Seminar on

A Coarse-Grain Parallel Formulation of Multilevel k-way Graph Partitioning Algorithm

Mohammad Iftakher Uddin & Mohammad Mahfuzur Rahman
Matrikel Nr: 9003357 Matrikel Nr : 9003358

Masters of Computer Science
Summer Semester 2002

Fachhochschule Bonn-Rhein-Sieg

Contents

- Abstract
- Introduction
- Multilevel k-way Graph Partitioning
    - Coarsening Phase
    - Partitioning Phase
    - Uncoarsening Phase
- Parallel Multilevel k-way Graph Partitioning
    - Need for Parallel Graph Partitioning
    - Parallel Formulation
    - Computing a Coloring of a Graph
    - Coarsening Phase
        - Coarsening on a shared memory architecture
        - Coarsening on a distributed memory architecture
    - Partitioning Phase
    - Uncoarsening Phase
    - Communication Pattern of the Algorithm
- A Coarse-Grain Parallel Multilevel k-way Graph Partitioning
    - Coarsening Phase
    - Uncoarsening Phase
- Conclusion
- References

1. Abstract

This seminar paper we study and present a Coarse-Grain parallel formulation of a multilevel *k*-way graph partitioning algorithm. The multilevel *k*-way partitioning algorithm reduces the size of the graph by successively collapsing vertices and edges (coarsening phase), finds a *k*-way partitioning of the smaller graph, and then it constructs a *k*-way partitioning for the original graph by projecting and refining the partition to successively finer graphs (uncoarsening phase).

2. Introduction

Graph partitioning is an important problem that has extensive applications in many areas, including scientific computing, VLSI design, geographical information systems, operation research, task scheduling and transportation system. The problem is to partition the vertices of a graph in p roughly equal partitions, such that the number of edges connecting vertices in different partitions is minimized.

A parallel graph partitioning algorithm can take advantage of the significantly higher amount of memory available in parallel computers. In many applications, the graph is already distributed among processors, but needs to be repartitioned due to the dynamic nature of the underlying computation.

Parallel formulation of the multilevel k-way partitioning scheme is even harder, as the refinement of the k-way partitioning appears to require global interactions.

A key feature of this parallel formulation is that it is able to achieve high degree of concurrency while maintaining the high quality of the partitions produced by the serial multilevel partitioning algorithm. Parallel formulation of the coarsening phase is generally applicable to any multilevel graph partitioning algorithm that does coarsening of the graph, and the parallel formulation of the k-way partitioning refinement algorithm can also be used in conjunction with any other parallel graph partitioning algorithm that requires refinement of a k-way partitioning.

3. Multilevel *k-way* Graph Partitioning

The ***k-way*** graph partitioning problem is defined as follows: Given a graph $G = (V, E)$ with $|V| = n$, partition V into k subsets, $V_1, V_2, \ldots\ldots, V_k$ such that $V_i \cap V_j = \emptyset$; for $i \neq j$, $|V_i| = n/k$, and $\cup_i V_i = V$, and the number of edges of E whose incident vertices belong to different subsets is minimized. A k-way partitioning of V is commonly represented by a partitioning vector P of length n, such that for every vertex $v \in V$, $P[v]$ is an integer between 1 and k, indicating the partition to which vertex v belongs. Given a partitioning P, the number of edges whose incident vertices belong to different partitions is called the ***edge-cut*** of the partitioning.

The basic structure of a multilevel k-way partitioning algorithm is very simple. The graph $G = (V, E)$ is first coarsened down to a small number of vertices, a k-way partitioning of this much smaller graph is computed and then this partitioning is projected back towards the original graph (finer graph), by successively refining the partitioning at each intermediate level.

Consider a weighted graph $G_0 = (V_0, E_0)$, with weights both on vertices and edges. A multilevel k-way partitioning algorithm works as follows:

Coarsening Phase The graph G_0 is transformed into a sequence of smaller graphs $G_1, G_2, \ldots\ldots\ldots, G_m$ such that $|V_0| > |V_1| > |V_2| > \ldots.. > |V_m|$.

Partitioning Phase A k-way partitioning P_m of the graph $G_m = (V_m, E_m)$ is computed that partitions V_m into k partitions, each containing roughly $|V_0|/k$ vertices of G_0.

Uncoarsening Phase The partitioning P_m of G_m is projected back to G_0 by going through intermediate partitioning $P_{m-1}, P_{m-2}, \ldots\ldots, P_1, P_0$.

This three stages coarsening, initial partitioning, and refinement is graphically illustrated in Figure 1.

Next we describe each of these phases in more detail.

3.1. Coarsening Phase

During the coarsening phase, a sequence of smaller graphs $G_i = (V_i , E_i)$, is constructed from the original graph $G_0 = (V_0 , E_0)$ such that $|V_i| > |V_{i+1}|$. Graph G_{i+1} is constructed from G_i by finding a maximal matching $M_i \subseteq E_i$ of G_i and collapsing together the vertices that are incident on each edge of the matching. In this process no more than two vertices are collapsed together because a matching of a graph is a set of edges, no two of which are incident on the same vertex. Vertices that are not incident on any edge of the matching are simply copied over to G_{i+1}.

