Parallel Graph Partitioning on Multicore Architectures

Xin Sui[1], Donald Nguyen[1], Martin Burtscher[2], and Keshav Pingali[1,3]

[1] Department of Computer Science, University of Texas at Austin
[2] Department of Computer Science, Texas State University-San Marcos
[3] Institute for Computational Engineering and Sciences, University of Texas at Austin

Abstract. Graph partitioning is a common and frequent preprocessing step in many high-performance parallel applications on distributed- and shared-memory architectures. It is used to distribute graphs across memory and to improve spatial locality. There are several parallel implementations of graph partitioning for distributed-memory architectures. In this paper, we present a parallel graph partitioner that implements a variation of the Metis partitioner for shared-memory, multicore architectures. We show that (1) the parallelism in this algorithm is an instance of the general amorphous data-parallelism pattern, and (2) a parallel implementation can be derived systematically from a sequential specification of the algorithm. The resulting program can be executed in parallel using the Galois system for optimistic parallelization. The scalability of this parallel implementation compares favorably with that of a publicly available, hand-parallelized C implementation of the algorithm, ParMetis, but absolute performance is lower because of missing sequential optimizations in our system. On a set of 15 large, publicly available graphs, we achieve an average scalability of 2.98X on 8 cores with our implementation, compared with 1.77X for ParMetis, and we achieve an average speedup of 2.80X over Metis, compared with 3.60X for ParMetis. These results show that our systematic approach for parallelizing irregular algorithms on multicore architectures is promising.

1 Introduction

Graph partitioning is a common preprocessing step in many high-performance parallel algorithms. It is used to find partitions of graph nodes such that each partition has roughly the same number of nodes and the sum of the weights of cross-partition edges is minimized. If the number of nodes in a partition is proportional to the amount of work involved in processing that partition and the edge weights are a measure of communication costs, such a partition attempts to achieve load balance while minimizing the inter-processor communication cost.

Graph partitioning is useful in distributed architectures where partitioning can reduce the amount of explicit communication between distributed processing elements. In shared memory settings, partitioning is useful for reducing memory contention and increasing spatial locality.

More formally, we define a weighted, undirected graph in the usual way: $G = (V, E)$, and w is a function assigning weights to each edge $(u, v) \in E$. For any subset of vertices $V_i \subseteq V$, the *cut set* induced by V_i is $C_i = \{(u, v) \in E \mid u \in V_i, v \in V - V_i\}$. The *value* (or *edge cut*) of the cut set C_i is $w_i = \sum_{e \in C} w(e)$. The subsets $P = V_1, V_2, \ldots, V_k$ are a *k-way partitioning* iff (1) $\cup_i V_i = V$ and (2) $\forall i, j : i \neq j \rightarrow V_i \cap V_j = \emptyset$. The balance of P is

$$B(V_1, V_2, \ldots, V_k) = \frac{k * \max_{i=1}^{k} w_i}{\sum_{i=1}^{k} w_i}$$

The *graph partitioning problem* is, given k and G, to find a k-way partitioning P such that the balance of P and the edge cut of P (i.e., $\sum w_i$) are minimized.

The most widely used graph partitioner is the sequential partitioner Metis from Karypis *et al.* [8]. There are several parallel graph partitioners: ParMetis [6] from Karypis *et al.* as well as PT-Scotch [1] and JOSTLE [13]. All of these implementations are explicitly parallel programs for distributed-memory architectures. With the rise of multicore, we are interested in parallel implementations that can take advantage of the lightweight synchronization and communication available on multicore machines. Additionally, we prefer implementations that hew as close as possible to their original sequential implementations as it is usually easier to write and determine the correctness of sequential programs.

The approach that we adopt in this paper is as follows. First, we recognize that the Metis algorithm exhibits a generalized form of data-parallelism that we call amorphous data-parallelism [12]. Second, we implement a version of Metis that exploits amorphous data-parallelism, and we show that it is possible to achieve better scalability than existing parallelizations without resorting to explicit parallel programming.

This paper is organized as follows. Section 2 describes the existing implementations of Metis, sequential and parallel, in detail. Section 3 introduces amorphous data-parallelism, and Section 4 describes the Galois system, a programming model and runtime system designed to exploit amorphous data-parallelism. In Section 5, we show how the Metis algorithm exhibits amorphous data-parallelism and how to implement it using the Galois system. Section 6 discusses results, and Section 7 presents conclusions from this study.

2 Graph Partitioning Algorithms

Metis is one of the most widely used graph partitioners. Recent work has shown that ParMetis is generally the best parallel graph partitioner for distributed-memory systems [5]. In this section, we describe both of these partitioners in more detail.

2.1 Metis

Metis is composed of a set of algorithms for graph and mesh partitioning. The key algorithm is KMetis. It uses a multilevel scheme that coarsens input graphs until they are small enough to employ more direct techniques and then interpolates the results of the smaller graph back onto the input graph. It consists of three phases: coarsening, initial partitioning, and refinement (see Figure 1).

