Multi-Threaded Graph Partitioning

Dominique LaSalle and George Karypis
Department of Computer Science & Engineering
University of Minnesota
Minneapolis, Minnesota 55455, USA
{*lasalle,karypis*}*@cs.umn.edu*

***Abstract*—In this paper we explore the design space of creating a multi-threaded graph partitioner. We present and compare multiple approaches for parallelizing each of the three phases of multilevel graph partitioning: coarsening, initial partitioning, and uncoarsening. We also explore the differences in thread lifetimes and data ownership in this context. We show that despite the options for fine-grain synchronization and task decomposition offered by current threading technologies, the best performance is achieved by preserving data ownership and minimizing synchronization. In addition to this we also present an unprotected approach to generating a vertex matching in parallel with little overhead. We use these findings to develop an OpenMP based implementation[1] of the *Metis* algorithms and compare it against MPI based partitioners on three different multi-core architectures. Our multi-threaded implementation not only achieves greater than a factor of two speedup over the other partitioners, but also uses significantly less memory.**

I. Introduction

Graphs are used in a large number of areas in computing including social networks, biological networks, scientific computing and distributed computing, and VLSI design. The graph partitioning problem is to divide the vertices of a graph into groups of roughly equal size, such that the number of edges connecting vertices of different groups is minimized. Graph partitioning is used as a first step in *divide & conquer* approaches to reduce complexity, task decomposition for parallel systems, and data decomposition for distributed systems. As the capacity to collect information and to model complex systems has soared, so has the size and diversity of the graphs that are being generated and need to be processed and analyzed. This growth in scale and scope presents new challenges to partitioning these graphs quickly and efficiently.

Because the problem of graph partitioning is known to be in NP-Complete [1], directly solving it is not feasible. However, graph partitioning is a well studied problem and as a result, many fast heuristic algorithms exist for finding high-quality partitionings. The multilevel paradigm is a widely used approach in graph partitioning libraries including *Chaco* [2], *Metis* [3], *Jostle* [4], *Party* [5], *Scotch* [6], and *KaFFPa* [7]. In multilevel approaches a series of successively smaller graphs that approximate the original graph are generated before a partitioning of the smallest graph is made, which is then applied to each successively larger graph with some optimization, until it is finally applied to the original graph. Many of these multilevel algorithms also have parallel formulations designed for distributed memory systems [8], [9], [10], [11].

In recent years we have seen everything from the nodes of high-end distributed systems to commodity workstations shift from being single-core/single-processor machines to multi-core/multi-processor shared memory machines. This shift has also caused the amount of available memory per processing element to decrease [12]. Although existing parallel graph partitioning algorithms designed for distributed memory systems can be used as is on these shared memory systems, doing so is sub-optimal. This is because they do not fully take advantage of the shared memory capability of the architectures to reduce the amount of data that needs to be replicated between the processes nor the ability that shared memory allows for fine-grain synchronization.

In this paper we explore the design space of creating a multi-threaded graph partitioner. We present and compare various algorithms for the key steps of the multilevel partitioning paradigm that take different strategies in terms of synchronization frequency, task granularity, data ownership, and thread lifetime. Our experiments, on three different multicore architectures using the threading functionality provided by OpenMP, show that even though multicore architectures allow for fine grain task decomposition and frequent synchronization, the best performance is achieved when the task decomposition is coarse and synchronization is infrequent. In addition to this we found that data locality, which when using OpenMP can only be controlled implicitly by having threads operate on the data they generate, is also crucial to achieving performance on a large number of cores. These findings are to a large extent consistent with the best practices of high performance parallel algorithms used on distributed memory machines. However, we identify a technique for implementing the coarsening phase of the multilevel paradigm that results in a significant speedup beyond the distributed formulation. In addition, without the need for caching remote data locally on multicore architectures, we were able to significantly reduce the aggregate amount of memory required as the number of threads increases

[1]The *mt-metis* software is available at http://www.cs.umn.edu/~metis

compared to distributed memory formulations.

The remainder of this paper is organized into five sections. Section II provides a definition of the graph partitioning problem and the associated notations used in this paper. Section III gives an overview of multilevel graph partitioning, followed by a detailed look at the algorithms used by *KMetis* and *ParMetis*. In Section IV we discuss the different approaches to parallelization we experimented with while engineering *mt-metis*, a multi-threaded implementation of the *Metis* algorithms. In Section V we detail the conditions of our experiments. The results of those experiments are presented in Section VI, including the comparison of the different algorithmic approaches, threading approaches, and our best implementation against *PT-Scotch* and *ParMetis*. Finally in Section VII we provide a summary of these results and discuss future directions of this work.

II. Definitions & Notation

A *graph* $G = (V, E)$ consists of a set of vertices V and a set of edges E, where each edge $e = \{v, u\}$ contains a pair of vertices (i.e., $v, u \in V$). Each vertex $v \in V$ and each edge $e \in E$ can have a positive weight associated with them that are denoted by $\eta(v)$ and $\theta(e)$, respectively. If there are no weights associated with the vertices and/or edges, then their weights are assumed to be one. Given a vertex $v \in V$, its set of adjacent vertices are denoted by $\Gamma(v)$ and it will be referred to as the *neighborhood* of v.

A partitioning of V into k non-empty and disjoint subsets $V_1, V_2, \ldots, V_k$ is called a *k-way partitioning* of G, and each set V_i is referred to as a *partition*. The partitioning of V will be represented using a *partitioning vector* P, such that P^v will store the partition number of vertex v. The edges that connect vertices in different partitions are considered to be *cut* by the partitioning and the sum of the weight of the cut edges is called the *edgecut* of a k-way partitioning. The vertices incident on cut edges are referred to as *boundary vertices*.

The *balance* of a k-way partitioning measures how evenly-weighted are the k partitions and is defined as the ratio of $k \max_i(\eta(V_i))/\eta(V)$, where $\eta(A)$ is the sum of the vertex weights of the vertices in set A. A balance close to one indicates that the partitions are evenly weighted, whereas values greater than one, indicate that there are some partitions whose vertex weight is much greater than the average.

Given a user supplied parameter ϵ, we define the *k-way partitioning problem* of G as the optimization problem of computing a k-way partitioning of G with minimum edgecut subject to the constraint that the balance is upper bounded by $1 + \epsilon$. Due to the balance constraint, this partitioning problem is also referred to as the *bounded capacity* graph partitioning problem.

III. Multi-Level Graph Partitioning

Since their introduction over 20 years ago [13], multilevel approaches for graph partitioning have become the standard approach for developing high-quality and computationally efficient approaches for graph partitioning. These algorithms solve the underlying optimization problem using a methodology that follows a *simplify & conquer* approach, initially used by multi-grid methods for solving systems of partial differential equations.

Multilevel partitioning methods consist of three distinct phases: *coarsening*, *initial partitioning*, and *uncoarsening*. In the coarsening phase, the original graph G_0 is used to generate a series of increasingly *coarser* graphs, $G_1, G_2, \ldots, G_m$. In the initial partitioning phase, a partitioning P_m of the much smaller graph G_m is generated using a more expensive algorithm. Finally, in the uncoarsening phase, the initial partitioning is used to derive partitionings of the successive finer graphs. This is done by first *projecting* the partition of G_{i+1} to G_i, followed by partitioning refinement whose goal is to reduce the edgecut by moving vertices among the partitions. Since the successive finer graphs contain more degrees of freedom, such refinement is often feasible and leads to dramatic edgecut reductions.