When vertices $v, u \in V_i$ are collapsed to form vertex $w \in V_{i+1}$, the weight of vertex w is set equal to the sum of the weights of vertices v and u, and the edges incident on w is set equal to the union of the edges incident on v and u minus the edge (v, u). For each pair of edges (x, v) and (x, u) (*i.e.*, x is adjacent to both v and u) a single edge (x, w) is created whose weight is set equal to the sum of the weights of these two edges. Thus, during successive coarsening levels, the weight of both vertices and edges increases.

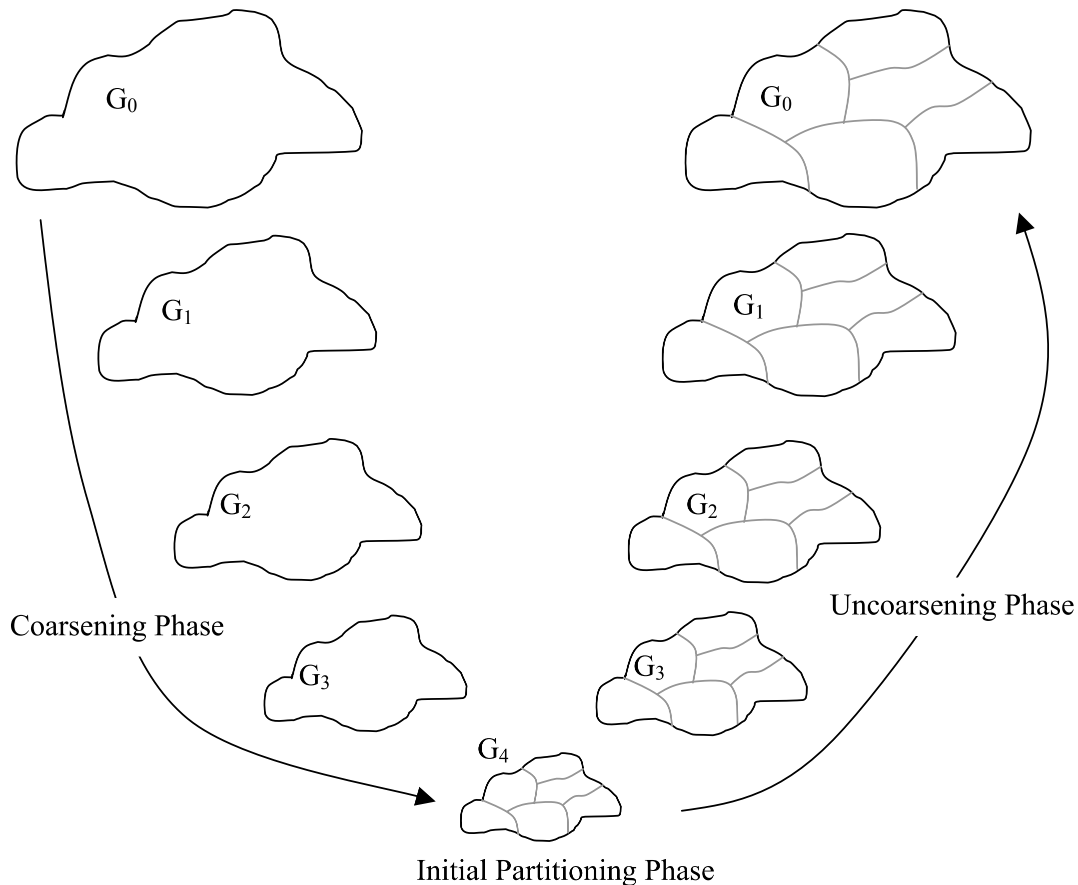

Figure 1: The various phases of the multilevel *k*-way partitioning algorithm. During the coarsening phase, the size of the graph is successively decreased; during the initial partitioning phase, a *k*-way partitioning of the smaller graph is computed (a 6-way partitioning in this example); and during the uncoarsening phase, the partitioning is successively refined as it is projected to the larger graphs.

The process of coarsening is illustrated in Figure 2. Each vertex and edge in Figure 2(a) has a unit weight. Figure 2(b) shows the coarsened graph that results from the contraction of shaded vertices in Figure 2(a). Numbers on the vertices and edges in Figure 2(b) show their resulting weights.

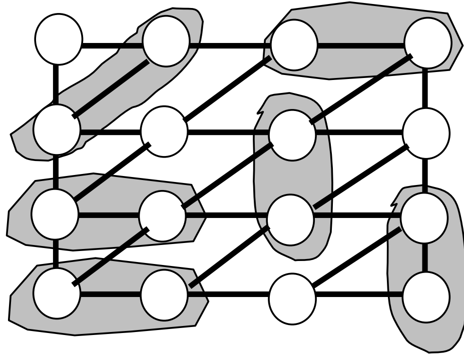

Figure 2(a)

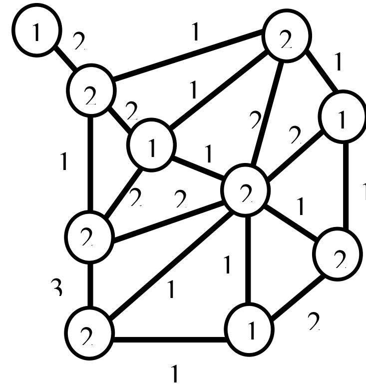

Figure 2 (b)

Figure 2: Process of coarsening

Maximal matchings can be computed in different ways. The method used to compute the matching greatly affects both the quality of the partition, and the time required during the uncoarsening phase. The *heavy-edge matching* (HEM) matching scheme is follwed here.It computes a matching M_i , such that the weight of the edges in M_i is high. The heavy-edge matching is computed using a randomized algorithm as follows. The vertices are visited in a random order. However, instead of randomly matching a vertex with one of its adjacent unmatched vertices, HEM matches it with the unmatched vertex that is connected with the heavier edge. As a result, the HEM scheme quickly reduces the sum of the weights of the edges in the coarser. The coarsening phase ends when the coarsest graph G_m has a small number of vertices.