Coarsening. The goal of coarsening is to construct a smaller graph from the input graph that preserves most of the connectivity information. This is achieved by constructing a sequence of graphs, each of which is obtained by contracting edges in the previous graph (see Figure 3). When contracting an edge (a, b), a new node is created in the coarse graph, and its weight is set to the sum of the weights of nodes a and b. If nodes a and b are both connected to a node c in the fine graph, a new edge with a weight equal to the sum of the weights of edges (a, c) and (b, c) is created in the coarse graph.

To find edges to collapse, KMetis computes a maximal matching. A matching is a set of edges in the graph that do not share any nodes. A matching is maximal if it is not possible to add any edges into it. There are several heuristics for finding the maximal matching. KMetis employs a heuristic called heavy edge matching. Heavy edge matching finds maximal matchings whose edges have large weight (Figure 2(b)). Collapsing such edges will greatly decrease the edge weights in the coarse graph and improve the quality of the resulting partitioning. The nodes in the graph are visited randomly, and each node is matched to the unmatched neighbor with the maximal edge weight. If there is no such neighbor, the node is matched to itself.

The coarsening phase ends when the number of nodes in the coarse graph is less than some threshold or the reduction in the size of successive graphs is less than some factor.

Initial partitioning. In this phase, a recursive bisection algorithm called PMetis is used to partition the coarsest graph. Each bisection has the same phases as KMetis except that PMetis (1) uses a breadth-first traversal to perform an initial bisection and (2) employs the Kernighan-Lin heuristic [9] to improve the quality of the bisection. Compared to KMetis, PMetis is slower when the desired number of partitions is large because it coarsens the input graph multiple times. Usually, this phase represents only a small fraction of the overall partitioning time.

Refinement. In this phase, the initial partitioning is projected back on the original graph via the sequence of graphs created in the coarsening phase (Figure 2(c)). KMetis uses a simplified version of the Kernighan-Lin heuristic called random k-way refinement. The nodes of the graph that are on the boundary between partitions are visited randomly. A boundary node is moved to its neighboring partition if doing so reduces the edge cut without leading to a significant imbalance in the partitioning.

2.2 ParMetis

ParMetis is a parallelization of KMetis using MPI. In ParMetis, each process owns a random portion of the input graph. Each of the phases of KMetis is parallelized as follows.

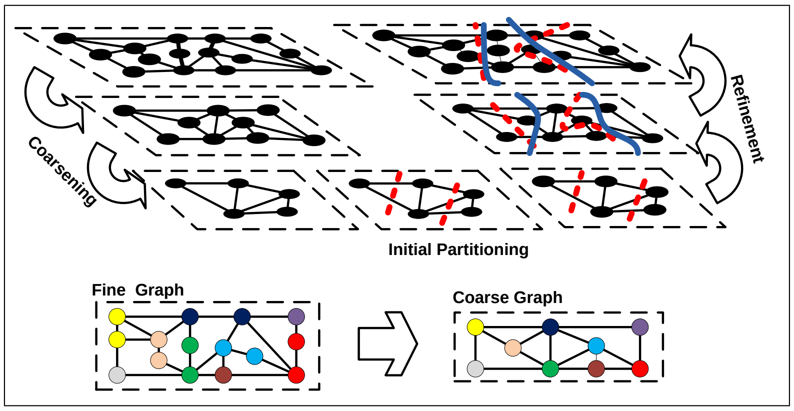


Fig. 1. The phases of a multilevel partitioning algorithm. Dashed lines illustrate the partitioning projected from the coarse graph, and solid shaded lines illustrate the refined partitioning.

```
Graph g = // Read in graph
int k = // Number of partitions
Graph original = g;
do {
  heavyEdgeMatching(g);
  Graph cg = coarsen(g);
  cg.setFinerGraph(g);
  g = cg;
} while (!g.coarseEnough());
PMetis.partition(g, k);
while (g != original) {
  Graph fg = g.finerGraph();
  g.projectPartitioning(fg);
  fg.makeInfoForRefinement();
  refine(fg);
  g = fg;
}
```

(a) Pseudocode for main Metis algorithm.

```
void heavyEdgeMatching(Graph g) {
  // Randomly access
  foreach (Node n : g) {
    if (n.isMatched()) continue;
    Node match = n.findMatch();
    g.setMatch(n, match);
  }
}
```

(b) Pseudocode for heavy edge matching.

```
void refine(Graph g) {
  Worklist wl = new Worklist();
  foreach (Node n : g.boundaryNodes()) {
    if (...) // Moving n to neighbor partition reduces edge cut
      wl.add(n);
  }
  foreach (Node n : wl) {
    part = // Neighbor partition with max edge cut gain;
    if (...) // Balancing condition is not violated
      moveNode(n, part);
  }
}
```

(c) Pseudocode for random k-way refinement.