The overall effectiveness of the multilevel paradigm depends on the approaches used to identify the set of all vertices that will be contracted during the coarsening phase and the partitioning refinement during the uncoarsening phase. Over the years various approaches have been developed and extensively evaluated [14]. For example, coarsening is usually performed by computing a matching [13], [15] or clustering [16] though approaches based on weighted aggregation have also been explored [17], [18]. The refinement is often performed using local search methods based on either on the Kernighan-Lin [19], Fiduccia-Mattheyses [20], or Greedy [15] refinement algorithms. More recently a second method of refinement has been used in [21] and [7], in which a modified version of *max-flow min-cut* algorithm is used.

Because the methods described in Section IV build on the algorithms in *KMetis* [22] and *ParMetis* [8] in the rest of this section we provide additional details about the data structures and algorithms used in both of these algorithms/codes.

A. KMetis

KMetis stores the graph in a structure similar to that of a Compressed Sparse Row format used for sparse matrices, with the addition of a vector storing the vertex weights, the partition vector P, and the vertex mapping vector C.

The coarsening phase in *KMetis* is made up of two parts: matching and contraction. Matching is where the vector C_i is generated from G_i, and contraction generates G_{i+1} from C_i and G_i. In matching we start with a matching vector M, which is used to generate C. If two vertices, v and u,

are matched together they have corresponding entries in M, $M^v = u$ and $M^u = v$. They are also mapped to the same coarse vertex $c = C_i^v = C_i^u$.

The vertices are visited in ascending order with respect to their degree. Vertices with the same degree are visited in random order, to encourage exploration of the solution space over multiple runs. If a visited vertex v has already been matched, it is skipped. Otherwise the neighbors $\Gamma(v)$ of v are searched for the unmatched neighbor u connected by the heaviest edge and is recorded in the matching vector as both $M^v = u$ and $M^u = v$. If v has no unmatched neighbors, it is matched with itself, $M^v = v$. The ascending order visitation is meant to reduce the occurrence of unmatched vertices, by giving low degree vertices first choice of neighbors with which to match. This coarsening scheme is known as Sorted Heavy Edge Matching or *SHEM*.

In contraction, two matched vertices $v, u \in V_i$ will share the same mapping to the coarse vertex c. The vertex weight of c is equal to the sum of the weights of v and u. The coarse edges connected to c are the fine edges connected to v and u minus the contracted edge $\{v, u\}$. Where multiple edges connect the same two coarse vertices, they are reduced to a single edge with a weight equal to the sum of all the weights of the combined edges.

The coarsening phase is terminated when the coarsest graph contains a smaller number of vertices than some threshold t, $|V_i| < t$. In *KMetis* t is based on the size of G_0, so that the ratio of time taken by initial partitioning in comparison to coarsening and uncoarsening is the same regardless of the size of G_0.

Partitioning of G_m in *KMetis* is handled by its sister program *PMetis*, which performs a recursive bisectioning in order to generate a k-way partitioning. A series of partitionings are generated by *PMetis* [3] for the coarsest graph, and the best partitioning is chosen for P_m.

Just as coarsening is made up of two parts, uncoarsening also consists of two parts: projection and refinement. During the projection part of the uncoarsening phase, the partition labels in P_{i+1} are assigned to the corresponding fine vertices in G_i via C_i. For a fine vertex v this can be expressed as:

$$P_i^v = P_{i+1}^{C_i^v}.$$

The weight of the cut edges and the partition weights are the same for P_{i+1} and P_i at this point.

For the refinement of the k-way partitioning on G_i, the Greedy [15] refinement algorithm is used and operates only on the boundary vertices. At the start of a refinement pass, all of the boundary vertices are added to the priority queue. The priority of a vertex is determined by its potential *gain*. The gain of v is determined by its connectivity to other partitions as detailed by:

$$gain_v = \sum_{u \in \Gamma(v)} \begin{cases} -\theta(\{v,u\}) & \text{if } P_i^u = P_i^v \\ \theta(\{v,u\}) & \text{else} \end{cases}.$$

Vertices are then extracted from this priority queue and considered for moving. If moving a vertex will reduce the total weight of cut edges and still result in balanced partition weights, the vertex is moved and the partition weights are updated. In order to speed up these checks, each vertex stores information about the weight of the edges connecting it to different partitions. If at least one vertex has been moved, and the maximum number of passes has not been exceeded, another pass is attempted with the new boundary vertices. Otherwise the partitioning is projected to the next finer graph.

Uncoarsening then finishes when the partitioning has been projected and refined on the original graph G_0.

B. ParMetis

ParMetis is an MPI based parallel formulation of *KMetis* and as a result achieves parallelism through p processes. Each process is assigned a contiguous chunk of $|V|/p$ vertices, and stores the associated edges and weights. Each process also stores a vector D of length $p+1$ which serves to describe the vertex distribution of the entire graph. A vertex v belongs to process i where $D^i \leq v < D^{i+1}$.

Matching in *ParMetis* is split up into passes. During each pass, processes select heavy edge matchings for their vertices. Matching between vertices local to a process is handled in the same way as in *KMetis*. Matchings with non-local vertices generate requests at the end of a pass that the other processes either grant or reject. After a set number of passes have been performed the graph is contracted. Because each edge is stored once for each incident vertex, contraction can be performed independently by each process once the matching information of adjacent non-local vertices has been communicated. The distribution of G_{i+1} is based on the distribution of G_i, where a process owns the coarse vertices resulting from the matches it generated.

The initial partition in *ParMetis* is made through recursive bisection. The processes perform an all-to-all broadcast resulting in each process having all of G_m. Then using the same random seed they each generate a branch of the bisection tree required to create the k-way partitioning. That is, all processes generate the initial bisection of the graph, then half of the processes generate a bisection of the left partition and half of the processes generate a bisection of the right partition, and so forth. These partitionings are then assembled so that each process has all k partitions applied to its local part of G_m.

Then in uncoarsening, each process projects the partition information from the coarse vertices V_{i+1} that it owns to the fine vertices V_i that it owns. It also transmits the partition information of coarse vertices that it owns and contain fine vertices from other processes.

Due to the serial nature of greedy refinement, *ParMetis* makes some significant diversions from the refinement used in *KMetis*. Selecting moves to make in parallel can result in a

situation where two boundary vertices in different partitions that are connected by a heavy edge are both chosen to move to each other's partitions, which can increase the weight of cut edges. For this reason, *ParMetis* splits each refinement iteration into two passes, once where vertices can only move from partitions with a lower label than the partition they reside in, and a second time where they can only move to partitions with a higher label than the one they reside in.

Furthermore, each of these passes consists of 3 parts. In the first part each process generates a vector of all the moves it would like to make. Then in the second part once the processes communicate how this would affect the partition balance, move requests are rejected until the remaining ones will result in a balanced set of partitions. In the third part, the vertex and partition information is updated for the vertices approved to move. As in *KMetis*, refinement stops when either no vertices are moved, or a maximum number of passes has been performed.

IV. Shared Memory Multilevel Graph Partitioning

In this section we present algorithms for parallelizing *KMetis* on a shared memory system using OpenMP. Performing parallel graph partitioning in shared memory presents more algorithmic options compared to that of using distributed memory. The availability of a shared address space and the locking mechanisms allow us to potentially exploit a more fine grain level of parallelism. We explored different approaches for each of coarsening, initial partitioning, and uncoarsening.

A. Coarsening

The coarsening phase of multilevel graph partitioning first computes a vertex matching and then builds the next-level coarser graph via graph contraction in which the matched vertices are combined and the adjacency lists of the combined vertices are combined.

