Contemporary Mathematics

Parallel Community Detection for Massive Graphs

E. Jason Riedy, Henning Meyerhenke, David Ediger, and David A. Bader

Abstract. Tackling the current volume of graph-structured data requires parallel tools. We extend our work on analyzing such massive graph data with a massively parallel algorithm for community detection that scales to current data sizes, clustering a real-world graph of over 100 million vertices and over 3 billion edges in under 500 seconds on a four-processor Intel E7-8870-based server. Our algorithm achieves moderate parallel scalability without sacrificing sequential operational complexity. Community detection partitions a graph into subgraphs more densely connected within the subgraph than to the rest of the graph. We take an agglomerative approach similar to Clauset, Newman, and Moore's sequential algorithm, merging pairs of connected intermediate subgraphs to optimize different graph properties. Working in parallel opens new approaches to high performance. We improve performance of our parallel community detection algorithm on both the Cray XMT2 and OpenMP platforms and adapt our algorithm to the DIMACS Implementation Challenge data set.

1. Communities in Graphs

Graph-structured data inundates daily electronic life. Its volume outstrips the capabilities of nearly all analysis tools. The Facebook friendship network has over 845 million users [**9**]. Twitter boasts over 140 million new messages each day [**34**], and the NYSE processes over 300 million trades each month [**25**]. Applications of analysis range from database optimization to marketing to regulatory monitoring. Global graph analysis kernels at this scale tax current hardware and software architectures due to the size *and* structure of typical inputs.

One such useful analysis kernel finds smaller communities, subgraphs that locally optimize some connectivity criterion, within these massive graphs. We extend the boundary of current complex graph analysis by presenting the first algorithm for detecting communities that scales to graphs of practical size, over 100 million vertices and over three billion edges in less than 500 seconds on a shared-memory parallel architecture with 256 GiB of memory.

Community detection is a graph clustering problem. There is no single, universally accepted definition of a community within a social network. One popular definition is that a community is a collection of vertices more strongly connected than would occur from random chance, leading to methods based on modularity [**22**].

2010 *Mathematics Subject Classification.* Primary 68R10, 05C85; Secondary 68W10, 68M20.

Another definition [**28**] requires vertices to be more connected to others within the community than those outside, either individually or in aggregate. This aggregate measure leads to minimizing the communities' conductance. We consider disjoint partitioning of a graph into connected communities guided by a local optimization criterion. Beyond obvious visualization applications, a disjoint partitioning applies usefully to classifying related genes by primary use [**36**] and also to simplifying large organizational structures [**18**] and metabolic pathways [**29**]. We report results for maximizing modularity, although our implementation also supports minimizing conductance.

Contributions. We present our previously published parallel agglomerative community detection algorithm, adapt the algorithm for the DIMACS Implementation Challenge, and evaluate its performance on two multi-threaded platforms. Our algorithm scales to practical graph sizes on available multithreaded hardware while keeping the same sequential operation complexity as current state-of-the-art algorithms. Our approach is both natively parallel and simpler than most current sequential community detection algorithms. Also, our algorithm is agnostic towards the specific criterion; any criterion expressible as individual edge scores can be optimized locally with respect to edge contractions. Our implementation supports both maximizing modularity and minimizing conductance.

Capability and performance. On an Intel-based server platform with four 10-core processors and 256 GiB of memory, our algorithm extracts modular communities from the 105 million vertex, 3.3 billion edge `uk-2007-05` graph in under 500 seconds. A 2 TiB Cray XMT2 requires around 2 400 seconds on the same graph. Our edge-list implementation scales in execution time up to 80 OpenMP threads and 64 XMT2 processors on sufficiently large graphs.

Outline. Section 2 presents our high-level algorithm and describes our current optimization criteria. Section 3 discusses implementation and data structure details for our two target threaded platforms. Section 4 considers parallel performance and performance on different graph metrics for two of the DIMACS Implementation Challenge graphs; full results are in the workshop report [**32**]. Section 5 discusses related work, and Section 6 considers future directions.