Fig. 2. Pseudocode for Metis algorithm. Foreach loops indicate the presence of amorphous data-parallelism.

```
Graph coarsen(Graph graph) {
  Graph cg = new Graph(); // Coarse graph
  for (Node n : graph) {
    if (n.visited) continue;
    Node cn = cg.createNode(n.weight+n.getMatch().weight);
    n.setRepresentative(cn);
    n.getMatch().setRepresentative(cn);
    n.getMatch().setVisited(true);
  }
  ... // Reset visited field for each node in the graph
  foreach (Node n : graph) {
    if (n.visited) continue;
    // Add edges in cg according to n's neighbors
    for (Node nn : n.getNeighbors()) {
      Edge e = graph.getNeighbor(n, nn);
      Node cn = n.getRepresentative();
      Node cnn = nn.getRepresentative();
      Edge ce = cg.getEdge(cn, cnn);
      if (ce == null) { cg.addEdge(cn, cnn, e.getWeight()); }
      else { ce.increaseWeight(e.getWeight()); }
      ... // Add edges in cg according to n.getMatch()'s neighbors,
      ... // and similarly for n
      n.getMatch().setVisited(true);
    }
  }
  return cg;
}
```

Fig. 3. Pseudocode for creating a coarse graph. Foreach loops indicate the presence of amorphous data-parallelism.

Coarsening. The parallelization of heavy edge matching proceeds in alternating even and odd rounds. There are two steps in each round. In the first step, each process scans its local unmatched nodes, and for each node v, each process tries to find a neighbor node u to match using the heavy edge matching heuristic. There are three cases for u: (1) if u is stored locally, then the matching is completed immediately; (2) if the round is odd and $v < u$ or the round is even and $v > u$, the process sends a matching request to the process owning u; or (3) otherwise, the match is deferred to the next round. In the second step, each process responds to its matching requests. Processes break conflicts arbitrarily and notify senders on whether their matching requests were successful. Heavy edge matching terminates when some large fraction of the nodes has been matched.

Initial partitioning. ParMetis does not use PMetis for its initial partitioning phase. Instead, it uses the following parallel algorithm: all the pieces of the graph are scattered to all threads using an all-to-all broadcast operation. Then, each process explores a single path of the recursive bisection tree. The recursive bisection is based on nested dissection [4].

Refinement. The random k-way refinement algorithm is parallelized similarly to heavy edge matching. The algorithm runs in rounds. Each round is divided into two steps. In the first step, nodes can only be moved to higher partitions. During the second step, nodes can only be moved to lower partitions. This alternation pattern helps avoid situations where moves of nodes to new partitions, when considered in isolation, would decrease the edge cut but, when considered *en masse*, actually increase the overall edge cut.

ParMetis also implements optimizations specific to the message-passing programming model. It coalesces small messages into larger messages, and it detects situations where the graph can be redistributed to a smaller set of processes.

3 Amorphous Data-parallelism

Amorphous data-parallelism is a form of parallelism that arises in irregular algorithms that operate on complex data structures like graphs [12]. At each point during the execution of such an algorithm, there are certain nodes or edges in the graph where computation might be performed. Performing a computation may require reading or writing other nodes and edges in the graph. The node or edge on which a computation is centered is called an *active element*, and the computation itself is called an *activity*. It is convenient to think of an activity as resulting from the application of an *operator* to the active node. We refer to the set of nodes and edges that are read or written in performing the activity as the *neighborhood* of that activity. Note that in general, the neighborhood of an active node is distinct from the set of its neighbors in the graph. Activities may modify the graph structure of the neighborhood by adding or removing graph elements.

In general, there are many active nodes in a graph, so a sequential implementation must pick one of them and perform the appropriate computation. In some algorithms such as Metis, the implementation is allowed to pick *any* active node for execution. We call these algorithms *unordered algorithms*. In contrast, other algorithms dictate an order in which active nodes must be processed. We call these *ordered algorithms*.

A natural way to program these algorithms is to use the Galois programming model [10], which is a *sequential*, object-oriented programming model (such as Java) augmented with two *Galois set iterators*:

- **Unordered-set iterator: foreach (e : Set S) { B(e) }**
  The loop body B(e) is executed for each element e of set S. The order in which iterations execute is indeterminate and can be chosen by the implementation. There may be dependences between the iterations. When an iteration executes, it may add elements to S.
- **Ordered-set iterator: foreach (e : OrderedSet S) { B(e) }**
  This construct iterates over an ordered set S. It is similar to the unordered set iterator above, except that a sequential implementation must choose a minimal element from set S at every iteration. When an iteration executes, it may add new elements to S.