As discussed in Section III, the matching algorithm used by *KMetis* visits the vertices of the graph in a certain order and then for each unmatched vertex, it matches it with its adjacent unmatched vertex that is connected via a highest weight edge. If no such vertex exists, then the vertex remains unmatched. In parallelizing this algorithm, we followed an approach in which the vertices of the graph are divided among the different threads, and each thread is responsible for matching the vertices assigned to it.

A direct implementation of this approach will use a shared matching vector M and each thread will be reading this vector in order to determine the matching status of the vertices and writing to it every time the matching status of a vertex has been determined. In order to ensure that there are no race conditions, each read/write operation on the shared vector M will need to be protected via a lock. This will result to excessive locking and lead to poor parallel performance. An alternate approach is for each thread to read M without locking but ensure that the write operations (i.e., the matching decisions) are valid. This is achieved as follows. Once a thread has chosen to match vertex v with vertex u, it locks both M^v and M^u and performs a final check to make sure that both vertices are still unmatched. If this holds, it then sets $M^v = u$ and $M^u = v$ before releasing the locks on them. Because the number of vertices in graphs can easily reach into the millions, creating a separate lock for each vertex is not feasible. In order to lock vertices we instead use a fixed number of locks much greater than the number of threads, and use a hashing function to map multiple vertices to the same lock. Acquiring a lock then locks multiple unrelated vertices at once. Locks are acquired in ascending order to prevent deadlocks. We will refer to this scheme as *fine-grain matching*.

Even though the above approach reduces the amount of time spent in locks/unlocks over the naive approach, we expect that it will still incur substantial shared data access synchronization overheads. For this reason, we evaluated another approach that is similar to the iterative two-pass approach used by *ParMetis*. Specifically, the vertices of the graph are initially divided among the threads. Each thread matches its vertices by giving preference to unmatched adjacent vertices that are also assigned to the same thread. Each thread also has a *request buffer* for each other thread. The matchings that involve adjacent vertices assigned to other threads, are placed into the *request* buffer corresponding to that thread. During the second pass, each thread processes the corresponding request buffers of other threads and either accepts or rejects the requested matches made for its vertices and modifies M accordingly. After several passes any unmatched vertices are matched with themselves. We will refer to this scheme as *multi-pass matching*.

The multi-pass matching approach addresses the high synchronization overheads of the fine-grain matching approach but it introduces the extra cost of maintaining and servicing the request buffers. The third approach relies on the heuristic nature of matching and exploits ideas from both the fine-grain and the multi-pass matching approaches. Just as in the fine-grain approach each thread populates M for both local and non-local vertices; however, unlike the fine-grain approach, the writes in M are done unprotected. So it is possible for vertex v to believe that it is matched to u, while vertex u believes it is matched to w. To correct this, after M is generated, each thread goes through its list of local vertices and any vertex v in an asymmetrical matching $M^v = u$ and $M^u \neq v$, are matched with themselves. As long as the number of vertices in the graph is much greater than the number of threads, these asymmetrical matchings occur infrequently so as not to disturb the size of the matching. In our experiments we observed 0.001% of vertices involved in asymmetrical matchings at the finest level, and 0.13% at the coarsest level. This corresponded

to about $120,000$ vertices per thread at the finest level, and about $1,000$ vertices per thread at the coarsest level. This unprotected approach attempts to gain the best of both approaches, by avoiding the synchronization overheads of the fine grained approach, and the extra memory accesses required for handling requests in the multi-pass approach. We will refer to this scheme as *unprotected matching*.

Once we have populated the matching vector M, a parallel prefix-sum is performed over the matchings, so that in a second pass coarse vertex numbers can be assigned to each match in parallel, generating C.

With the matching vector M and its associated vertex mapping vector C, the parallelization of the graph contraction operation on a shared-memory system is straightforward. Irrespective of the scheme used to compute the matching, our parallel graph contraction algorithm operates as follows. The vertices of the next-level coarse graph are divided among the threads and each thread is responsible for merging the adjacent lists of the corresponding matched vertices.

B. Initial Partitioning

Since initial partitioning will always be performed on a small problem size, the design space in which it can effectively be parallelized is quite small. The two approaches we explore are parallelizing each bisection, *parallel bisectioning*, and parallelizing all of the k-way partitionings, *parallel k-sectioning*.

In parallel bisectioning, each thread bisects G_m into G_a and G_b, and the best bisection is selected. The threads are then split into two groups, and one group recursively performs a parallel bisectioning on G_a and the other recursively performs a parallel bisectioning on G_b. This is done until a k-way partitioning is obtained. This requires the threads to synchronize $\log_2 k$ times to perform a min-reduce operation and select the best partitioning. At each bisection, 16 partitionings are made.

In parallel k-sectioning, each thread independently generates k-way partitionings of G_m via recursive bisection, and the best partitioning among all of the threads is selected. Among all of the threads, 16 k-way partitionings are generated. Although the number of parallel tasks in parallel k-sectioning is less than that of parallel bisectioning, the only synchronization point is the min-reduce operation at the end to select the best partitioning.

C. Uncoarsening

The uncoarsening phase of the multilevel graph partitioning paradigm consists of two steps. First, the partition labels P_{i+1} of the next-level coarse graph are projected to the current coarse graph in order to compute P_i, and then a greedy move-based refinement algorithm is used to further reduce the edgecut of the resulting k-way partitioning.

Since all data is accessible by all threads, including the partition label vector P_{i+1}, the projection step can be easily parallelized by dividing the vertices among the threads, and having each thread determine the partition label of its assigned vertices. Because this is a memory bound operation, memory bandwidth can become a limiting factor when attempting to achieve high speedup with a large number of cores.

On the other hand, parallelizing the greedy move-based refinement algorithm is considerably more complicated. First in order to ensure concurrent refinement, the global greedy strategy needs to be relaxed. Second, as different threads move vertices among partitions concurrently, care must be taken to ensure that balance is maintained. For example, the partitioning solution can become unbalanced if thread t_a considers moving vertex v to partition V_i, and at the same time thread t_b considers moving vertex u to partition V_i. When both threads check to make sure their moves will result in balanced partitions, they see $|V_i|+\eta(v)$ and $|V_i|+\eta(u)$ as being below the maximum allowable partition weight, but the resulting weight of $|V_i| + \eta(v) + \eta(u)$ is greater than the maximum allowable partition weight. Third, the concurrent move of vertices can also lead to a degradation of the edgecut, even if these moves were initially selected based on greedy strategy. This happens in the cases in which the vertices that are moved concurrently are connected via an edge. For example, consider two adjacent vertices v and u belonging to partitions one and two, respectively with $\theta(\{v,u\})$ being greater than the sum of the weights of the rest of the edges incident on v and u. In a situation like that, moving either v or u to the other vertex's partition, will improve the edgecut. However, if these two vertices were assigned to different threads, and each thread decided to perform the move, then the overall edgecut will not decrease and depending on the rest of the edges incident on v and u, it can potentially increase. Fourth, the serial implementation of the greedy move-based refinement algorithm in *KMetis* considers only the vertices that are at the partition boundaries and precomputes the per-adjacent partition cuts for each boundary vertex. This information is efficiently updated as vertices get moved, resulting in an extremely fast and tight k-way refinement implementation. As a result, the total time spent in uncoarsening is much smaller than the time spent in coarsening, making hard to hide parallelization overheads.