2. Parallel Agglomerative Community Detection

Agglomerative clustering algorithms begin by placing every input graph vertex within its own unique community. Then neighboring communities are merged to optimize an objective function like maximizing modularity [**22, 2, 21**] (internal connectedness) or minimizing conductance (normalized edge cut) [**1**]. Here we summarize the algorithm and break it into primitive operations. Section 3 then maps each primitive onto our target threaded platforms.

We consider maximizing metrics (without loss of generality) and target a local maximum rather than a global, possibly non-approximable, maximum. There are a wide variety of metrics for community detection [**12**]. We discuss two, modularity and conductance, in Section 2.1.

Our algorithm maintains a *community graph* where every vertex represents a community, edges connect communities when they are neighbors in the input graph, and weights count the number of input graph edges either collapsed into a single community graph edge or contained within a community graph vertex. We currently do not require counting the vertices in each community, but such an extension is straight-forward.

From a high level, our algorithm repeats the following steps until reaching some termination criterion:

(1) associate a score with each edge in the community graph, exiting if no edge has a positive score,
(2) greedily compute a weighted maximal matching using those scores, and
(3) contract matched communities into a new community graph.

Each step serves as our primitive parallel operations.

The first step scores edges by how much the optimization metric would change if the two adjacent communities merge. Computing the change in modularity and conductance requires only the weight of the edge and the weight of the edge's adjacent communities. The change in conductance is negated to convert minimization into maximization.

The second step, a greedy approximately maximum weight maximal matching, selects pairs of neighboring communities where merging them will improve the community metric. The pairs are independent; a community appears at most once in the matching. Properties of the greedy algorithm guarantee that the matching's weight is within a factor of two of the maximum possible value [**27**]. Any positive-weight matching suffices for optimizing community metrics. Some community metrics, including modularity [**6**], form NP-complete optimization problems. Additional work computing our heuristic by improving the matching may not produce better results. Our approach follows existing parallel algorithms [**15, 20**]. Differences appear in mapping the matching algorithm to our data structures and platforms.

The final step contracts the community graph according to the matching. This contraction primitive requires the bulk of the time even though there is little computation. The impact of the intermediate data structure on improving multithreaded performance is explained in Section 3.

Termination occurs either when the algorithm finds a local maximum or according to external constraints. If no edge score is positive, no contraction increases the objective, and the algorithm terminates at a local maximum. In our experiments with modularity, our algorithm frequently assigns a single community per connected component, a useless local maximum. Real applications will impose additional constraints like a minimum number of communities or maximum community size. Following the DIMACS Implementation Challenge rules [**3**], Section 4's performance experiments terminate once at least half the initial graph's edges are contained within the communities, a coverage ≥ 0.5.

Assuming all edges are scored in a total of $O(|E_c|)$ operations and some heavy weight maximal matching is computed in $O(|E_c|)$ [**27**] where E_c is the edge set of the current community graph, each iteration of our algorithm's loop requires $O(|E|)$ operations. As with other algorithms, the total operation count depends on the community growth rates. If our algorithm halts after K contraction phases, our algorithm runs in $O(|E| \cdot K)$ operations where the number of edges in the original graph, $|E|$, bounds the number of edges in any community graph. If the community graph is halved with each iteration, our algorithm requires $O(|E| \cdot \log |V|)$ operations, where $|V|$ is the number of vertices in the input graph. If the graph is a star, only two vertices are contracted per step and our algorithm requires $O(|E| \cdot |V|)$ operations. This matches experience with the sequential CNM algorithm [**35**].

2.1. Local optimization metrics. We score edges for contraction by modularity, an estimate of a community's deviation from random chance [**22, 2**], or conductance, a normalized edge cut [**1**]. We maximize modularity by choosing the largest independent changes from the current graph to the new graph by one of

two heuristics explained below. Minimization measures like conductance involve maximizing changes' negations.