Opportunities for exploiting parallelism arise if there are many active elements at some point in the computation, each one is a site where a processor can perform computation. When active nodes are unordered, multiple active nodes may be processed concurrently as long as their neighborhoods do not overlap. For ordered active elements, there is an additional constraint that activities must appear to commit in the same order as the ordering on the set elements.

Definition 1. *Given a set of active nodes and an ordering on active nodes,* amorphous data-parallelism *is the parallelism that arises from simultaneously processing active nodes, subject to neighborhood and ordering constraints.*

Amorphous data-parallelism is a generalization of conventional data-parallelism in which (1) concurrent operations may conflict with each other, (2) activities can be created dynamically, and (3) activities may modify the underlying data structure.

4 The Galois System

The *Galois system* is a set of data structures and a runtime system for Java that allows programs to exploit amorphous data-parallelism. The runtime system uses an optimistic parallelization scheme, and the data structures implement the necessary features for optimistic execution: conflict detection and rollback.

Data structure library. The system provides a library of concurrent implementations of data structures, such as graphs, maps, and sets, which are commonly used in irregular algorithms. Programmers can either use one of the existing implementations or provide new ones. For a data structure implementation to be suitable for the Galois system it must satisfy three properties: (1) operations on the data structure must appear to execute atomically, (2) it should enforce the appropriate neighborhood constraints, and (3) it should enable rollback in case of conflicts.

Execution Model. The data structures are stored in shared-memory, and active nodes are processed by some number of threads. A free thread picks an arbitrary active node and speculatively applies the operator to that node. Each data structure ensures that its neighborhood constraints are respected. Note that this is performed by the library code not the application code. If a neighborhood constraint is violated, a conflict is reported to the runtime system, which rolls back one of the conflicting activities. To enable rollback, each library method that modifies a data structure makes a copy of the data before modification. Like lock manipulation, rollbacks are a service implemented by the library and runtime system.

Runtime System. The Galois runtime system coordinates the parallel execution of the application. A `foreach` construct is executed by some number of threads. Each thread works on an active node from the Galois iterator and executes speculatively, rolling back the activity if needed. Library code registers with the runtime to ensure that neighborhood conflicts and rollbacks are implemented correctly.

5 GMetis

In this section, we show how the Galois system can be used to parallelize Metis. Parallelization proceeds in three steps: (1) we identify instances of amorphous data-parallelism, (2) we modify those instances to use a Galois set iterator, and (3) we modify the algorithm to use the graph data structure from the Galois library. Once we identify the amorphous data-parallelism loops, the subsequent steps (2–3) are straightforward.

Coarsening. The heavy edge matching algorithm is amorphous data-parallel. All the graph nodes are active nodes, and the neighborhood of an active node consists of itself and its direct neighbors in the graph. Nodes can be processed in any order, so this is an unordered algorithm.

Creating a coarser graph is also amorphous data-parallel. All the graph nodes in the finer graph are the active elements, and nodes in the coarser graph can be created in any order. When processing an active node n, an activity will access the neighbors of n and the neighbors of the node with which n matches. Edges are added between this set of nodes and the corresponding set in the coarser graph. This entire set of elements is the neighborhood of an activity.

Initial Partitioning. This phase generally accounts for only a small fraction of the overall runtime of Metis, so we did not investigate parallelizing it.

Refinement. Random k-way refinement is amorphous data-parallel. The boundary nodes can be processed in any order. They are the active nodes. When moving a node n to a neighbor partition, the partitioning information of the direct neighbors of n has to be updated. Thus, the neighborhood of each active node consists of these direct neighbors.

5.1 Optimizations

Graph Representation. The graphs in the Galois library are object-based adjacency lists. The graph keeps a list of node objects, and each node object keeps a list of its neighbors. However, this implementation is very costly compared to the compressed sparse row (CSR) format based on arrays used in Metis (see Figure 4). For instance, a serial version of Metis using Galois and using object-based adjacency lists is about an order of magnitude slower than the standard implementation of Metis written in C (see Section 6). Note that this includes the overhead of Java over C. This slowdown has nothing to do with parallelism *per se* but rather is a sequential optimization that is difficult to perform starting from the Galois library of graph implementations. The API of the CSR graph is incompatible with the more general graph API used by the library.

We have developed a variant of the Galois parallelization of Metis, which differs only in that it has been modified by hand to use a CSR graph. It is this variant that will be our main point of comparison in Section 6.

One-shot. Each one of the instances of amorphous data-parallelism identified above benefits from the one-shot optimization [11]. Briefly, if the neighborhood of an activity can be determined before executing it, then the neighborhood constraints of the activity can evaluated eagerly. This provides three benefits: (1) no rollback information needs to be stored during the execution of the activity because the activity is guaranteed to complete after the initial check, (2) the cost of an aborted activity is less because conflicts are found earlier, and (3) there are no redundant checks of neighborhood constraints because all the constraints are checked at once.