To address the above challenges, we developed two different parallel formulations of the greedy move-based refinement algorithm. In both approaches, we replaced the single global priority queue with per-thread priority queues. Thus, the boundary vertices are divided among the threads, each thread inserts them in its own priority queue based on their gain, and then proceeds to process them. Note that even though an approach like that may not be as effective as a globally greedy strategy, our experience with schemes like that in *ParMetis* has shown that their overall

impact on partitioning quality is minimal. However, the two formulations differ on how they ensure balance, how they deal with potential cut degrading moves, and how they update the refinement-related data-structures.

The first approach attempts to keep all of the partition information up to date, by performing updates as each move is made. Once a thread has removed a vertex v from the top of its priority queue, it finds the best eligible partition to move v to. A partition V_i is eligible only if its weight plus the weight of v is under the maximum allowable partition weight. The best partition is the one to which the sum of the weight of the edges connecting v to the partition is the greatest. If the best partition for v is the one it is already in, or no eligible partition can be found, v is not moved. Otherwise v and all of its neighbors $\Gamma(v)$ are locked using the same hashing technique as in Section IV-A, and both the partition containing v and partition receiving v are locked as well. Final checks are then performed to ensure the pending weight changes will be within the balance constraint, and that the move will still result in an edgecut reduction. Locking the partition weights and performing the final check, ensures the balance of the partitioning is preserved. Locking v and its neighbors ensures that no moves that increase the edgecut are made, and that the refinement-related data-structures are consistent. Finally, once the updates have been performed, the locks are released. We refer to this approach as *fine-grain refinement*.

There are three potential limitations with the above approach. First it will tend to incur a high synchronization overhead due to the frequent locking. Second it requires the number of partitions be greater than the number of threads in order to keep updating the partition weight from becoming a bottleneck. Third, since the pre-computed refinement-related data-structures are updated by multiple threads, it can lead to false sharing.

To address these issues, the second approach closely follows the algorithm used in *ParMetis*. To ensure that two vertices connected by a sufficiently heavy edge will not swap partitions and increase the edgecut, vertices are restricted to moving across partition boundaries in only one direction at a time. When a thread removes a vertex v from its priority queue, it decides whether or not to move v following the same process as fine-grain refinement (described above), with the added restriction of move direction.

Additionally, each thread has an *update buffer* for each other thread. If it decides to move a vertex v, it updates its local vertices and places updates to adjacent non-local vertices into its corresponding update buffer. After each thread has removed a fixed number of vertices from its priority queue, all threads communicate what the potential partition weights would be after their moves are committed. The balance of the k-way partitioning is maintained by undoing pending moves until the remaining moves would result in a balanced partitioning. The remaining moves are then committed and threads update their local vertices' partition connectivity for the moves made by other threads by reading from their corresponding update buffers. This is then repeated until all of the priority queues are emptied. We refer to this approach as *coarse-grain refinement*.

For both approaches, a pass ends when the priority queues of all threads are empty. Refinement terminates when the maximum number of passes has been reached, or no vertices were moved in the last refinement pass.

D. Thread Lifetimes

Our discussion so far has focused on algorithmic aspects related to the frequency of synchronization and task granularity. However, another equally important aspect has to do with when and for how long spawned threads live, which we will refer to as *thread lifetime*. In this work we explore two different approaches, which we will refer to as *fork-join* and *thread-persistence*.

In the fork-join approach, threads are started at the beginning of a parallel block of work and stopped at the end. This is the paradigm around which OpenMP was created. In the multilevel graph partitioning algorithms we have discussed so far, this means starting threads at the start of matching and joining them at the end, and doing the same for each of contraction, initial partitioning, projection, and refinement. Note that whether threads are spawned and killed for each parallel block of work or pulled from an existing thread pool is implementation dependent.

The thread-persistence approach creates all of the threads at the start of program execution, similar to MPI, and does not join them again until the program's termination. This approach requires threads to synchronize at the same points in the multilevel paradigm where threads were spawned and joined in the fork-join approach.

E. Data Ownership

A design space tightly coupled with thread lifetimes is that of data ownership. In this context, data ownership refers to a thread performing work on a fixed set of vertices and their incident edges. We investigated three approaches to data ownership: one which assigns work dynamically, one which assigns works statically at the start of each block of parallel work, and one for which data ownership persists throughout the entire execution the multilevel paradigm. The fork-join model of OpenMP restricts data ownership to within a parallel block. As a result, for the third approach this accomplished while using the thread-persistence described above.

The first approach has no concept of data ownership and uses dynamic work scheduling. This provides the benefit of dynamic load-balancing at the cost of increased overhead for distributing the work. This approach also fails to preserve data locality. For the multilevel paradigm, this means that threads pull vertices from a shared pool on which to

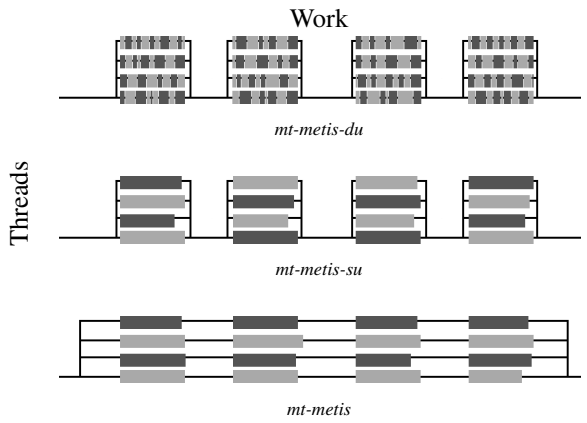


Figure 1. The threading and work distribution models for *mt-metis-du*, *mt-metis-su*, and *mt-metis*, where a line represents a thread and the gray blocks represent chunks of work.

perform matching, contraction, projection, and refinement as described in Sections IV-A and IV-C. The initial partitioning approaches we took in IV-B are unaffected by this design space. We will refer to this approach as *dynamic work-distribution*.

In a second approach to data ownership, threads are given ownership of data statically at the start of a block of parallel work. This approach relies on the even distribution of work at the start of each parallel work block, but does not suffer from the overhead associated with dynamic load-balancing. Data locality with this approach is limited to the parallel work block. In the multilevel graph partitioning paradigm, this means vertices are divided among the threads at the start of matching, contraction, projection and refinement for each of the graphs $G_0, G_1, \ldots, G_m$. We will refer to this approach as *static work-distribution*.

In the third approach, threads are given ownership of data at the start of the program, and continue to own that data and all derived data for the duration of the program. This approach is not possible with the fork-join approach to thread lifetimes. This approach ensures the locality of the data worked on by each thread. However, not only does this approach lack dynamic load-balancing, it also provides no guarantee that the work will be evenly distributed at the start of a work block. In the context of the multilevel graph partitioning paradigm, this means that the vertices of G_0 are initially divided among the threads, and in G_1 threads own the coarse vertices they created while contracting the fine vertices of G_0 that they owned. We will refer to this approach as *persistent work-distribution*.

For evaluating these data ownership approaches as well as the associated thread lifetime approaches discussed in Section IV-D, we developed three OpenMP based implementations of the algorithms discussed in Sections IV-A, IV-B, and IV-C. The first, *mt-metis-du*, uses the fork-join approach for thread lifetimes and the dynamic work-distribution approach for data ownership. The second implementation, *mt-metis-su*, also uses the fork-join approach for thread lifetimes, but uses the static work-distribution approach for data ownership. The third implementation, *mt-metis*, uses the thread-persistent approach to thread lifetimes and the persistent work-distribution approach for data ownership. These differences are illustrated by Figure 1.

