applications, the value of k need not be determined in advance. For example, in parallel block Cholesky factorization, a partition into many vertex groups allows the algorithm to exploit the inherent parallelism of the problem at a fine-grained level, while a partition into few groups allows for relatively less parallelism. No aspect of the problem, however, requires a partition into, say, 48 parts and forbids a partition into 49 parts. Instead, the communication characteristics of the given parallel machine make tasks, and thus vertex groups, below a certain size inefficient, because computation time fails to completely overlap communication time. A good partition for such a problem is one in which the number of external edges is small and the number of vertices in each group is not too far from some value that depends on the problem and hardware. The number of vertex groups is less important. Thus, a graph with nine identical connected components need not be divided into eight pieces with size imbalances and several external edges just to meet an artificial problem constraint.

First, we consider using F_α to measure the quality of a partition when k is variable. We note that an equivalent way to express F_α is as

$$F_\alpha(c_k) = XE(c_k) + \alpha k(\mu^2 + \sigma^2),$$

where σ is the standard deviation in the sizes of the vertex sets and μ is the mean vertex set size. The term $\alpha k\mu^2$ is inversely proportional to k; therefore, making the number of vertex sets smaller and more numerous always reduces that term. The cost function $S\alpha(c_k)$, originally designed to keep the pieces of the partition balanced in size, has the side effect in this case of increasing the number of pieces in the partition. The more we increase α to maintain balance, the more we inadvertently increase the value of k. A reasonable adjustment to

F_α to push vertex set sizes toward some "good" value M is

$$F_{\alpha M}(c,k) = XE(c) + \alpha k((\mu - M)^2 + \sigma^2),$$

Using this function and a suitable pairwise equivalent like

$$f_{\alpha M}(V_i,V_j) = xe(V_i,V_j) + \alpha(|V_i| - M)^2 + \alpha(|V_j| - M)^2,$$

would seem to penalize partitions which vary too far from M in the average size of vertex groups. However, the new term we've added to f_α, $2\alpha M(M - |V_i| - |V_j|)$, does not change when a vertex is moved from one group to the other; for purposes of pairwise vertex exchange, $f_{\alpha M}$ and f_α are equivalent. $F_{\alpha M}$ and F_α differ only by $\alpha kM^2 - 2\alpha nM$; the first term depends only upon the the number of vertex groups in the partition and the second is constant. This suggests that an equivalent choice for F is

$$F_{\alpha M}(c, k) = XE(c, k) + S_\alpha(c, k) + \alpha kM^2,$$

and that it is compatible with f_α. Other cost functions are certainly possible, but this cost function, which allows us to control loosely the average vertex set size, seems to us most satisfactory for those applications, like parallel block Cholesky factorization, that allow the number of groups in the partition to vary.

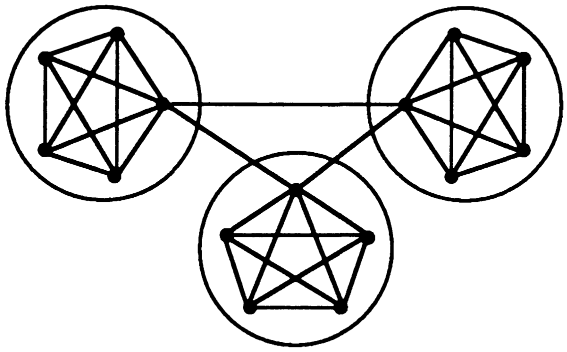
6 Variable-sized partitions and the round-robin algorithm

A simple way to find the best k-way partition when k can vary is to find the best k-way partition for particular values of k and choose the best of them. A better way is to modify our round-robin algorithm to permit vertex groups to split and merge as they move through the ring of processors, so that the number of vertex groups k is allowed to vary. Of course, every split or merger must reduce $F_{\alpha M}$. A process does not need to know the current value of k, however, to know whether a split or merge is profitable. By the formula for $F_{\alpha M}$ in the previous section, merging V_i and V_j is worthwhile when $f_\alpha(V_i,V_j) > f_\alpha(V_i \cup V_j, \varnothing) - \alpha M^2$. That is, the algorithm should merge the groups when the best two-way partition of them is too close in cost to the most lopsided partition. The value αM^2 is a sort of bonus that encourages vertex groups to merge. Likewise, a vertex set should split if the set can be partitioned into subsets V_i and V_j with $f_\alpha(V_i,V_j) < f_\alpha(V_i \cup V_j, \varnothing) - \alpha M^2$. When the algorithm follows these rules for splits and mergers, the value of FaM is guaranteed to decrease with every change to the partition, and the termination of the round-robin algorithm is still guaranteed.