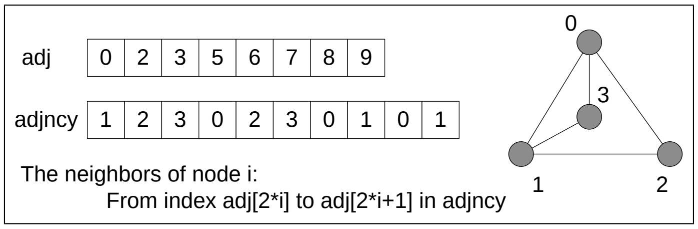


Fig. 4. The graph data structure of GHMetis is a variation of the compressed sparse row (CSR) format that allows creating the graph in parallel because the maximum degree of a node is specified beforehand.

6 Evaluation

6.1 Methodology

To evaluate the impact of exploiting amorphous data-parallelism in graph partitioning, we implemented a version of Metis written in Java and using the Galois system to exploit amorphous data-parallelism (GMetis) and a version of GMetis that additionally implements the data structure optimization mentioned in Section 5.1 by hand (GHMetis). We compared the performance of our implementations with two publicly available graph partitioners: the sequential Metis program and ParMetis, a MPI parallelization of Metis by the same authors.

Figure 5 summarizes the graph partitioners that we evaluated. As we described before, Metis and ParMetis have the same algorithm framework, but they differ in (1) heuristics, for example, ParMetis gives priority to internal nodes owned by a process; (2) parameter values, such as the coarsening threshold; and (3) initial partitioning algorithm. We configured Metis to use the heavy edge matching (HEM) and random k-way refinement (KWAYRANDOM) options. This makes the implementation similar to the algorithm described in [7]. GMetis is adapted directly from Metis (with the same heuristics and parameter values) but (1) with a general-purpose graph implementation, (2) with a different algorithm to randomize visiting nodes, and (3) written in Java. GHMetis is a modification of GMetis that replaces the general-purpose graph implementation with the CSR representation described in Section 5.1.

We conducted two sets of experiments. A small-scale experiment with all four partitioners, and a large-scale experiment with only Metis, ParMetis, and GHMetis. For all experiments, we partitioned the input graph into 64 partitions. We transformed the input graphs or sparse matrices into suitable inputs to graph partitioning by making them symmetric with unit edge weights and removing all self edges.

For the small-scale experiment, we selected three graphs of road networks from the University of Florida Sparse Matrix Collection [2] and from the DIMACS shortest path competition [3]. For the large-scale experiment, we chose the 15 inputs from the University of Florida collection with the largest number of edges (number of non-zeros) whose fraction of edges to nodes was less than 20 (to select sparse matrices). The choice of cutoff was arbitrary, but, generally, more dense matrices would not benefit from exploiting amorphous data-parallelism because the number of conflicts would be high. In cases where there were multiple matrices from the same problem family, we selected only the largest input of the family except for the wikipedia family of inputs where we selected the smallest input because we had trouble running the largest input with Metis. Figure 6 shows the matrices selected.

We ran all the implementations on the same test machine: a Sun Fire X2270 running Ubuntu Linux 8.04.4 LTS 64-bit. The machine contains two quad-core 2.93 GHz Intel Xeon X5570 processors. The two CPUs share 24 GB of main memory. Each core has a 32 KB L1 cache and a unified 256 KB L2 cache. Each processor has an 8 MB L3 cache that is shared among the cores.

We used Metis 5.0pre2 and ParMetis 3.1.1 compiled with gcc 4.2.4 and with the default 64-bit build options. For both programs, we configured the graph index data type to be a 64-bit integer as well. With ParMetis, we used OpenMPI 1.4.2, and multiple MPI processes were launched on the same machine. For GMetis and GHMetis, we used the Sun JDK 1.6.0 to compile and ran the programs with a heap size of 20 GB. To control for JIT compilation, we ran each input 5 times within the same JVM instance and report the median run time.

	Language	Parallelization	Graph Data Structure	Adapted From
Metis	C		CSR	-
ParMetis	C	MPI	Distributed CSR	-
GMetis	Java	Galois	Object-based adjacency list	Metis
GHMetis	Java	Galois	CSR	GMetis

Fig. 5. Summary of graph partitioning algorithms evaluated.

6.2 Results

Figures 7 and 8 show the results from the small-scale experiment. The results are typical of many of the trends we see in the large-scale experiment as well. We define scalability as the runtime relative to the single-threaded runtime of the same program. Speedup is the runtime relative to the runtime of Metis.