Table I
THE FOUR GRAPHS AND THEIR PROPERTIES USED IN THESE EXPERIMENTS

Name	Type	# Vertices	# Edges
DFEG	Dual	988,605	1,947,069
NFEG	Nodal	1,118,496	31,255,782
RDMAP	Road Network	23,947,347	28,854,312
VLSICRCT	VLSI Circuit	49,375,364	76,768,132

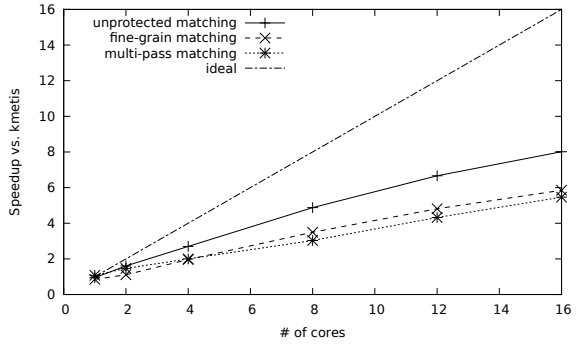
V. Experimental Design

Experiments were performed on four different graphs, representing three different domains of graph partitioning. The first two (DFEG & NFEG) correspond to dual and nodal graphs of 3D meshes used in scientific computing. The third, RDMAP, corresponds to the road network of the US [23]. The fourth one (VLSICRCT) corresponds to the netlist of a VLSI circuit. In deriving VLSICRCT, we only kept the nets corresponding to edges in the original design. The size of these graphs are shown in Table I.

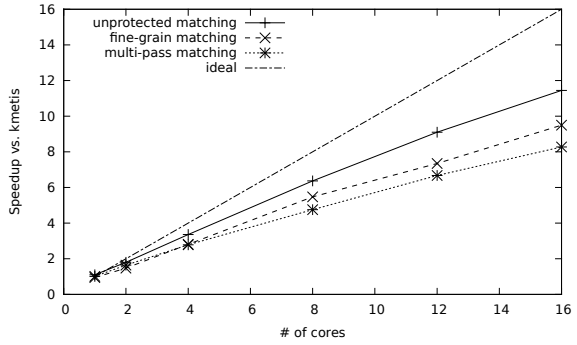
Each run was repeated 50 times with a different starting random seed, and the average time and edgecut were taken. An imbalance tolerance of 3% was used for all partitionings, all of which were 64-way. Run times are only for the three phases of multilevel partitioning, and not for IO. We used the k-way partitioning portion (referred to in this paper as *KMetis*) of *Metis* 5.0.2, *ParMetis* 4.0.2, and *PT-Scotch* 5.1.2. The default settings were used for both *KMetis* and *ParMetis*. Both the *scalability* and *speed* flags were used when running *PT-Scotch*.

Speedup is measured with respect to the runtime of *KMetis*. Where speedups are aggregated for all four graphs, the geometric mean has been taken of their speedup with respect to *KMetis*. Edgecut is measured relative to *KMetis*. Where edgecuts are aggregated for all four graphs, their geometric mean has been taken relative to *KMetis*.

Three systems were used for these experiments: an HP ProLiant BL280c G6 with 2x 8-core Xeon E5-2670 @ 2.6 GHz, a Dell PowerEdge R815 with 4x 8-core Opteron 6220 @ 3.0 GHz, and a Sun Fire X4600 with 8x 4-core Opteron 8356 @ 2.3 GHz. Codes were compiled using Intel's ICC compiler version 11.1 on the 8x 4-core Opteron system, and 11.2 on the 2x 8-core Xeon and 4x 8-core Opteron systems, all with *O3* optimizations enabled. Regarding the thread lifetimes discussed in Section IV-D, Intel's implementation of OpenMP, as used in these experiments, makes use of thread pooling [24]. We were not able to produce reliable

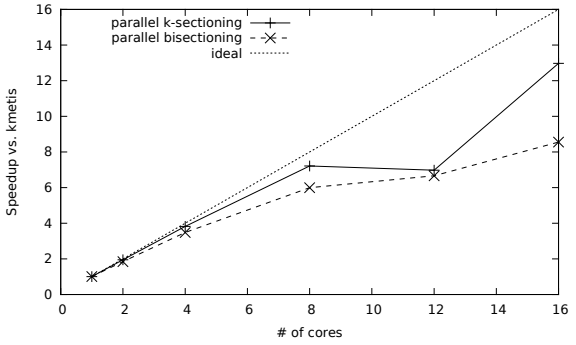


(a) DFEG

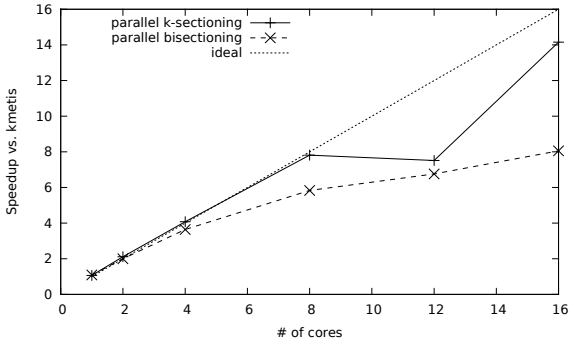


(b) VLSICRCT

Figure 2. Coarsening



(a) DFEG



(b) VLSICRCT

Figure 3. Initial Partitioning

results when utilizing all 32 cores of the 4x 8-core Opteron system for any of the partitioners, and as a result we only report using up to 28 cores here.

VI. Results

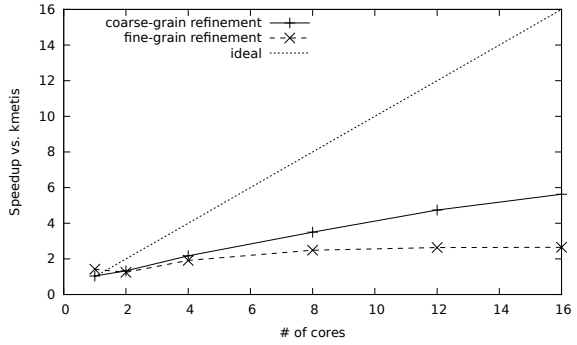
Our experimental evaluation consists of three parts. First we evaluate the impact of the various algorithmic choices for the various phases of the multilevel paradigm as it relates to task granularity and synchronization frequency. Second, we evaluate the impact of thread lifetime and data ownership. Finally, we evaluate the performance of the best performing OpenMP formulation of *KMetis* against the performance achieved by two MPI-based parallel multilevel algorithms, *ParMetis* and *PT-Scotch*.

A. Granularity of Parallelism

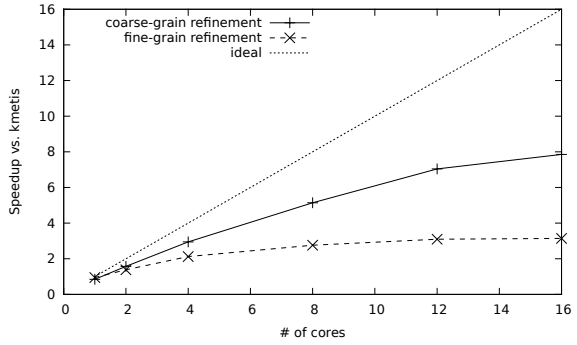
To study the effect of the various algorithmic choices for coarsening, initial partitioning, and refinement, we performed a series of experiments in which the various algorithms were implemented and evaluated using the *mt-metis-du* framework as described in Section IV-E. Due to space constraints we only present results for the smallest and largest graphs in our dataset, DFEG and VLSICRCT, on the 2x 8-core Xeon system; however the results on the other systems were similar.