3.2. Partitioning Phase

The second phase of a multilevel *k*-way partition algorithm is to compute a *k*-way partition of the coarse graph $G_m = (V_m, E_m)$ such that each partition contains roughly $|V_0| / k$ vertex weight of the original graph. Since during coarsening, the weights of the vertices and edges of the coarser graph were set to reflect the weights of the vertices and edges of the finer graph, G_m contains sufficient information to intelligently enforce the balanced partition and the minimum edge-cut requirements. The *k*-way partition of G_m is computed using multilevel recursive bisection algorithm.

3.3. Uncoarsening Phase

During the uncoarsening phase, the partitioning of the coarser graph G_m is projected back to the original graph by going through the graphs G_{m-1}, G_{m-2}, ……….., G_1. Since each vertex $u \in V_{i+1}$ contains a distinct subset U of vertices of V_i, the projection of the partition from G_{i+1} to G_i is constructed by simply assigning the vertices in U to the same partition in G_i that vertex u belongs in G_{i+1}.

Even though the partition of G_{i+1} is at a local minima, the projected partition of G_i may not. Since G_i is finer, it has more degrees of freedom that can be used to improve the partition and thus decrease the edge-cut. The basic purpose of a partition refinement algorithm is to select vertices such that when moved from one partition to another the resulting partition has smaller edge-cut and remains balanced *(i.*e., each partition has the same weight).
The multilevel k-way partitioning algorithm is based on a simplified version of the Kernighan-Lin algorithm, extended to provide k-way partition refinement. This algorithm is called *greedy refinement* (GR). Its complexity is largely independent of the number of partitions being refined. The GR algorithm consists of a number of iterations, and in each iteration all the vertices are checked in a random order to see if they can be moved. Let v be such a vertex. If v is a boundary vertex *(i.*e., it is connected with a vertex that belongs to an other partition), then v is moved to the partition that leads to the largest reduction in the edge-cut, subject to partition weight constraints. These weight constraints ensure that all partitions have roughly the same weight. If the movement of v cannot achieve any reduction in the edge-cut, it is then moved to the partition (if any) that improves the partition-weight balance but leads to no increase in the edge-cut. The GR algorithm converges after a small number of iterations (within four to eight iterations). If the GR algorithm is not able to enforce the partition balance constraints, an explicit balancing phase is used that moves vertices between partitions even if this movement leads to an increase in the edge-cut.

4. Parallel Multilevel k-way Graph Partitioning

The key feature of parallel formulation is that it is able to achieve high degree of concurrency while maintaining the high quality of the partitions produced by the serial multilevel partitioning algorithm. Parallel formulation of the coarsening phase is generally applicable to any multilevel graph partitioning algorithm that does coarsening of the graph, and the parallel formulation of the k-way partitioning refinement algorithm can also be used in conjunction with any other parallel graph partitioning algorithm that requires refinement of a k-way partitioning.

4.1. Need for Parallel Graph Partitioning

Even though the multilevel partitioning algorithms produce high quality partitions in a very small amount of time, the ability to perform partitioning in parallel is important for many reasons. The amount of memory on serial computers is not enough to allow the partitioning of graphs corresponding to large problems that can now be solved on massively parallel computers and workstation clusters. By performing graph partitioning in parallel, the algorithm can take advantage of the significantly higher amount of memory available in parallel computers.

4.2. Parallel Formulation

Developing a highly parallel formulation for the multilevel k-way partitioning algorithm is particularly difficult because both the task of computing a maximal matching during the coarsening phase, and the task of refining the partition during the uncoarsening phase appear to be quite serial in nature.

Out of the three phases of the multilevel k-way partitioning algorithm, the coarsening and the uncoarsening phases require the bulk of the computation (over 95%). Hence, it is critical for any efficient parallel formulation of the multilevel k-way partitioning algorithm to successfully parallelize these two phases. Recall that during the coarsening phase, a matching of the edges is computed, and it is used to contract the graph. One possible way of computing the matching in parallel is to have each processor only compute matching between the vertices that it stores locally, and use these local matching to contract the graph. Since each pair of matched vertices resides on the same processor, this approach requires no communication during the contraction step. This approach works well as long as each processor stores relatively well connected portions of the entire graph. In particular, if the graph was distributed among the processors in a partitioned fashioned, then this approach would have worked extremely well. This is not a realistic assumption in most cases, since finding a good partition of the graph is the problem we are trying to solve by the multilevel k-way partitioner. Nevertheless, this approach of *local matching* can work reasonably well when the number of processors used is small relative to the size of the graph and the average degree of the graph is relatively high. The reason is that even a random partition of a graph among a small number of processors can leave many connected components at each processor. An alternate approach is to allow vertices belonging to different processors to be matched together. Compared to local matching schemes, this type

of matching provides matching of better quality, and its ability to contract the graph does not depend on the number of processors, or the existence of a good pre-partition. However, this *global matching* significantly complicates the parallel formulation because it requires a distributed matching algorithm. For example, if vertices v and u are located in two different processors P_1 and P_2, then on P_1 vertex v might be matched to u, while on P_2 vertex u may be matched to a different vertex w. Furthermore, another processor P_3 may match its vertex z to vertex u as well. Any correct and usable distributed matching algorithm must resolve both of these conflicts efficiently. Note that since pairs of vertices that are contracted together can reside on different processors, a global communication is required when the contracted graph is constructed.