For single-threaded runs, GMetis is about five times slower than GHMetis, and GHMetis is about twice as slow as Metis and ParMetis. The scalability of ParMetis, GMetis and GHMetis is similar for the smaller inputs, roadNet-CA and roadNet-TX, but for USA-road-d.W, GMetis and GHMetis have better scalability than ParMetis. In the large-scale experiments, the scalability gap between GHMetis and ParMetis becomes more pronounced. For USA-road-d.W, ParMetis produces worse partitions in terms of balance and edge cut than the other three partitioners. We believe this is due to the different initial partitioning phase. Also, ParMetis uses a different partitioning strategy when run with one process than with more than one process. This may explain the much larger edge cut in the one process case. ParMetis has better speedup than GHMetis largely due to starting with a better single-threaded runtime. Recall that GHMetis is implemented in Java whereas ParMetis and Metis are implemented in C.

Figures 9 and 10 show the results from the large-scale experiment. Instead of showing results for each number of threads/processes, we only show the best performing result for ParMetis and GHMetis and its corresponding edge cut and balance. The trends from the small-scale experiment show up here as well. GHMetis achieves better scalability and produces better partitions than ParMetis, but ParMetis is faster than GHMetis. In fact, it is often faster than Metis as well. Observe that the speedup of ParMetis is greater than its scalability. Over a large set of inputs, we see that GHMetis scales better than ParMetis, suggesting that amorphous data-parallelism is a fruitful form of parallelism to exploit.

The missing runs for GHMetis are generally due to lack of memory. They occur on larger inputs when GHMetis spends most of its time doing garbage collection. For inputs as-Skitter, rel9 and Ruccil, Metis performs particularly poorly compared to ParMetis or GHMetis. We believe that this is due to the randomization strategy used in Metis, which causes the coarsening phase to stop early and consequentially produces a very large input to the initial partitioning phase. When we used the same randomization strategy in GHMetis, we observed similarly poor performance.

7 Conclusion

Graph partitioning is an important problem in parallel computing, and we have shown how one common graph partitioning application, Metis, naturally exhibits a form of parallelism that we call amorphous data-parallelism. Using the Galois system, we can exploit this parallelism to achieve reasonable parallel scalability from a sequential specification, and this scalability is comparable to that of an explicitly parallel implementation over a suite of large test matrices. An

	$\lvert V\rvert$	$\lvert E\rvert$	$\lvert E\rvert/\lvert V\rvert$	Description
roadNet-CA	1,965,206	2,766,607	1.41	Road network of California
roadNet-TX	1,379,917	1,921,660	1.39	Road network of Texas
USA-road-d.W	6,262,104	7,559,642	1.21	Road network of western USA
as-Skitter	1,696,415	11,095,298	6.54	Internet topology graph
cage15	5,154,859	47,022,346	9.12	DNA electrophoresis
circuit5M_dc	3,523,315	8,562,474	2.43	Large circuit, DC analysis
cit-Patents	3,774,768	16,518,947	4.38	Citation network among US patents
Freescale1	3,428,754	8,472,832	2.47	Circuit problem
GL7d19	1,955,309	37,322,139	19.09	Differentials of Voronoi complex
kkt_power	2,063,494	6,482,320	3.14	Nonlinear optimization
memchip	2,707,524	6,621,370	2.45	Memory chip
patents	3,750,822	14,970,766	3.99	NBER US patent citations
rajat31	4,690,002	7,813,751	1.67	Circuit simulation matrix
rel9	5,921,786	23,667,162	4.00	Relations
relat9	9,746,232	38,955,401	4.00	Relations
Rucci1	1,977,885	7,791,154	3.94	Ill-conditioned least-squares problem
soc-LiveJournal1	4,846,609	42,851,237	8.84	LiveJournal online social network
wikipedia-20051105	1,598,534	18,540,603	11.60	Link graph of Wikipedia pages

Fig. 6. Summary of inputs used in evaluation. The top portion lists the small-scale inputs; the bottom portion lists the large-scale inputs. All inputs are from [2] except USA-road-d.W, which is from [3].

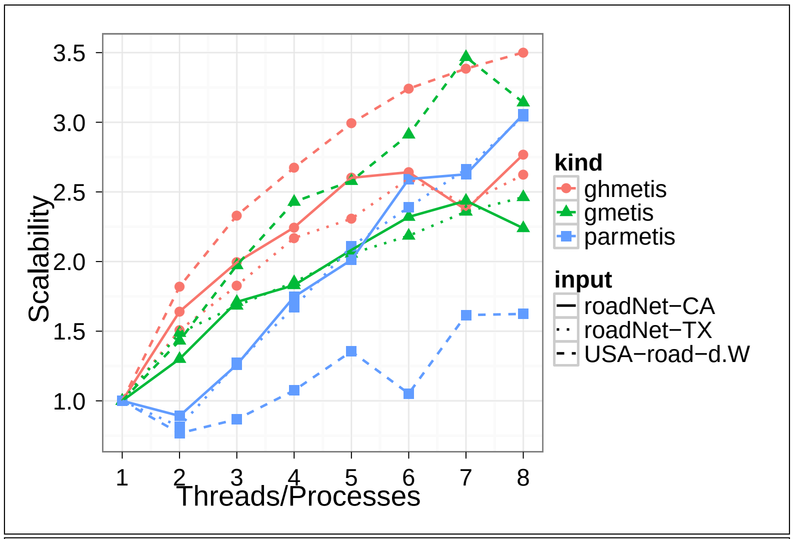
advantage of the Galois version is that it is derived directly from the sequential application.