1) Coarsening: The results of the three coarsening approaches: fine-grain matching, multi-pass matching, and unprotected matching are shown in Figure 2. The results show that the unprotected approach outperformed the other two. The better performance of the fine-grain approach over the multi-pass approach can be attributed to the following to reasons. First, the fine-grain approach did not suffer from lock contention because the number of locks was much greater than the number of threads. Second, the overhead associated with acquiring and releasing a lock per matching in the fine-grain approach was slightly less than that of the extra memory accesses required by the request buffer used by the multi-pass approach. The unprotected approach avoids these overheads and using 16 threads achieved a speedup of 8.7 for DFEG and 11.4 for VLSICRCT, compared to the speedups of 5.5 and 8.3 achieved by the multi-pass approach and the speedups of 5.7 and 9.4 achieved by the fine-grain approach respectively.

2) Initial Partitioning: The results of the two approaches for initial partitioning, parallel bisectioning and parallel k-sectioning, are shown in Figure 3. The parallel k-sectioning approached scaled near linearly when the number of threads was a power of two. This is a result of the 16 partitionings being evenly divided and each partitioning being generated independently. It suffered a slight decrease in speed when using twelve threads because four of the threads still generated



(a) DFEG

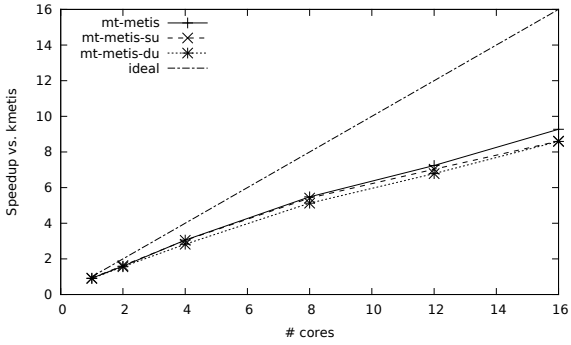


(b) VLSICRCT

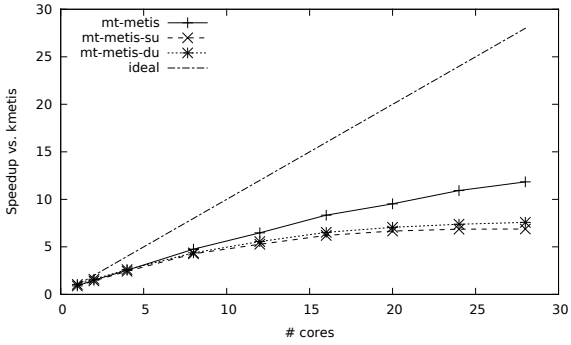
Figure 4. Uncoarsening

two partitionings causing the other eight threads that each only generated one partitioning to wait. Parallel recursive bisectioning did not scale as well because of the increased number of synchronization points. However, because of the finer grain task decomposition, parallel recursive bisectioning still had a balanced work distribution when using twelve threads.

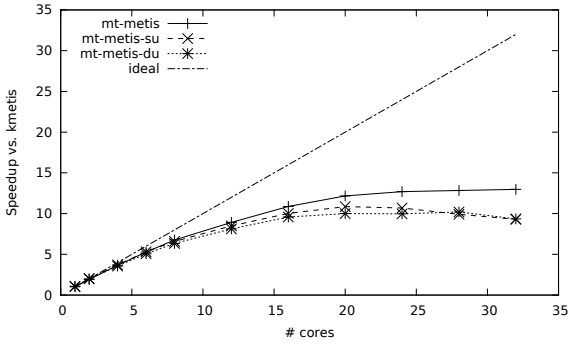
3) Uncoarsening: The results of the two approaches for uncoarsening, fine-grain refinement and coarse-grain refinement, are shown in Figure 4. From these results we can see that the coarse-grain approach outperforms the fine-grain approach. The fine-grain approach manages to gain a small speedup of 2.5 for DFEG and 2.8 for VLSICRCT using eight threads, and moves up to a speedup of 2.6 for DFEG and 3.1 for VLSICRCT using 16 threads. This plateau is likely the result of lock contention for modifying the partition weights. The coarse-grain approach however, steadily gained speedup as the number of threads increased, with a speedup of 3.5 for DFEG and 5.1 for VLSICRCT using eight threads, and a speedup of 5.6 for DFEG and 7.8 for VLSICRCT using 16 threads. The greater speedup for the large graph compared to the small graph is because the majority of the work in the uncoarsening phase is only on border vertices, which account for a small fraction of the total vertices in the graph, and as a result the parallel overheads are not as well hidden while performing refinement on the smaller graph.



(a) 2x 8-core Xeon



(b) 4x 8-core Opteron



(c) 8x 4-core Opteron

Figure 5. Mean speedup of the threading approaches on the three different systems.

B. Thread Lifetimes and Data Ownership

The best granularity strategies for coarsening, initial partitioning, and uncoarsening as determined by the outcome of these experiments, were used in the multi-threaded implementations to explore the thread lifetime and data ownership design space discussed in Sections IV-D and IV-E. That is, for matching we used the unprotected matching approach described in Section IV-A, for initial partitioning we used the parallel k-sectioning method described in Section IV-B, and for uncoarsening we used the coarse-grain refinement described in Section IV-C.

The mean speedup across all four graphs on each of the

three systems for these three implementations can be seen in Figure 5. On the 2x 8-core Xeon system we can see that the three multi-threaded implementations performed relatively similarly, with *mt-metis* taking a slight lead when using 16 threads. On the two systems with higher core counts, *mt-metis* has a prominent lead over *mt-metis-du* and *mt-metis-su* when using more than 16 threads. This performance gap exists because *mt-metis-du* and *mt-metis-su* are not designed to preserve data ownership across synchronization points (i.e., transitioning from matching to contraction, projection to refinement, and moving between levels in coarsening and uncoarsening). When one of these synchronization points are encountered, a thread may work on a different part of the graph than it had previously. Data ownership is preserved by *mt-metis*, so when the active level of the graph is small enough such that the portion assigned to a thread fits within its cache, it performs the majority of its memory accesses from cache where *mt-metis-du* and *mt-metis-su* are still accessing the slower DRAM the majority of the time. As the number of threads increases, the portion of the graph assigned to each thread decreases and a larger level of the graph can fit within the cache. This increased data locality enables *mt-metis* to outperform *mt-metis-du* and *mt-metis-su* in both coarsening and uncoarsening for a large number of threads.

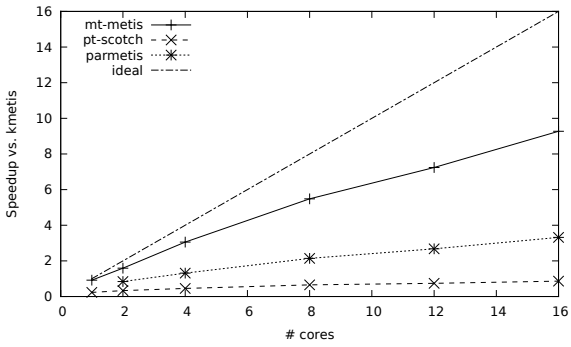
The better utilization of the aggregate available cache also explains the relative performance of *mt-metis* over the other methods on the three different architectures. Specifically, *mt-metis* achieved the best relative performance on the 4x 8-core Opteron system that has 1MB of L2 cache/core, its second best performance on the 8x 4-core Opteron system that has 512KB of L2 cache/core, and its worst relative performance (though still better) on the 2x 8-core Xeon system that has only 256KB of L2 cache/core.