Recall that we described the round-robin algorithm by comparing it to a chess tournament played on a row of tables, with one of the players occupying the same seat at one end of the row throughout the tournament. We designate the end of the row of processes at which the corresponding vertex group resides the right end. The process on the right end has special responsibilities in the algorithm. However, the process at the right end changes as the algorithm progresses. When the number of vertex groups increases to an odd number, we no longer have enough processes to handle all the vertex groups simultaneously, so the algorithm must create a new process to accommodate the new vertex group. This is the role of the rightmost process, which notifies the processor to its right that it must create a new rightmost process. The number of vertex groups increases only when a vertex group is split, so it is most convenient to have such vertex splits occur on the rightmost processor only. Since each vertex group is on the rightmost process periodically, this guarantees that every vertex group that is a good candidate for splitting gets split.

The decision to merge or not to merge a pair of vertex groups is necessarily made with every meeting of a pair of groups. Leaving that responsibility to the rightmost process is impossible because not every pair of vertex groups meet there. If a merger is deemed profitable, the union of the vertex sets is sent to the left and the empty vertex set is sent to the right. When that empty set arrives at the right end, the rightmost process notes the decrease in the number of groups. If that number has gone from even to odd, the stationary vertex group and the empty vertex group change places, so that a nonempty vertex group is sent back to the left. If that number has gone from odd to even, then the rightmost process hosts two empty groups, and the number of processes necessary to serve all the nonempty vertex groups has decreased. Therefore, the rightmost process tells its neighbor to become the new rightmost process and then terminates.

The ability to change the number of vertex groups complicates slightly the decision processes must make about when to halt. If the processes pass, with each vertex group, the latest information about the number of vertex groups, then that information propagates from the rightmost process to the left in time to keep the algorithm from taking any extra unneeded rounds. With the use of



A good 3-way partition with a dense quotient graph

compatible cost functions, we can now, as we could not before, guarantee that the round-robin algorithm halts after a finite number of rounds.

7 Graph partitioning for sparse quotient graphs

A related, but distinctly different graph partitioning problem is that of partitioning a graph so that the induced quotient graph is sparse. To see how this problem differs from the usual partitioning problem, consider the graph above. The illustration suggests a good 3-partition of the graph, but one that induces a complete quotient graph on three vertices. The movement of any single vertex from one vertex set to another makes a less balanced partition with more external edges. However, moving one of the three articulation points to a different vertex set changes the partition to one with an induced quotient graph of only two edges.

The particular application that motivates this problem is block-oriented Cholesky factorization. In block-oriented Cholesky factorization, computational tasks are associated with the vertices, edges, and certain paths in the quotient graph. Roughly speaking, the complexity of a task (i,j) is proportional to the product $|V_i||V_j|$ and does not depend on $xe(V_i,V_j)$, so the need to keep the quotient graph sparse is greater than the need to keep the connection between any two pieces sparse.

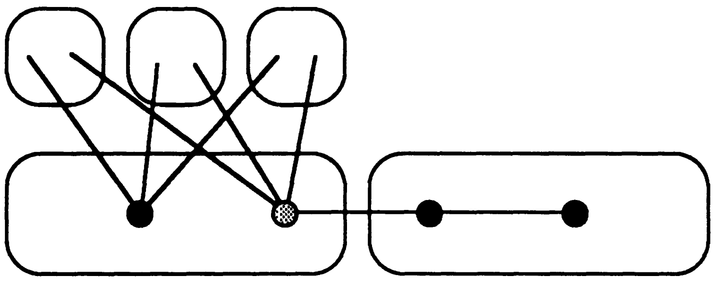
The desire to keep the partition balanced does not change when considering this new problem, but the connectivity terms xe and XE are no longer relevant, so we define