Our naïve implementation still does not obtain consistent speedup over sequential Metis, but we have shown how changing the graph data structure bridges the gap considerably. In addition, we have not parallelized the initial partitioning phase of the algorithm. We also believe that a significant overhead exists because our implementation is in Java, whereas Metis is hand-tuned C.

The previous approaches to parallelizing graph partitioning [1, 6, 13] are complementary to our approach. On a hybrid architecture consisting of multiple multicore machines, amorphous data-parallelism can exploit intra-machine parallelism while message-passing can exploit inter-machine parallelism.

References

1. C. Chevalier and F. Pellegrini. Pt-scotch: A tool for efficient parallel graph ordering. *Parallel Computing*, 34(6–8):318–331, 2008.
2. T. A. Davis and Y. F. Hu. The university of florida sparse matrix collection (in submission). *ACM Transactions on Mathematical Software*, 2010.
3. DIMACS. 9th dimacs implementation challenge—shortest paths, 2005. `http://www.dis.uniroma1.it/~challenge9`.
4. A. George. Nested dissection of a regular finite element mesh. *SIAM Journal on Numerical Analysis*, 10(2):345–363, 1973.



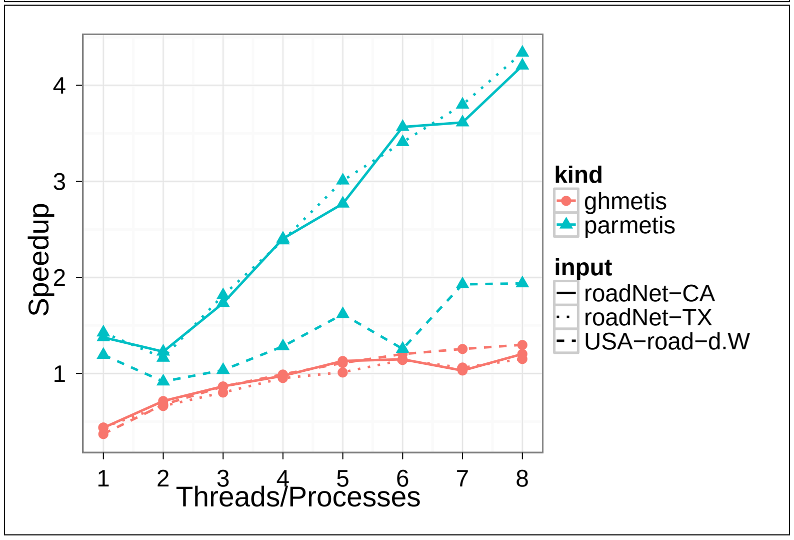


Fig. 7. Scalability and speedup of ParMetis, GMetis and GHMetis. Scalability is runtime relative to runtime with one thread/process. Speedup is runtime relative to runtime of sequential Metis.

		ParMetis			GMetis			GHMetis		
	T	Time	Cut	Bal.	Time	Cut	Bal.	Time	Cut	Bal.
roadNet-CA	1	1,195	5,752	1.03	16885	9886	1.03	3785	5525	1.02
	2	1,340	6,455	1.04	12982	9495	1.03	2307	5882	1.02
	4	684	6,341	1.04	9226	9401	1.02	1686	5605	1.02
	8	391	6,472	1.03	7540	9707	1.03	1367	5865	1.03
roadNet-TX	1	791	4,426	1.03	12067	4760	1.03	2570	4592	1.02
	2	970	4,715	1.05	8114	4443	1.02	1706	4237	1.03
	4	473	4,705	1.05	6517	4433	1.03	1185	4165	1.02
	8	260	4,611	1.04	4901	4329	1.02	980	4232	1.02
USA-road-d.W	1	4,781	11,012	1.18	68151	3057	1.01	15384	2930	1.02
	2	6,230	6,382	1.23	47598	3007	1.01	8457	2951	1.00
	4	4,449	5,868	1.22	28064	2951	1.00	5754	2971	1.01
	8	2,944	5,455	1.22	21691	3050	1.00	4394	3175	1.01

Fig. 8. Time, edge cut and balance of ParMetis, GMetis and GHMetis as a function of input and number of threads. All times are in milliseconds. Metis results (Time, Cut, Balance) for roadNet-CA, roadNet-TX and USA-road.d.W are (1644, 6010, 1.02), (1128, 4493, 1.02), and (5704, 3113, 1.01) respectively.