C. Comparison with Other Partitioners

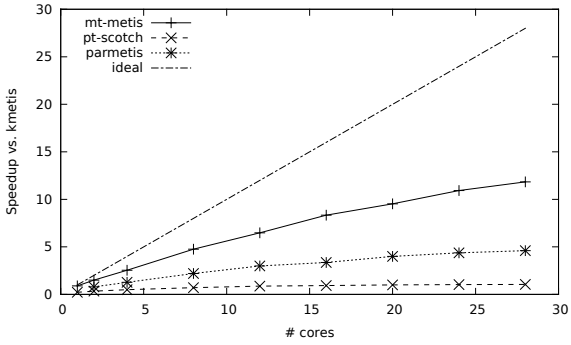
We have compared our best multi-threaded implementation, *mt-metis*, with two publicly available distributed memory partitioners, *ParMetis* [8] and *PT-Scotch* [10].

1) Speedup: The mean speedup for partitioning the four graphs can be seen for each system in Figure 6. In all three of the figures, it can be seen that *mt-metis* averaged over twice the speedup of both *ParMetis* and *PT-Scotch* when using more than four cores on each of the three systems.

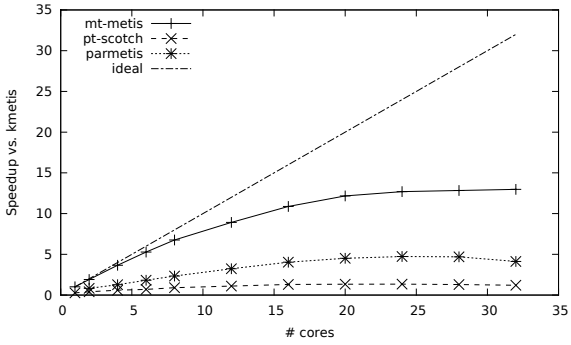
The speedups achieved partitioning the individual graphs on each system are shown in Table II, and the run times are shown in Table III. The achieved speedups are largely dependent on the graph, and to a lesser extent, the system. The worst that *mt-metis* performed relative to the other partitioners, was partitioning DFEG on the 4x 8-core Opteron system. *ParMetis* reached a speedup of 6.9 compared to the 8.9 speedup of *mt-metis*. On the same system however, *mt-metis* saw its largest lead in performance with a speedup of 17.9 compared to *ParMetis*'s speedup of 2.9 and *PT-Scotch*'s



(a) 2x 8-core Xeon



(b) 4x 8-core Opteron



(c) 8x 4-core Opteron

Figure 6. Mean speedups of the distributed memory partitioners and *mt-metis* on the three different systems.

speedup of 1.5 when partitioning VLSICRCT. As *mt-metis* and *ParMetis* implement similar algorithms, a great deal of the difference in their runtimes can be attributed to the overheads of message passing. Furthermore, as coarsening is the most time consuming stage of multi-level graph partitioning, *mt-metis*'s unprotected matching provides a significant advantage over the request based scheme of *ParMetis*.

Note that *PT-Scotch* uses recursive bisection to generate a k-way partitioning [10], causing it to go through the multilevel process several times. On the other hand *ParMetis* and *mt-metis* only use recursive bisectioning to generate the

Table II
SPEEDUP ON INDIVIDUAL GRAPHS (VS *KMetis*)

	2x 8-core Xeon			4x 8-core Opteron (28 cores)			8x 4-core Opteron		
Graph	*ParMetis*	*PT-Scotch*	*mt-metis*	*ParMetis*	*PT-Scotch*	*mt-metis*	*ParMetis*	*PT-Scotch*	*mt-metis*
DFEG	3.395	0.677	6.56	5.195	0.856	7.888	6.862	1.314	8.924
NFEG	2.466	0.366	7.57	3.273	0.479	10.535	3.142	0.470	12.036
RDMAP	3.703	1.500	11.658	5.681	2.078	14.886	4.689	2.197	14.733
VLSICRCT	3.909	1.491	12.755	4.633	1.411	15.8962	2.881	1.534	17.891

Table III
TIME ON INDIVIDUAL GRAPHS (SECONDS)

	2x 8-core Xeon			4x 8-core Opteron (28 cores)			8x 4-core Opteron		
Graph	*ParMetis*	*PT-Scotch*	*mt-metis*	*ParMetis*	*PT-Scotch*	*mt-metis*	*ParMetis*	*PT-Scotch*	*mt-metis*
DFEG	0.233	1.168	0.121	0.281	1.697	0.185	0.585	3.057	0.450
NFEG	1.421	9.566	0.463	1.892	12.921	0.588	6.746	45.146	1.761
RDMAP	7.401	18.276	2.351	6.884	18.817	2.627	24.445	52.155	7.779
VLSICRCT	17.416	45.653	5.337	19.387	63.671	5.651	96.578	181.391	15.553

Table IV
MEMORY USAGE (MB)

	Cores					
Name	1	2	4	8	16	32
KMetis	522	-	-	-	-	-
ParMetis	522	1,218	1,501	1,895	2,899	4,691
PT-Scotch	593	742	1,019	1,241	2,251	5,001
mt-metis	665	680	700	696	742	752

Table V
GEOMETRIC MEANS OF AVERAGE CUTS SCALED RELATIVE TO *KMetis*

	Cores					
Name	1	2	4	8	16	32
KMetis	1.000	-	-	-	-	-
ParMetis	1.000	1.131	1.221	1.116	1.113	1.108
PT-Scotch	1.111	1.113	1.113	1.089	1.102	1.100
mt-metis	1.075	1.072	1.076	1.085	1.104	1.102

Table VI
GEOMETRIC MEANS OF MINIMUM CUTS SCALED RELATIVE TO *KMetis*

	Cores					
Name	1	2	4	8	16	32
KMetis	1.000	-	-	-	-	-
ParMetis	1.000	1.063	1.060	1.056	1.049	1.047
PT-Scotch	1.037	1.039	1.037	1.042	1.029	1.033
mt-metis	1.033	1.041	1.040	1.031	1.050	1.048

initial partition, and as a result only go through the multilevel process once. This contributed to its higher runtime seen in our experiments.

2) Memory Usage: In Table IV, we present the aggregate memory usage of these partitioners for creating k-way partitioning of DFEG. Memory usage was measured using the GNU *time* utility. It can be seen that all three of the parallel partitioners increase their memory usage with the number of cores utilized. However, where both *ParMetis* and *PT-Scotch* use over eight times the amount of memory of *KMetis* on 32 cores, *mt-metis* only uses 44% more than *KMetis*, and only 13% more than it did running serially. Thread private data structures used during coarsening and uncoarsening account for the slight increase in memory usage by *mt-metis* as the number of threads increases. The large difference in memory usage between *mt-metis* and the MPI based partitioners is in large part because *mt-metis* stores information for each vertex only once, where *PT-Scotch* and *ParMetis* need to communicate and store the information of remote neighbor vertices.

3) Partition Quality: To ensure a valid comparison, we studied the average and minimum number of cut edges of the partitions of the four graphs generated by the partitioners. Each partitioner generated 50 partitionings of each graph. The geometric mean of both the average edgecut and minimum edgecut relative to *KMetis* are shown in Tables V and VI. The tables show that the quality of partitionings created by *mt-metis* does not diverge from that of *ParMetis* and *PT-Scotch*.