$$qe(V_i,V_j) = 1 \text{ if } xe(V_i,V_j) > 0, 0 \text{ otherwise}$$

and

$$QE(c,k) = \sum_{i \neq j} qe(V_i,V_j).$$

Unfortunately, qe and QE are not compatible. In fact, a local change that reduces $qe(V_i,V_j)$ can increase $QE(c_k)$ by an arbitrary amount, as in this example:



The number of quotient graph edges can increase globally as it decreases locally

Moving the gray vertex from the left group to the right group disconnects the two groups, but creates many new connections between the right group and other groups. Apparently, we need more than just local information about a pair of vertex sets to make a good decision.

Let us assume, for the moment, that any process can compute c(v) in constant time for any vertex v. Consider then a particular vertex set V_h that is distinct from the sets V_i and V_j that a particular process is examining. All those vertices in $V_i \cup V_j$ which are adjacent to vertices in V_h should, in the absence of other considerations, be together in either V_i or V_j so that only one edge connects V_h to $V_i \cup V_j$ in the quotient graph. This applies equally for all vertex sets other than V_i and V_j. The interconnections between V_i and V_j matter only insofar as how many connected components there are in the subgraph induced by V_i and V_j; if the subgraph is connected, no rearrangement of vertices can eliminate the quotient graph edge between V_i and V_j. If there are several connected components, each should, ideally, reside completely in either V_i or V_j.

The partitioning algorithm we apply, then, is a hypergraph partitioning algorithm. We construct a hypergraph with $|V_i|+|V_j|$ vertices and up to $k+C-2$ nets, where k is the number of vertex sets and C is the number of connected components in the subgraph induced by V_i and V_j. Up to $k-2$ of the nets correspond to vertex sets on other processes, where the vertices on net N_h are all the vertices in $V_i \cup V_j$ with at least one adjacent vertex in V_h. The other C nets correspond to connected components of the induced subgraph, with all vertices in a component on the corresponding net. For each net that can be completely moved into one of V_i and V_j, we have a single quotient graph edge rather than two. Any algorithm for hypergraph partitioning, such as the more general version of the F-M algorithm, may be applied to exchange vertices to reduce the number of external nets. Thus, rather than attempt to reduce $qe(V_i,V_j)$, we attempt to reduce

$$ce(V_i,V_j) + \sum_{h \neq i,j}(qe(V_i,V_h) + qe(V_j,V_h))$$

with each vertex exchange, where $ce(V_i,V_j)$ is the number of connected components of the induced subgraph with vertices in each of V_i and V_j. When the induced subgraph is connected or consists of two connected components, ce = qe, and the function is compatible with QE. When the induced subgraph consists of more than two connected components, the compatibility is not so clear, but it is reasonable to expect that this potential problem will rarely arise.

The main problem, then, is how to compute c(v) for any vertex v. Without extensive communication, exact knowledge is impossible. A good approximation, however, is obtainable with no extra communication at the cost of O(N) extra space. For each vertex v in the graph, record c(v) as of the last time vertex v passed through this processor. At the beginning of the calculation, as vertex exchanges are relatively frequent, the accuracy of the information will degrade quickly with time, but as the calculation progresses and the number of exchanges grows smaller, the information will become more and more accurate. We will experiment with these ideas and report on the results in a future technical report.

8 Experimental results

We have implemented the round-robin algorithm as described in the previous sections. The code, written in C, runs on both the Cornell Theory Center's Intel iPSC hypercube, running Xenix and version 3.0 of the node operating system, and on the Oak Ridge National Laboratory's hypercube simulator [D86], on a Vax 780 under Berkeley Unix. We have used the simulator to experiment with partitioning a number of the graphs from [E79], which are finite element meshes for various physical structures, and several grid graphs. We summarize important properties of these graphs in Table 1. Although the structural graphs are more interesting in a sense, the simplicity of grid graphs offers us some hope that we can find optimal partitions, or at least lower bounds on the values of optimal partitions, for grid graphs, to make comparisons to the results of our algorithm more meaningful.