Modularity. Newman [**21**]'s modularity metric compares the connectivity within a collection of vertices to the expected connectivity of a random graph with the same degree distribution. Let m be the number of edges in an undirected graph $G = G(V, E)$ with vertex set V and edge set E. Let $S \subset V$ induce a graph $G_S = G(S, E_S)$ with $E_S \subset E$ containing only edges where both endpoints are in S. Let m_S be the number of edges $|E_S|$, and let $\overline{m_S}$ be an expected number of edges in S given some statistical background model. Define the modularity of the community induced by S as $Q_S = \frac{1}{m}(m_S - \overline{m_S})$. Modularity represents the deviation of connectivity in the community induced by S from an expected background model. Given a partition $V = S_1 \cup S_2 \cup \cdots \cup S_k$, the modularity of that partitioning is $Q = \sum_{i=1}^{k} Q_{S_i}$.

Newman [**21**] considers the specific background model of a random graph with the same degree distribution as G where edges are independently and identically distributed. If x_S is the total number of edges in G where either endpoint is in S, then we have $Q_S = (m_S - x_S^2/4m)/m$ as in [**2**]. A subset S is considered a module when there are more internal edges than expected, $Q_S > 0$. The m_S term encourages forming large modules, while the x_S term penalizes modules with excess external edges. Maximizing Q_S finds communities with more internal connections than external ones. Expressed in matrix terms, optimizing modularity is a quadratic integer program and is an NP-complete optimization problem [**6**]. We compute a local maximum and not a global maximum. Different operation orders produce different locally optimal points.

Section 3's implementation scores edges by the change in modularity contracting that one edge would produce, analogous to the sequential CNM algorithm. Merging the vertex U into a disjoint set of vertices $W \in C$, requires that the change $\Delta Q(W, U) = Q_{W \cup U} - (Q_W + Q_U) > 0$. Expanding the expression for modularity,

$$\begin{aligned} m \cdot \Delta Q(W,U) &= m\left(Q_{W\cup U} - (Q_W + Q_U)\right) \\ &= (m_{W\cup U} - (m_W + m_U) - \\ &\qquad (\overline{m_{W\cup U}} - (\overline{m_W} + \overline{m_U})) \\ &= m_{W\leftrightarrow U} - \overline{(m_{W\cup U} - (m_W + m_U))}, \end{aligned}$$

where $m_{W \leftrightarrow U}$ is the number of edges between vertices in sets W and U. Assuming the edges are independent and identically distributed across vertices respecting their degrees [**8**],

$$\overline{(m_{W\cup U} - (m_W + m_U))} = m \cdot \frac{x_W}{2m} \cdot \frac{x_U}{2m}, \text{ and}$$

$$\Delta Q(W,U) = \frac{m_{W\leftrightarrow U}}{m} - \frac{x_W}{2m} \cdot \frac{x_U}{2m}. \tag{1}$$

We track $m_{W \leftrightarrow U}$ and x_W in the contracted graph's edge and vertex weights, respectively. The quantity x_W equals the sum of W's degrees or the volume of W. If we represent the graph G by an adjacency matrix A, then ΔQ is the rank-one update $A/m - (v/2m) \cdot (v/2m)^T$ restricted to non-zero, off-diagonal entries of A. The data necessary for computing the score of edge $\{i, j\}$ are $A(i, j)$, $v(i)$, and $v(j)$, similar in spirit to a rank-one sparse matrix-vector update.

Modularity can be defined slightly differently depending on whether you double-count edges within a community by treating an undirected graph as a directed graph

with edges in both directions. The DIMACS Implementation Challenge uses this variation, and we have included an option to double-count edges.

Modularity has known limitations. Fortunato and Barthélemy [**11**] demonstrate that global modularity optimization cannot distinguish between a single community and a group of smaller communities. Berry *et al.* [**4**] provide a weighting mechanism that overcomes this resolution limit. Instead of this weighting, we compare CNM with the modularity-normalizing method of McCloskey and Bader [**2**]. Lancichinetti and Fortunato [**17**] show that multiresolution modularity can still have problems, e. g., merging small clusters and splitting large ones.