	Metis	Best ParMetis		Best GHMetis	
	Time	m/t	t1/t	m/t	t1/t
as-Skitter	273,581			14.86	2.50
cage15	23,677	2.86	2.31	1.04	3.16
circuit5M_dc	3,164	1.63	1.38	1.26	3.20
cit-Patents	58,740	4.34	1.62	2.22	2.93
Freescale1	2,944	1.90	1.60	1.22	3.61
GL7d19	168,199	4.51	6.64	2.43	2.50
kkt_power	10,445	3.80	1.34	1.22	2.91
memchip	2,368	2.14	1.84	1.35	3.27
patents	51,695	4.10	1.65	2.08	2.91
rajat31	4,044	4.27	3.49	1.35	3.37
rel9			1.13		
relat9	1,106,377	2.63	1.06		
Ruccil	1,065,551	20.26	1.03	48.45	3.15
soc-LiveJournal1	304.295				
wikipedia-20051105	358,030			8.98	2.54
Geomean		**3.60**	**1.77**	**2.80**	**2.98**

Fig. 9. Performance of Metis, ParMetis and GHMetis. m/t is speedup, runtime relative to sequential Metis (m). t/t1 is scalability, runtime relative to single-threaded runtime. All times are in milliseconds. Blank entries correspond to runs that terminated abnormally or exceeded the timeout of 30 minutes.

	Metis		Best ParMetis		Best GHMetis	
	Cut	Bal.	Cut	Bal.	Cut	Bal.
as-Skitter	3,054,856	1.03			1,991,020	1.03
cage15	4,536,885	1.03	4,697,417	1.05	4,629,593	1.03
circuit5M_dc	13,187	1.03	14,764	1.05	14,133	1.02
cit-Patents	3,036,598	1.02	3,258,823	1.05	2,900,222	1.02
Freescale1	13,429	1.03	15,093	1.05	13,828	1.02
GL7d19	31,168,010	1.03	34,248,358	1.26	31,295,495	1.03
kkt_power	453,357	1.02	578,264	1.04	392,115	1.01
memchip	16,235	1.02	18,524	1.05	16,534	1.02
patents	2,672,325	1.02	2,841,655	1.05	2,550,783	1.01
rajat31	27,391	1.01	27,907	1.04	26,851	1.00
rel9			12,774,163	1.05		
relat9	22,417,154	1.03	21,532,960	1.05		
Ruccil	1,890,352	1.03	1,928,212	1.01	1,074,278	1.00
soc-LiveJournal1	13,838,247	1.03				
wikipedia-20051105	10,081,144	1.03			9,389,056	1.03
Geomean		**1.02**		**1.06**		**1.02**

Fig. 10. Balance and EdgeCut of Metis, ParMetis and GHMetis. Blank entries correspond to runs that terminated abnormally or exceeded the timeout of 30 minutes.

5. A. Gupta. An evaluation of parallel graph partitioning and ordering software on a massively parallel computer. Technical Report RC25008 (W1006-029), IBM Research Division, Thomas J. Watson Research Center, 2010.
6. G. Karypis and V. Kumar. A coarse-grain parallel formulation of multilevel k-way graph-partitioning algorithm. In *Proc. 8th SIAM Conference on Parallel Processing for Scientific Computing*, 1997.
7. G. Karypis and V. Kumar. Multilevel k-way partitioning scheme for irregular graphs. *Journal of Parallel and Distributed Computing*, 48(1):96–129, 1998.
8. G. Karypis and V. Kumar. A fast and high quality multilevel scheme for partitioning irregular graphs. *SIAM Journal on Scientific Computing*, 20(1):359–392, 1999.
9. B. W. Kernighan and S. Lin. An effective heuristic procedure for partitioning graphs. *The Bell System Technical Journal*, pages 291–308, Feb. 1970.
10. M. Kulkarni, K. Pingali, B. Walter, G. Ramanarayanan, K. Bala, and L. P. Chew. Optimistic parallelism requires abstractions. *SIGPLAN Not. (Proceedings of PLDI 2007)*, 42(6):211–222, 2007.
11. M. Mendez-Lojo, D. Nguyen, D. Prountzos, X. Sui, M. A. Hassaan, M. Kulkarni, M. Burtscher, and K. Pingali. Structure-driven optimizations for amorphous data-parallel programs. In *Proceedings of the 15th ACM SIGPLAN symposium on Principles and practice of parallel programming*, pages 3–14, 2010.
12. K. Pingali, M. Kulkarni, D. Nguyen, M. Burtscher, M. Mendez-Lojo, D. Prountzos, X. Sui, and Z. Zhong. Amorphous data-parallelism in irregular algorithms. regular tech report TR-09-05, The University of Texas at Austin, 2009.
13. C. Walshaw and M. Cross. Jostle: Parallel multilevel graph-partitioning software—an overview. In F. Magoules, editor, *Mesh Partitioning Techniques and Domain Decomposition Techniques*, pages 27–58. Civil-Comp Ltd., 2007.