Problem	Source	Vertices	Edges	Density
1	21 x 34 grid	714	1373	0.54%
2	27 x 34 grid	918	1775	0.42%
3	29 x 31 grid	899	1738	0.43%
4	valve	869	3208	0.85%
5	beam	918	3233	0.77%
6	baseplate	1005	3808	0.75%
7	simple grid	1007	3784	0.75%
8	sea chest	1242	4592	0.60%
9	destroyer	2680	11173	0.31%

Table 1: Test problems.

For each application of the algorithm, we record number of vertex sets in the partition produced by our algorithm, the balance in the vertex set sizes, as measured by their standard deviation, the number of external edges, the number of quotient graph edges, and the execution time measured in rounds. A round begins when a process receives two vertex groups and ends when it forwards them to another process. The number of rounds after the last vertex exchange is not included in the running time figure. Table 2 summarizes the running time and partition quality for one application of the round robin algorithm to each of our test problems. For these applications, we have arbitrarily chosen $\alpha = 0.1$ and $M = 50$.

The time to partition the grid graphs should not be taken as representative of the speed of the algorithm; rather, the initial ordering for these graphs happens to be close to a local minimum. The structural graphs give a more accurate picture of typical running times. The running time on those graphs is roughly quadratic in the number of vertex groups. Counting rounds to measure time makes large and small vertex sets seem equally time-consuming, so it is the number of vertex sets, not the number of vertices, that affects the number of rounds the algorithm must complete.

Problem	Vertex Groups	Quotient Edges	Standard deviation	External edges	Function value	Time
1	14	19	0.00	225	226.4	1
2	18	25	0.00	299	300.8	1
3	16	31	0.53	237	298.7	24
4	16	27	1.99	427	463.1	53
5	18	46	1.83	506	513.8	60
6	20	51	1.64	706	711.5	96
7	20	40	0.73	622	623.3	94
8	24	66	1.48	797	809.6	173
9	52	169	1.35	2312	2333.8	540

Table 2: Sample partitions for test problems.

Even if the grid graphs hadn't had ridiculously easy-to-find partitions, partitioning the same graphs from a different starting partition would be a worthwhile exercise. Many partitioning algorithms are sensitive to the quality of the initial partition, and the problem of finding good starting partitions is interesting in its own right. Often the graph comes automatically with a good initial partition arising naturally from the problem domain. The results in Table 3 are obtained by randomly relabelling the vertices of the graph, then applying the round-robin algorithm as before. The results suggest that the round-robin algorithm is not particularly sensitive to starting partition in terms of the quality of the final partition. It is as likely to do better as it is to do worse. However, a randomly labelled graph does apparently require more time to partition than one that has come directly from an environment in which a reasonably good starting partition is likely.

Problem	Vertex Groups	Quotient Edges	Standard deviation	External edges	Function value	Time
1	14	34	0.65	187	189.0	70
2	18	62	0.67	290	292.6	94
3	16	44	1.18	241	304.5	89
4	16	34	1.04	459	490.5	92
5	18	50	1.41	502	507.4	224
6	20	58	2.00	763	771.1	73
7	20	49	1.15	649	651.9	156
8	24	74	1.61	851	864.6	278
9	52	168	1.56	2284	2309.0	582

Table 3: Partitions of randomly relabelled test problems.

Table 3 and 4 give partition data that results from varying the parameters α and M, respectively, in the fourth test problem. It comes as no surprise that increasing α forces down the standard deviation of the sizes of the vertex sets, nor that decreasing it allows substantial reductions in the number of external edges. It is less expected that, with one glaring exception, increasing α reduces the running time of the algorithm. But consider that for very large values of α, substantial rearrangements of the graph are impossible, and a graph can freeze into a locally optimal partition fairly quickly. We expect that large values of M decrease the number of rounds the algorithm requires, because larger vertex sets inevitably mean fewer of them. The evidence mildly supports this conclusion, but is hardly convincing.