McCloskey and Bader's algorithm (MB) only merges vertices into the community when the change is deemed statistically significant against a simple statistical model assuming independence between edges. The sequential MB algorithm computes the mean $\overline{\Delta Q(W,:)}$ and standard deviation $\sigma(\Delta Q(W,:))$ of all changes adjacent to community W. Rather than requiring only $\Delta Q(W,U) > 0$, MB requires a tunable level of statistical significance with $\Delta Q(W,U) > \overline{\Delta Q(W,:)} + k \cdot \sigma(\Delta Q(W,:))$. Section 4 sets $k = -1.5$. Sequentially, MB considers only edges adjacent to the vertex under consideration and tracks a history for wider perspective. Because we evaluate merges adjacent to all communities at once by matching, we instead filter against the threshold computed across all current potential merges.

Conductance. Another metric, graph conductance, measures a normalized cut between a graph induced by vertex set S and the graph induced by the remaining vertices $V \setminus S$. Denote the cut induced by a vertex set $S \subset V$ by

$$\partial(S) = \{\{u,v\} | \{u,v\} \in E, u \in S, v \notin S\},$$

and the size of the cut by $|\partial(S)|$. Then the conductance of S is defined [**1**] as

$$\phi(S) = \frac{|\partial(S)|}{\min\{\mathrm{Vol}(S), \mathrm{Vol}(V \setminus S)\}}. \tag{2}$$

If $S = V$ or $S = \emptyset$, let $\phi(S) = 1$, the largest obtainable value.

The minimum conductance over all vertex sets $S \subset V$ is the graph's conductance. Finding a subset with small modularity implies a bottleneck between the subset's induced subgraph and the remainder. Random walks will tend to stay in the induced subgraph and converge rapidly to their stationary distribution [**5**]. Given a partition $V = S_1 \cup S_2 \cup \cdots \cup S_k$, we evaluate the conductance of that partitioning as $\sum_{i=1}^{k} \phi(S_i)$.

We score an edge $\{i,j\}$ by the negation of the change from old to new, or $\phi(S_i) + \phi(S_j) - \phi(S_i \cup S_j)$. We again track the edge multiplicity in the edge weight and the volume of the subgraph in the vertex weight.

3. Mapping the Agglomerative Algorithm to Threaded Platforms

Our implementation targets two multithreaded programming environments, the Cray XMT [**16**] and OpenMP [**26**], both based on the C language. Both provide a flat, shared-memory view of data but differ in how they manage parallelism. However, both environments intend that ignoring the parallel directives produces correct although sequential C code. The Cray XMT environment focuses on implicit, automatic parallelism, while OpenMP requires explicit management.

The Cray XMT architecture tolerates high memory latencies from physically distributed memory using massive multithreading. There is no cache in the processors; all latency is handled by threading. Programmers do not directly control the threading but work through the compiler's automatic parallelization with occasional

pragmas providing hints to the compiler. There are no explicit parallel regions. Threads are assumed to be plentiful and fast to create. Current XMT and XMT2 hardware supports over 100 hardware thread contexts per processor. Unique to the Cray XMT are full/empty bits on every 64-bit word of memory. A thread reading from a location marked empty blocks until the location is marked full, permitting very fine-grained synchronization amortized over the cost of memory access. The full/empty bits permit automatic parallelization of a wider variety of data dependent loops. The Cray XMT provides one additional form of parallel structure, futures, but we do not use them here.

The widely-supported OpenMP industry standard provides more traditional, programmer-managed threading. Parallel regions are annotated explicitly through compiler pragmas. Every loop within a parallel region must be annotated as a work-sharing loop or else every thread will run the entire loop. OpenMP supplies a lock data type which must be allocated and managed separately from reading or writing the potentially locked memory. OpenMP also supports tasks and methods for interaction, but our algorithm does not require them.

3.1. Graph representation. We use the same core data structure as our earlier work [**31, 30**] and represent a weighted, undirected graph with an array of triples (i, j, w) for edges between vertices i and j with $i \neq j$. We accumulate repeated edges by adding their weights. The sum of weights for self-loops, $i = j$, are stored in a $|V|$-long array. To save space, we store each edge *only once*, similar to storing only one triangle of a symmetric matrix.