VII. SUMMARY AND FUTURE WORK

In this paper we explored the design space of multi-threaded graph partitioning, specifically using OpenMP, and demonstrated the performance improvement it can offer over traditional MPI codes on multi-core/multi-processor machines. Our final implementation, *mt-metis*, is on average over twice as fast as *ParMetis* and *PT-Scotch*. This speedup is due to a combination of avoiding message passing overheads and modifying the existing parallel algorithms used in *ParMetis*. Specifically the unprotected matching scheme significantly reduces the runtime of the most time consuming phase of multilevel graph partitioning. Beyond the improved speedup, *mt-metis* also uses significantly less total memory than either *ParMetis* or *PT-Scotch*. This reduced memory footprint plays an important role in enabling the partitioning

of large graphs on modern machines that have a decreasing memory to processing element ratio.

ACKNOWLEDGEMENT

This work was supported in part by NSF (IOS-0820730, IIS-0905220, OCI-1048018, CNS-1162405, and IIS-1247632) and the Digital Technology Center at the University of Minnesota. Access to research and computing facilities was provided by the Digital Technology Center and the Minnesota Supercomputing Institute.

REFERENCES

[1] T. N. Bui and C. Jones, "Finding good approximate vertex and edge partitions is np-hard," *Information Processing Letters*, vol. 42, no. 3, pp. 153 – 159, 1992.

[2] B. Hendrickson and R. Leland, "A multilevel algorithm for partitioning graphs," in *Proceedings of the 1995 ACM/IEEE conference on Supercomputing (CDROM)*, ser. Supercomputing '95. New York, NY, USA: ACM, 1995. [Online]. Available: http://doi.acm.org/10.1145/224170.224228

[3] G. Karypis and V. Kumar, "A fast and high quality multilevel scheme for partitioning irregular graphs," *SIAM J. Sci. Comput.*, vol. 20, no. 1, pp. 359–392, Dec. 1998. [Online]. Available: http://dx.doi.org/10.1137/S1064827595287997

[4] C. Walshaw and M. Cross, "Mesh Partitioning: a Multilevel Balancing and Refinement Algorithm," *SIAM J. Sci. Comput.*, vol. 22, no. 1, pp. 63–80, 2000.

[5] B. Monien and S. Schamberger, "Graph partitioning with the party library: Helpful-sets in practice," in *Proceedings of the 16th Symposium on Computer Architecture and High Performance Computing*, ser. SBAC-PAD '04. Washington, DC, USA: IEEE Computer Society, 2004, pp. 198–205. [Online]. Available: http://dx.doi.org/10.1109/SBAC-PAD.2004.18

[6] F. Pellegrini and J. Roman, "Scotch: A software package for static mapping by dual recursive bipartitioning of process and architecture graphs," in *Proceedings of the International Conference and Exhibition on High-Performance Computing and Networking*, ser. HPCN Europe 1996. London, UK, UK: Springer-Verlag, 1996, pp. 493–498. [Online]. Available: http://dl.acm.org/citation.cfm?id=645560.658570

[7] P. Sanders and C. Schulz, "Distributed evolutionary graph partitioning," *CoRR*, vol. abs/1110.0477, 2011.

[8] G. Karypis and V. Kumar, "Parallel multilevel k-way partitioning scheme for irregular graphs," in *Proceedings of the 1996 ACM/IEEE conference on Supercomputing (CDROM)*, ser. Supercomputing '96. Washington, DC, USA: IEEE Computer Society, 1996. [Online]. Available: http://dx.doi.org/10.1145/369028.369103

[9] C. Walshaw and M. Cross, "Parallel Optimisation Algorithms for Multilevel Mesh Partitioning," *Parallel Comput.*, vol. 26, no. 12, pp. 1635–1660, 2000.

[10] F. Pellegrini, "PT-Scotch and libScotch 5.1 User's Guide," Aug. 2008, 76 pages User's manual. [Online]. Available: http://hal.archives-ouvertes.fr/hal-00410328

[11] P. Sanders and C. Schulz, "Engineering multilevel graph partitioning algorithms," in *Algorithms Ž013 ESA 2011*, ser. Lecture Notes in Computer Science, C. Demetrescu and M. Halldrsson, Eds. Springer Berlin / Heidelberg, 2011, vol. 6942, pp. 469–480. [Online]. Available: http://dx.doi.org/10.1007/978-3-642-23719-5_40

[12] K. Bergman, S. Borkar, D. Campbell, W. Carlson, W. Dally, M. Denneau, P. Franzon, W. Harrod, J. Hiller, S. Karp *et al.*, "Exascale computing study: Technology challenges in achieving exascale systems peter kogge, editor & study lead," 2008.

[13] B. Hendrickson and R. Leland, "A multilevel algorithm for partitioning graphs," Sandia National Labratories, Tech. Rep. SAND93-1301, 1993.

[14] K. Schloegel, G. Karypis, V. Kumar, J. Dongarra, I. Foster, G. Fox, K. Kennedy, A. White, and M. Kaufmann, "Graph partitioning for high performance scientific simulations," 2000.

[15] G. Karypis and V. Kumar, "Multilevel graph partitioning schemes," in *ICPP (3)*, 1995, pp. 113–122.

[16] ——, "Multilevel k-way hypergraph partitioning," in *Proceedings of the 36th annual ACM/IEEE Design Automation Conference*, ser. DAC '99. New York, NY, USA: ACM, 1999, pp. 343–348. [Online]. Available: http://doi.acm.org/10.1145/309847.309954

[17] I. Safro, D. Ron, and A. Brandt, "Multilevel algorithms for linear ordering problems," *J. Exp. Algorithmics*, vol. 13, pp. 4:1.4–4:1.20, Feb. 2009. [Online]. Available: http://doi.acm.org/10.1145/1412228.1412232

[18] C. Chevalier and I. Safro, "Learning and intelligent optimization," T. Stützle, Ed. Berlin, Heidelberg: Springer-Verlag, 2009, ch. Comparison of Coarsening Schemes for Multilevel Graph Partitioning, pp. 191–205. [Online]. Available: http://dx.doi.org/10.1007/978-3-642-11169-3_14

[19] B. W. Kernighan and S. Lin, "An Efficient Heuristic Procedure for Partitioning Graphs," *The Bell system technical journal*, vol. 49, no. 1, pp. 291–307, 1970.

[20] C. Fiduccia and R. Mattheyses, "A linear-time heuristic for improving network partitions," in *Design Automation, 1982. 19th Conference on*, june 1982, pp. 175 –181.

[21] D. Delling, A. V. Goldberg, I. Razenshteyn, and R. F. F. Werneck, "Graph partitioning with natural cuts," in *IPDPS*. IEEE, 2011, pp. 1135–1146.

[22] G. Karypis and V. Kumar, "Multilevel k-way partitioning scheme for irregular graphs," *Journal of Parallel and Distributed Computing*, vol. 48, pp. 96–129, 1998.

[23] C. Demetrescu, A. Golberg, and D. Johnson, "Challenge benchmarks," 9th DIMACS Implementation Challenge - Shortest Paths, 2006.

[24] *Intel C++ Compiler 12.1 User and Reference Guides*, Intel, 2012. [Online]. Available: http://software.intel.com/sites/products/documentation/hpc/composerxe/en-us/2011Update/cpp/lin/index.htm