α	Vertex Groups	Quotient Edges	Standard deviation	External edges	Function value	Time
0.0125	14	20	7.82	347	383.21	95
0.0250	16	29	4.57	406	421.78	97
0.0500	16	33	1.89	434	451.75	143
0.1000	16	34	1.04	459	490.50	92
0.2000	16	48	0.77	537	598.40	77
0.4000	16	33	0.85	453	576.60	75

Table 4: The effect of varying the balance factor α.

Target	Vertex Groups	Quotient Edges	Standard deviation	External edges	Function value	Time
20	42	89	1.86	776	792.5	328
30	28	59	1.72	616	627.3	150
40	20	46	1.24	537	563.9	189
50	16	34	1.04	459	490.5	92
60	14	30	1.28	433	441.3	160
70	12	29	1.98	441	452.7	110

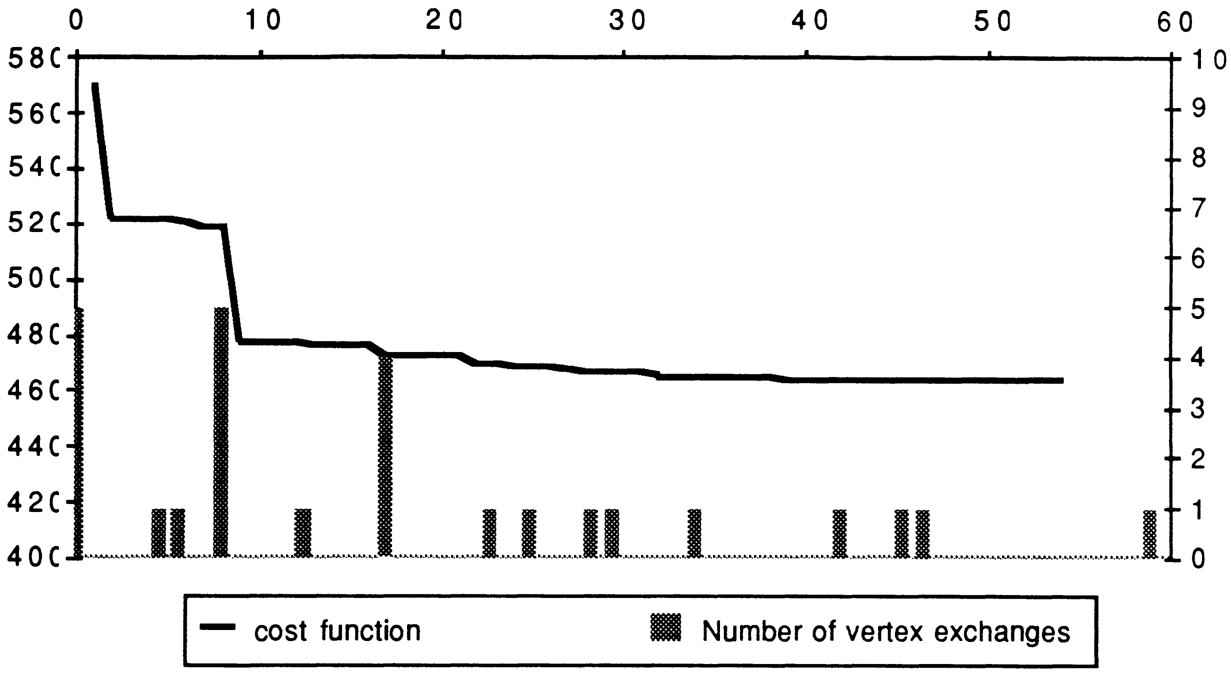
Table 5: The effect of varying the target vertex set size M.