Unlike our initial work, however, the array of triples is kept in buckets defined by the first index i, and we hash the order of i and j rather than storing the strictly lower triangle. If i and j both are even or odd, then the indices are stored such that $i < j$, otherwise $i > j$. This scatters the edges associated with high-degree vertices across different source vertex buckets.

The buckets need not be sequential. We store both beginning and ending indices into the edge array for each vertex. In a traditional sparse matrix compressed format, the entries adjacent to vertex $i + 1$ would follow those adjacent to i. Permitting the buckets to separate reduces synchronization within graph contraction. We store both i and j for easy parallelization across the entire edge array. Because edges are stored only once, edge $\{i, j\}$ could appear in the bucket for either i or j but not both.

A graph with $|V|$ vertices and $|E|$ non-self, unique edges requires space for $3|V| + 3|E|$ 64-bit integers plus a few additional scalars to store $|V|$, $|E|$, and other book-keeping data.

3.2. Scoring and matching. Each edge's score is an independent calculation for our metrics. An edge $\{i, j\}$ requires its weight, the self-loop weight for i and j, and the total weight of the graph. Parallel computation of the scores is straightforward, and we store the edge scores in an $|E|$-long array of 64-bit floating point data.

Computing the heavy maximal matching is less straight-forward. We repeatedly sweep across the vertices and find the best adjacent match until all vertices are either matched or have no potential matches. The algorithm is non-deterministic when run in parallel. Different executions on the same data may produce different matchings. This does not affect correctness but may lead to different communities.

Our earlier implementation iterated in parallel across all of the graph's edges on each sweep and relied heavily on the Cray XMT's full/empty bits for synchronization

of the best match for each vertex. This produced frequent hot spots, memory locations of high contention, but worked sufficiently well with nearly no programming effort. The hot spots crippled an explicitly locking OpenMP implementation of the same algorithm on Intel-based platforms.

We have updated the matching to maintain an array of currently unmatched vertices. We parallelize across that array, searching each unmatched vertex u's bucket of adjacent edges for the highest-scored unmatched neighbor, v. Once each unmatched vertex u finds its best current match, the vertex checks if the other side v (also unmatched) has a better match. If the current vertex u's choice is better, it claims both sides using locks or full/empty bits to maintain consistency. Another pass across the unmatched vertex list checks if the claims succeeded. If not and there was some unmatched neighbor, the vertex u remains on the list for another pass. At the end of all passes, the matching will be maximal. Strictly this is not an $O(|E|)$ algorithm, but the number of passes is small enough in social network graphs that it runs in effectively $O(|E|)$ time.

If edge $\{i, j\}$ dominates the scores adjacent to i and j, that edge will be found by one of the two vertices. The algorithm is equivalent to a different ordering of existing parallel algorithms [**15, 20**] and also produces a maximal matching with weight (total score) within a factor of 0.5 of the maximum.

Social networks often follow a power-law distribution of vertex degrees. The few high-degree vertices may have large adjacent edge buckets, and not iterating across the bucket in parallel may decrease performance. However, neither the Cray XMT nor OpenMP implementations currently support efficiently composing general, nested, light-weight parallel loops. Rather than trying to separate out the high-degree lists, we scatter the edges according to the graph representation's hashing. This appears sufficient for high performance in our experiments.

Our improved matching's performance gains over our original method are marginal on the Cray XMT but drastic on Intel-based platforms using OpenMP. Scoring and matching together require $|E| + 4|V|$ 64-bit integers plus an additional $|V|$ locks on OpenMP platforms.

3.3. Graph contraction. Contracting the agglomerated community graph requires from 40% to 80% of the execution time. Our previous implementation was relatively efficient on the Cray XMT but infeasible on OpenMP platforms. We use the bucketing method to avoid locking and improve performance for both platforms.