During the uncoarsening phase, the k-way partition is iteratively refined as it is projected to successively finer graphs. The serial algorithm scans the vertices and moves any vertices that lead to a reduction in the edge-cut. Any parallel formulation of this algorithm will need to move a group of vertices at a time in order to speedup the refinement process. This group of vertices needs to be carefully selected so that every vertex in the group contributes to the reduction in the edge-cut. For example, it is possible that processor P_i decides to move a set of vertices S_i to processor P_j to reduce the edge-cut because the vertices in S_i are connected to a set of vertices T that are located on processor P_j . But, in order for the edge-cut to improve by moving the vertices in S_i, the vertices in T must not move. However, while P_i selects S_i , processor P_j may decide to move some or all the vertices in T to some other processor. Consequently, when both sets of vertices are moved by P_i and P_j, the edge-cut may not improve; and it may even get worse. Clearly, the group selection algorithm must eliminate this type of unnecessary vertex movements.

The developed highly parallel formulations for all three phases of the multilevel k-way graph partitioning algorithm is described on the following. This formulation utilizes graph coloring to eliminate conflicts in the computation of global matching in the coarsening phase and to eliminate unnecessary vertex movement in the parallel variation of the Kernighan-Lin refinement in the uncoarsening phases.

Let p be the number of processors used to compute a p-way partition of the graph $G = (V,E)$. G is initially distributed among the processors using a one-dimensional distribution, so that each processor receives n/p vertices and their adjacency lists. At the end of the algorithm, a partition number is assigned to each vertex of the graph.

4.3. Computing a Coloring of a Graph

A coloring of a graph $G = (V, E)$ assigns colors to the vertices of G so that adjacent vertices have different color. So it requires to find a coloring such that the number of distinct colors used is small. This parallel graph coloring algorithm consists of a number of iterations. In each iteration, a maximal independent set of vertices I is selected using a variation of Luby's algorithm. All vertices in this independent set are assigned the same color. Before the next iteration begins, the vertices in I are removed from the graph, and this smaller graph becomes the input graph for the next iteration. A maximal independent set I of a set of vertices S is computed in an incremental fashion using this algorithm. A random number is assigned to each vertex, and if a vertex has a random number that is smaller than all of the random numbers of the adjacent vertices, it is then included in I. Now this process is repeated for the vertices in S that are neither in I nor adjacent to vertices in I, and I is

augmented similarly. This incremental augmentation of I ends when no more vertices can be inserted in I.

Luby's algorithm can be implemented quite efficiently on a shared memory parallel computer, since for each vertex v, a processor can easily determine if the random value assigned to v is the smaller among all the random values assigned to the adjacent vertices. However, on a distributed memory parallel computer, for each vertex, random values associated with adjacent vertices that are not stored on the same processor needs to be explicitly communicated. In this implementation of Luby's algorithm, prior to performing the coloring in parallel, a *communication setup* phase is performed, in which appropriate data structures are created to facilitate this exchange of random numbers. In particular, it is predetermined which vertices are located on a processor boundary *(i.e.*, a vertex connected with vertices residing on different processors), and which are internal vertices *(i.e.*, vertices that are connected only to vertices on the same processors). These data structures are used in all the phases of this parallel multilevel graph partitioning algorithm.

4.4. Coarsening Phase

During the coarsening phase a sequence G_1, G_2, ……, G_m of successively smaller graphs is constructed. Graph G_{i+1} is derived from G_i by finding a maximal matching M_i of G_i and then collapsing the vertices incident on the edges of M_i . A matching algorithm that is based on the coloring of the graph is used to coarse the graph. This coloring algorithm also happens to be essential for parallelizing the partitioning refinement phase.

This parallel matching algorithm is based on an extension of the serial algorithm and utilizes graph coloring to structure the sequence of computations. Consider the graph $G_i = (V_i, E_i)$ that has been colored using this parallel formulation of Luby's algorithm, and let *Match* be a variable associated with each vertex of the graph, that is initially set to -1. At the end of the computation, the variable *Match* for each vertex v stores the vertex that v is matched to. If v is not matched, then *Match* = v.

4.4.1. Coarsening on a shared memory architecture

The matching M_i is constructed in an iterative fashion. During the cth iteration, vertices of color c that have not been matched yet (*i.e.*, *Match = -1*) select one of their unmatched neighbors using the heavy-edge heuristic, and modify the *Match* variable of the selected vertex by setting it to their vertex number. Let v be a vertex of color c and (v, u) be the edge that is selected by v. Since the color of u is not c, this vertex will not be selecting a partner vertex at this iteration. However, there is a possibility that another vertex w of color c may select (w, u). Since both vertices v and w perform their selections at the same time, there is no way of preventing that. This is handled as follows. After all vertices of color c select an unmatched neighbour, they synchronize. The vertices of color c that have just selected a neighbour, read the *Match* variable of their selected vertex. If the value read is equal to their vertex number, then their matching was successful, and they set their *Match* variable equal to the selected vertex; otherwise the matching fails, and the vertex remains unmatched. Note that if more than one vertex (*e.g.*, v and w) want to match with the same vertex (*e.g.*, u), only one of the writes in the *Match* variable of the selected vertex will

succeed; and this determines which matching survives. However, by using coloring, it is restricted which vertices select partner vertices during each iteration; thus, the number of such conflicts is significantly reduced. Also note that even though a vertex of color *c* may fail to have its matching accepted due to conflicts, this vertex can still be matched during a subsequent iteration corresponding to a different color.