Problem	Target	Quotient Edges	Standard deviation	Ext. edges	Ext. Edg. [GZ87]	Function value	Time
Not randomly relabelled							
1	40	35	0.48	202	296	236.6	57
2	55	32	1.05	261	374	271.8	24
3	50	31	0.53	237	408	298.7	24
4	50	27	1.99	427	492	463.1	53
5	55	37	0.99	453	587	463.6	78
6	60	37	1.29	608	696	623.3	91
7	60	29	0.83	546	660	560.9	117
8	70	46	1.93	733	879	832.0	98
9	155	43	1.58	1412	2087	1666.0	98
Randomly relabelled							
1	40	46	0.99	216	348	251.8	82
2	55	48	0.99	249	335	259.6	112
3	50	44	1.18	241	406	304.5	89
4	50	34	1.04	459	621	490.5	92
5	55	50	0.99	513	680	523.6	65
6	60	40	1.63	623	874	639.9	153
7	60	33	1.68	524	797	542.3	117
8	70	50	1.41	674	1006	770.2	170
9	155	70	0.79	1661	2097	1912.00	186

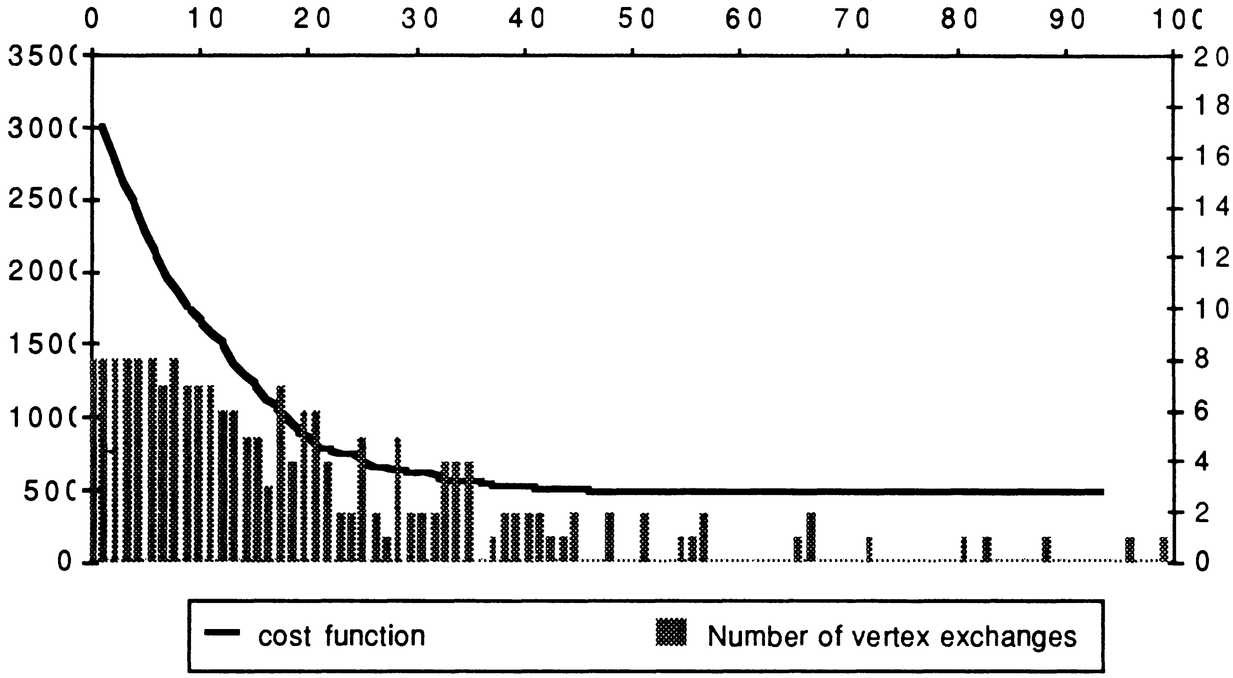
Table 6: Comparing to the partitions of [ZG87].

Because consideration of the partitioning problem is usually restricted to single-processor graph bisection algorithms, there are few algorithms that we can meaningfully compare to the round-robin algorithm. The divide-and-conquer Kernighan-Lin algorithm of Zmijewski [GZ87],[Z88] is the algorithm most like ours in its intent, so we compare round-robin partitions with parallel-K-L partitions in Table 7. It is important to note that the parallel K-L algorithm is restricted to equipartitions throughout the execution of the algorithm, so cannot escape from bad local minima as well as can the round-robin algorithm. Furthermore, that algorithm was implemented with the goal of reducing communications overhead; the round-robin algorithm is more communications-intensive than the parallel-K-L algorithm. Still, it is encouraging that when compared on our test problems, both randomly relabelled and not, the round-robin algorithm consistently produced fewer external edges and displayed less sensitivity to bad starting partitions. All partitionings were into 16 vertex groups, since the parallel-K-L algorithm deals exclusively with partitionings into powers of two. Trial and error supplied values of M that generate 16-way partitionings in the round-robin algorithm.

Finally, we briefly explore the question of how the quality of the partition increases with time. The plots below illustrate the progress of the round-robin algorithm from round to round on the fourth test problem. The first table indicates that the default starting partition for that problem is a good one; only two of the rounds dramatically enhance the partition, and the partition improvements after the first ten rounds amount to little. The same graph with a random node ordering exhibits much steadier progress at first, but after the first half of the rounds is over, little progress is made.



A graph with a meaningful ordering converges quickly, at first.



A randomly relabelled graph converges slowly to a locally optimal partition.

We included three grid graphs among our test problems because we can reasonably estimate the cost of the best partition for a grid graph, if we make enough simplifying assumptions. First, we disregard the inevitable difficulties at the boundary of the grid and any divisibility problems as well. We assume that we are forming vertex groups in the center of a very large grid. This uniformity allows us to further assume that our only task is to pick an appropriate rectilinear polygon that can tile the infinite plane. The unit squares within this rectilinear polygon correspond to vertices and when unit squares share an edge the corresponding vertices share a vertex. The balance component of our cost function is $\alpha(V-M)^2$ for each vertex group with V vertices. The external edge component for each vertex group is proportional to the perimeter of the corresponding rectilinear polygon. For rectangular vertex groups this is clear; a 2 X 3 subgrid is incident on 10 external edges. Since each one gets counted twice, each 2 X 3 subgrid is associated with 5 external edges. For each vertex set size V, there is some rectilinear polygon of perimeter $2\lceil 2\sqrt{V}\rceil$ that can tile the plane, and this is the least possible perimeter. To construct this rectilinear polygon, begin with a $\lceil \sqrt{V}\rceil$ by $\lfloor \sqrt{V}+\frac{1}{2}\rfloor$ rectangle, which is necessarily large enough to hold at least V unit squares, and remove unit squares from one corner until V squares remain. The resulting polygon can tile the plane and is responsible for $\lceil 2\sqrt{V}\rceil$ edges. Thus, one vertex set of size V is responsible for at least $\lceil 2\sqrt{V}\rceil + \alpha(V-M)^2$ cost, and since the number of sets is inversely proportional to V, we want a V that minimizes $\frac{\lceil 2\sqrt{V}\rceil + \alpha(V-M)^2}{V}$. Unfortunately, even if we could throw away the brackets and make this function continuous in V, we would still have to solve a quartic equation to find the exact value of the minimizing V in terms of α and M.

For the values used in our experiments, $\alpha = 0.1$ and $M = 50$, the best rectilinear polygon is one with area 51 and perimeter 30, for a cost of 15.1 per vertex set. Put another way, in the ideal vertex partition, all vertex sets have size 51 and cost 0.1 for the slight size imbalance, and 14.7% of the edges in the graph are external edges. If the original grid is $N_1 X N_2$, then the original graph has $N_1N_2-N_1-N_2$ edges and the best possible partition must cost at least $0.147(N_1N_2-N_1-N_2)$ for edges and $0.1\frac{N_1N_2}{51}$. In the case of our sample problems, the lower bounds thus defined are 98.31, 127.83 and 125.15. None of these lower bounds are terribly close to the values that the round-robin algorithm finds, but for such small graphs they are unlikely to be tight lower bounds.

We can also state loose upper bounds on the worst possible partition that the round-robin algorithm can find if we again make assumptions like those above. There are limits on how large a vertex group can be before the algorithm must inevitably split it into smaller components, and limits on how small a pair of groups can be without merging. Such bounds are no more likely to be tight, and are just as likely to involve the solution to a quartic equation, so we forego any attempt at such analyses here. Tighter bounds for grid graphs would be quite interesting.