4.4.2. Coarsening on a distributed memory architecture

The above algorithm is implemented quite easily on a distributed memory parallel computer as follows. The *writes* into the *Match* variables are gathered all together and are sent to the corresponding processors in a single message. If a processor receives multiple *write* requests for the same vertex, the one that corresponds to the heavier edge is selected. Any ties are broken arbitrarily. Similarly, the *reads* from the *Match* variables are gathered by the processors that store the corresponding variables and they are sent in a single message to the requesting processors. Furthermore, during this *read* operation, the processors who own the *Match* variables also determine if they will be the ones storing the collapsed vertex in G_{i+1}.
After a matching M_i is computed, each processor knows how many vertices (and the associated adjacency lists) it needs to send and how many it needs to receive. Each processor then sends and receives these sub-graphs, and it forms the next level coarser graph by merging the adjacency lists of the matched vertices.

4.5. Partitioning Phase

During the partitioning phase, a *p*-way partition of the graph is computed using a recursive bisection algorithm. In this algorithm this phase is parallelized by using a parallel algorithm that parallelizes the recursive nature of the algorithm. This is done as follows: The various pieces of the coarse graph are gathered to all the processors using an all-to-all broadcast operation. At this point the processors perform recursive bisection using an algorithm that is based on nested dissection and greedy partition refinement. However, as illustrated in the following figure, each processor explores only a single path of the recursive bisection tree. At the end each processor stores the vertices that correspond to its partition of the *p*-way partition. Note that after the initial all-to-all broadcast operation, the algorithm proceeds without any further communication.

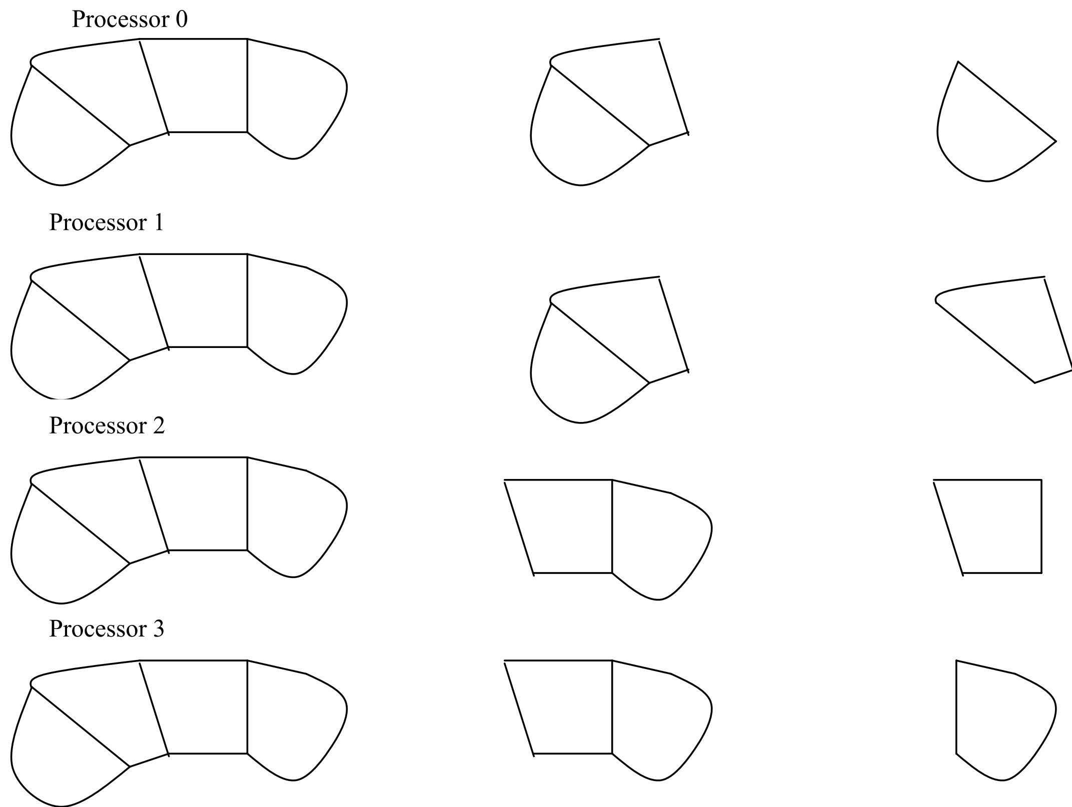

Figure 3: Performing the initial k-way partitioning in parallel. Each processor explores only a single path from the root to the leaves in the recursive bisection tree.

4.6. Uncoarsening Phase

In the uncoarsening phase, the partition is projected from the coarse graph to the next level finer graph, and it is refined using the greedy refinement algorithm. Recall that during a single phase of the refinement algorithm the vertices are randomly traversed, and the vertices that lead to a decrease in the edge cut switch partitions. After each such vertex movement, the external degrees of the adjacent vertices are updated to reflect the new partition.

In the parallel formulation of greedy refinement, the spirit of the serial algorithm is retained, but the order in which the vertices are traversed to determine if they can be moved to different partitions is changed. In particular, the single phase of the refinement algorithm is broken up into c sub-phases, where c is the number of colors of the graph to be refined. During the ith phase, all the vertices of color i are considered for movement, and the subset of these vertices that lead to a reduction in the edge-cut (or improve the balance without increasing the edge-cut) are moved. Since, the vertices with the same color form an independent set, the total reduction in the edge-cut achieved by moving all vertices at the same time is equal to the sum of the edge-cut reductions achieved by moving these vertices

one after the other. After performing this *group* movement, the external degrees of the vertices adjacent to this group are updated, and the next color is considered.

During the parallel refinement, the vertices can be moved physically as they change partitions. That is, each processor initially stores all the vertices of a single part, and as vertices move between partitions during refinement, they can also move between the corresponding processors. However, in the context of multilevel graph partitioning such an approach requires significant communication. This is because for each vertex v in the coarse graph G_i that will be moved it needs to send not only the adjacency list of v but also the adjacency lists of all the vertices collapsed in v for the higher level finer graphs G_{i-1}, G_{i-2} ….. , G_0.

In this parallel refinement algorithm this problem is solved as follows. Vertices do not move from processor to processor, but only the partition number associated with each vertex changes. Since the vertices are initially distributed in a random order, each processor stores vertices that belong to almost all p partitions. This ensures that during refinement each processor will have some boundary vertices that needs to be moved, leading to a generally load balanced computation. Furthermore, this also leads to a simpler implementation of the parallel refinement algorithm, since vertices (and their adjacency lists) do not have to be moved around. Of course, all the vertices are moved to their proper location at the end of the partitioning algorithm, using a single all-to-all personalized communication .

The balance conditions are maintained as follows. Initially, each processor knows the weights of all p partitions. During each refinement sub-phase, each processor enforces balance constraints based on these partition weights. For every vertex it decides to move, it locally updates these weights. At the end of each sub-phase, the global partition weights are recomputed, so that each processor knows the exact weights.

4.7. Communication Pattern of the Algorithm

The parallel formulation of the multilevel k-way partitioning algorithm is made of five different parallel algorithms, namely coloring, matching, contraction, initial partitioning, and refinement. Out of these five algorithms, three of them (coloring, matching, and refinement) have similar communication requirements. The amount of communication performed by each one of these three algorithms depends on the number of interface vertices. For example, during coloring, each processor needs to know the random numbers of the vertices adjacent to the locally stored vertices. Similarly, during refinement, every time a vertex is moved, the adjacent vertices need to be notified to update their partitioning information. Initially, each processor stores n/p vertices and nd/p edges, where d is the average degree of the graph. Thus, the number of interface vertices is at most $O(n/p)$. Since the vertices are initially distributed randomly, these interface vertices are equally distributed among the p processors. Hence, each processor needs to exchange data with $O(n/p^2)$ vertices of each processor. Alternatively, each processor needs to send information for about $O(n/p^2)$ locally stored vertices to each other processor. This can be accomplished by using the all-to-all personalized communication operation. As the size of the coarser graphs successively decreases, the amount of data that needs to be exchanged also decreases. However, each processor still needs to send and receive data from almost all

other processors. If t_s is the message startup overhead, then each of these all-to-all personalized communication operations requires pt_s time just due to startup overhead.

The number of all-to-all personalized operations performed for each coarse graph G_i depends on the number of colors c of G_i . In particular, it performs c operations during coloring (one for each color that it computes), $2c$ during matching (two for each color), and $4c$ during refinement (it performs two passes of the refinement algorithm, each consisting of c sub-phases and it needs to communicate twice during each sub-phase). Thus, for each graph G_i, it performs $7c$ all-to-all personalized operations.

Consider now a graph with a million vertices, that is partitioned on 128 processors, and that the coarsest graph consists of about one thousand vertices. This level of contraction can be achieved by going through about ten coarsening levels. Also, assume that the average number of colors of each coarse graph is around ten. Given these parameters, the parallel multilevel k-way partitioning algorithm will perform a total of 700 all-to-all personalized communication operations. On 128 processors, these operations will incur a total of $89600t_s$ overhead due to message startup time.

5. A Coarse-Grain Parallel Multilevel *k*-way Graph Partitioning

Because of the high message startup overhead it is required to modify the parallel multilevel *k*-way partitioning algorithm presented so that it performs fewer all-to-all personalized communication operations. From the discussion, it is clear that one way of doing this is to develop matching and refinement algorithms that do not depend on the number of colors of the graph.

5.1. Coarsening Phase

In fact, the motivation in using a coloring-based matching algorithm was to minimize the number of conflicts; thus, a non-coloring based algorithm can be used at the expense of a higher number of matching conflicts. The new matching algorithms consists of a number of phases. During phase i, each processor scans its local unmatched vertices. For each such vertex v, it matches it with another unmatched vertex u (if such a vertex exists) using the heavy-edge heuristic. If u is stored locally, then the matching is granted right away, otherwise a matching request is issued to the processor that stores u, depending on the ordering of v and u. In particular, if i is odd, a match request is issued only if $v< u$, whereas if i is even, a match request is issued only if $v>u$. This ordering is done to ensure that conflicts can be resolved with a single communication step. Next, every processor processes the matching requests that it received, grants some of these requests by breaking conflicts arbitrarily, and notifies the corresponding processors. The matching algorithm terminates when a large fraction of the vertices has been matched. This experiments show that for most graphs, very large matchings can be obtained with only four phases. Thus, the number of all-to-all personalized communications are reduced from two times the number of colors to only eight.