9 Conclusions and future work

The round-robin algorithm provides a framework for the easy parallelization of many of the graph partitioning heuristics of the last two decades. Experiments with other heuristics and with our heuristic for sparse quotient graphs are planned. From our experiments so far, however, we can draw some conclusions. First, the round-robin algorithm is not particularly efficient in terms of reducing the amount of information sent from node to node. However, the messages it sends are large; algorithms that attempt to reduce communication volume without also reducing the number of messages suffer communication delays because, at least on the iPSC hypercube, the message buffer size is over a thousand bytes and the performance cost of a small message is as large as that of a big one. We feel that the large amount of communication in every round of the round-robin algorithm is not a fatal weakness of the algorithm because of the small number of messages it requires.

The large number of rounds, many of them useless, is a concern however. We do not expect to derive a tight theoretical bound on the number of rounds the algorithm will take in the worst case for a particular input or a particular 2-way graph partitioning procedure. A more promising approach is the possibility of detecting when several consecutive rounds have failed to reduce the partition cost significantly. After a number of insignificant improvements, the algorithm could terminate with a fairly good solution. More intriguing is the possibility of some kind of intelligent rerouting of vertex sets. In the later rounds of the algorithm, few rounds result in any improvement. If we could identify the vertex sets most likely to exchange vertices with others and make sure that those vertex sets met frequently, many of the useless later rounds could be eliminated.

Finally, it is interesting to consider the effect of inserting different 2-way partitioning procedures into the round-robin framework to see what properties the resulting partitions have. For example, Leiserson and Lewis [LL88] discuss a graph partitioning strategy based on the F-M algorithm for finding smallest wide vertex separators. Their algorithm, plugged into the round-robin framework, might tend to produce sparse quotient graphs, or partitions with some other interesting property.

Acknowledgements

I thank Earl Zmijewski and John Gilbert for carefully reading an earlier draft of this paper. Their suggestions have improved its organization and clarity. I also thank Mike Heath and Esmond Ng for providing the Oak Ridge National Laboratory hypercube simulator and Cornell University's Theory Center for allowing me to use their Intel iPSC hypercube.

References

[D86] T. H. Dunigan. *A message-passing multiprocessor simulator.* Technical Report ORNL/TM-9966, Oak Ridge National Laboratory, 1986.

[E79] G. C. Everstine. A comparison of three resequencing algorithms for the reduction of matrix profile and wave front. *Int. J. for Numerical Methods in Engineering*, 14, 1979, pp. 837-853.

[FM82] C. M. Fiduccia and R. M. Mattheyses. A linear-time heuristic for improving network partitions. *Proc. 19th Design Automation Conference*, 1982, pp. 175-181.

[GB83] Mark K. Goldberg and Michael Burstein. Heuristic improvement technique for bisection of VLSI networks. *Proc. IEEE International Conf. on Computer Design*, Port Chester, New York, 1983, pp. 122-125.

[GZ87] John R. Gilbert and Earl Zmijewski. *A parallel graph partitioning algorithm for a message-passing multiprocessor.* Technical Report 87-803, Cornell University, 1987.

[JAMS88] David S. Johnson, Cecilia R. Aragon, Lyle A. McGeoch, and Catherine Schevon. Optimization by simulated annealing: an experimental evaluation (part I). Submitted to *Operations Research.*

[KL70] B. W. Kernighan and S. Lin. An efficient heuristic procedure for partitioning graphs. *Bell System Tech. J.* 49 (Feb., 1970), pp. 291-307.

[K84] Balakrishnan Krishnamurthy. An improved min-cut algorithm for partitioning VLSI networks. *IEEE Transactions on computers* C-33, 5 (May, 1984), pp. 438-446.

[LL88] Charles E. Leiserson and John G. Lewis. *Orderings for parallel sparse symmetric factorization.*

[Z88] Earl Zmijewski. A parallel graph partitioning algorithm for a message-passing multiprocessor. To appear in *International Journal of Parallel Computing.*