5.2. Uncoarsening Phase

However, performing refinement without using coloring is somewhat more difficult. Recall that by moving a group of vertices of a single color at a time, we were able to ensure that no thrashing occurs during refinement. For example, consider the situation illustrated in the following figure 4(a), in which two vertices *v* and *u* are connected via an edge and belong to partitions *i* and *j* , respectively. Note that if we move vertex *v* to partition *j* we reduce the edge-cut by two, and if we move vertex *u* to partition *i* we reduce the edge-cut by five. However, as illustrated in figure 4(b), if we move both vertex *v* to partition *j* and vertex *u* to partition *i* , then the edge-cut actually increases by five.

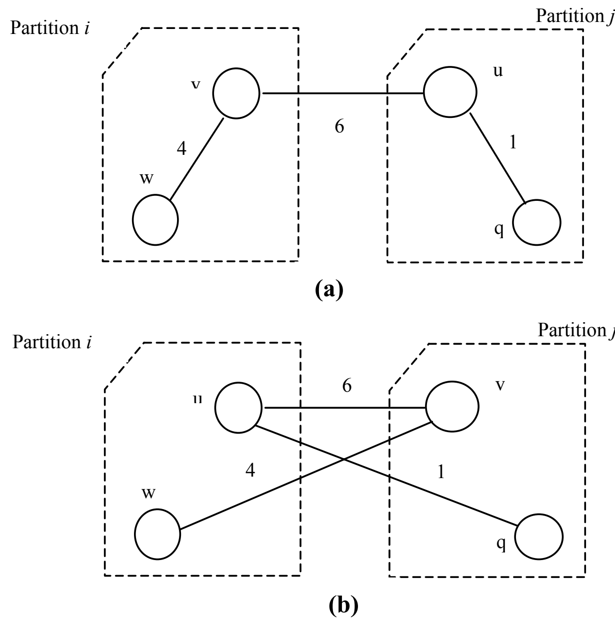

Figure 4: two vertices *v* and *u* belonging to partitions *i* and *j*, respectively are moving partitions *j* and *i* respectively increasing edge-cut by five.

The coloring-based refinement algorithm is able to prevent such moves since it only allows concurrent movement of vertices that are not connected *(i.e.*, independent). However, by looking closer at this example we see that an alternate way of preventing such type of movements is to devise a refinement algorithm that does not concurrently move vertices between the same partitions. That is, during each refinement step, for any pair of partitions *i* and *j*, it should only move vertices in one direction, *i.e.*, it should move vertices only from partition *i* (*j*) to partition *j* (*i*). In particular, each phase of the new refinement algorithm consists of only two sub-phases. In the first sub-phase, the group of vertices to be moved is

selected so that vertices move from lower- to higher-numbered partitions, and during the second sub-phase, vertices move in the opposite direction. Thus, the new refinement algorithm reduces the number of all-to-all personalized communication operations that are required in each refinement phase from two times the number of colors to four.

Note that this new refinement scheme allows vertices that are connected and belong to the same partition to be moved concurrently. For example, consider the example illustrated in the following figure 5(a),

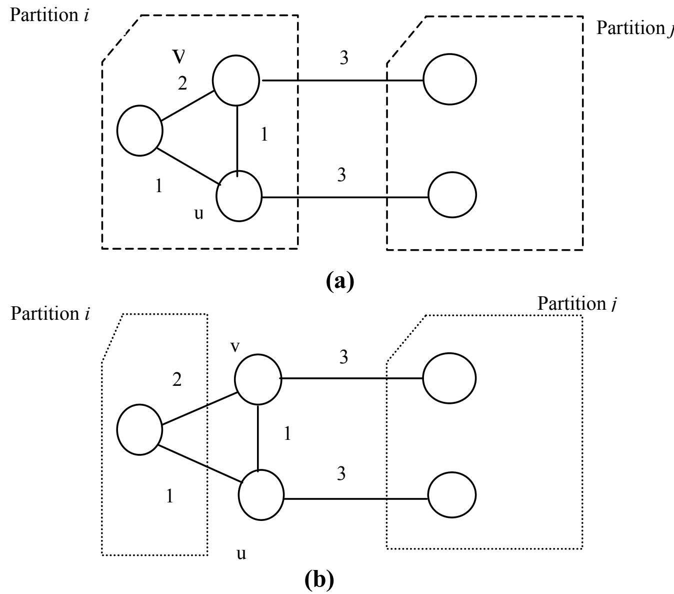

Figure 5: two vertices *v* and *u* belonging to partitions *i* moving concurrently to partitions *j* are increasing edge-cut at least as high as the sum of the edge-cut reductions of each individual move.

in which vertices *v* and *u*, both of them belonging to partition *i*, are moved concurrently to partition *j*, since each such move individually leads to a reduction in the edge-cut. However, this type of moves will never lead to thrashing. In fact, the reduction in the edge-cut obtained by concurrently moving connected vertices from the same partition, is at least as high as the sum of the edge-cut reductions of each individual move. This is illustrated by the example in figure 5(b). Thus, the coloring-based refinement algorithm was in essence too restrictive while selecting vertices for movement.

By using the new refinement scheme, there are certain type of moves that may potentially lead to thrashing. Consider the example shown in the following figure 6(a), in which vertex *u* is connected to vertices *v* and *w* each belonging to a different partition. If vertex *v* is moved to partition *j* the edge-cut reduces by one, and if vertex *u* moves to partition *k* the edge-cut reduces by one. However, as illustrated in Figure 6(b), if both moves take place concurrently, then the edge-cut actually increases by one. Fortunately, there are not many vertices that can lead to this type of movement. This is because, this type of moves can only

happen among a sequence of vertices that are connected via path and they are interface vertices to multiple domains.

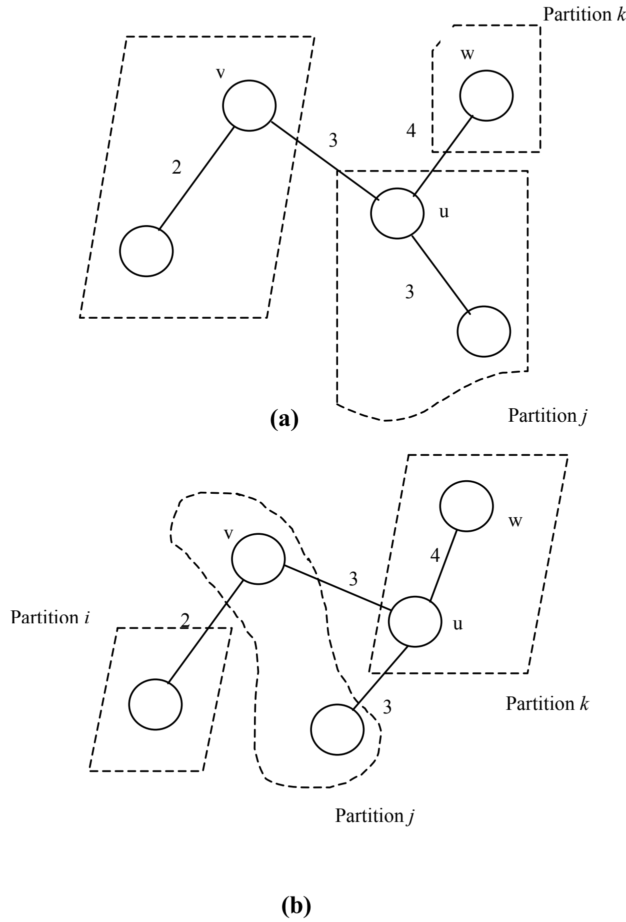

Figure 6: Vertices *v, u and w* belonging to partitions *i, j and k* respectively where *v* and *u* are moving concurrently to partitions *j* and *k* respectively increases edge-cut by one.

The number of all-to-all personalized communication operations required for each graph G_i by the new matching and refinement algorithms is now only 16 where eight for matching and eight for refinement, assuming that it performs two passes of the refinement algorithm.

The implementation of this coarse-grain algorithm is memory efficient. In particular, each processor requires memory proportional to the size of the locally stored portion of the graph, *i.e.*, $O(n/p)$, where *n* is the number of vertices in the graph.

6. Conclusion

The coarse-grained parallel multilevel k-way partitioning algorithm has a number of enhancements over the parallel coloring based algorithm that both improve its performance as well as extend its functionality.

As the size of the successively coarser graphs decreases, the amount of time required to generate the next level coarser graphs is dominated by the communication overheads. This is because, the graphs become too small and the message startup overheads dominate the communication time. At this point, the overall amount of time required to generate the remaining coarse graphs as well as the amount of time spent in refining them, will decrease if the work associated with that is assigned to fewer processors. The coarse-grain parallel algorithm performs such type of graph folding. In particular, as the coarsening progresses, the size of the coarse graph is monitored, and if it falls bellow a certain threshold, it is then folded to only half the processors. Now these processors perform any subsequent coarsening (and refinement during the uncoarsening phase). This folding of the graph to fewer processors is repeated again if necessary. These experiments have shown that this successive folding of the graphs to fewer processors improves the overall run-time of the partitioning algorithm. The size of the graph after which folding is triggered depends on the characteristics of the underlying interconnection network. If the message startup overhead is very small, then smaller graphs will trigger a folding, whereas, if the message startup time is high, a larger graph will be required.

References

[1] G. Karypis and V. Kumar. A Coarse-Grain Parallel Formulation of Multilevel k-way Graph Partitioning Algorithm, University of Minnesota, Department of Computer Science.

[2] G. Karypis and V. Kumar. A fast and high quality multilevel scheme for partitioning irregular graphs. Technical Report TR 95-035, Department of Computer Science, University of Minnesota, 1995. Also available on WWW at URL http://www.cs.umn.edu/˜karypis/papers/mlevel serial.ps. A short version appears in Intl. Conf. on Parallel Processing 1995.

[3] G. Karypis and V. Kumar. Multilevel k-way partitioning scheme for irregular graphs. Technical Report TR 95-064, Department of Computer Science, University of Minnesota, 1995. Also available on WWWat URL http://www.cs.umn.edu/˜karypis/papers/mlevel kway.ps.

[4] G. Karypis and V. Kumar. Parallel multilevel k-way partitioning scheme for irregular graphs. Technical Report TR 96-036, Department of Computer Science, University of Minnesota, 1996. Also available on WWW at URL http://www.cs.umn.edu/˜karypis/papers/mlevel kparallel.ps. A short version appears in Supercomputing 96.