Our current implementation relabels the vertex endpoints and re-orders their storage according to the hashing. We then roughly bucket sort by the first stored vertex in each edge. If a stored edge is $(i, j; w)$, we place $(j; w)$ into a bucket associated with vertex i but leave i implicitly defined by the bucket. Within each bucket, we sort by j and accumulate identical edges, shortening the bucket. The buckets then are copied back out into the original graph's storage, filling in the i values. This requires $|V| + 1 + 2|E|$ storage, more than our original implementation, but permits much faster operation on both the XMT2 and Intel-based platforms.

Because the buckets need not be stored contiguously in increasing vertex order, the bucketing and copying do not need to synchronize beyond an atomic fetch-and-add. Storing the buckets contiguously requires synchronizing on a prefix sum to compute bucket offsets. We have not timed the difference, but the technique is interesting.

3.4. DIMACS adjustments. Our original implementation uses 64-bit integers to store vertex labels. All of the graphs in the DIMACS Implementation

Challenge, however, require only 32-bit integer labels. Halving the space required for vertex labels fits the total size necessary for the largest challenge graph, `uk-2007-05`, in less than 200 GiB of RAM. Note that indices into the edge list must remain 64-bit integers. We also keep the edge scores in 64-bit binary floating-point, although only 32-bit floating-point may suffice.

Surprisingly, we found no significant performance difference between 32-bit and 64-bit integers on smaller graphs. The smaller integers should decrease the bandwidth requirement but not the number of memory operations. We conjecture our performance is limited by the latter.

The Cray XMT's full/empty memory operations work on 64-bit quantities, so our Cray XMT2 implementation uses 64-bit integers throughout. This is not a significant concern with 2TiB of memory.

4. Parallel Performance

We evaluate parallel performance on two different threaded hardware architectures, the Cray XMT2 and an Intel-based server. We highlight two graphs, one real and one artificial, from the Implementation Challenge to demonstrate scaling and investigate performance properties. Each experiment is run three times to capture some of the variability in platforms and in our non-deterministic algorithm. Our current implementation achieves speed-ups of up to 13× on a four processor, 40-physical-core Intel-based platform. The Cray XMT2 single-processor times are too slow to evaluate speed-ups on that platform.

4.1. Evaluation platforms. The next generation Cray XMT2 is located at the Swiss National Supercomputing Centre (CSCS). Its 64 processors run at 500 MHz and support four times the memory density of the Cray XMT for a total of 2 TiB. These 64 processors support over 6 400 hardware thread contexts. The improvements over the XMT also include additional memory bandwidth within a node, but exact specifications are not yet officially available.

The Intel-based server platform is located at Georgia Tech. It has four ten-core Intel Xeon E7-8870 processors running at 2.40GHz with 30MiB of L3 cache per processor. The processors support HyperThreading, so the 40 physical cores appear as 80 logical cores. This server, mirasol, is ranked #17 in the November 2011 Graph 500 list and is equipped with 256 GiB of 1 067 MHz DDR3 RAM.

Note that the Cray XMT allocates entire processors, each with at least 100 threads, while the OpenMP platforms allocate individual threads which are mapped to cores. Results are shown per-Cray-XMT processor and per-OpenMP-thread. We run up to the number of physical Cray XMT processors or logical Intel cores. Intel cores are allocated in a round-robin fashion across sockets, then across physical cores, and finally logical cores.

4.2. Test graphs. We evaluate on two DIMACS Implementation Challenge graphs. Excessive single-processor runs on highly utilized resources are discouraged, rendering scaling studies using large graphs difficult. We cannot run the larger graph on a single XMT2 processor within a reasonable time. Table 1 shows the graphs' names and number of vertices and edges. Maximum-thread and -processor timings for the full 30 DIMACS Implementation Challenge graphs are in the workshop report [**32**].

Additionally, we consider execution time on the largest Implementation Challenge graph, `uk-2007-05`. This graph has 105 896 555 vertices and 3 301 876 564 edges.

Graph	$\|V\|$	$\|E\|$
`uk-2002`	18 520 486	261 787 258
`kron_g500-simple-logn20`	1 048 576	44 619 402

TABLE 1. Sizes of graphs used for performance evaluation.

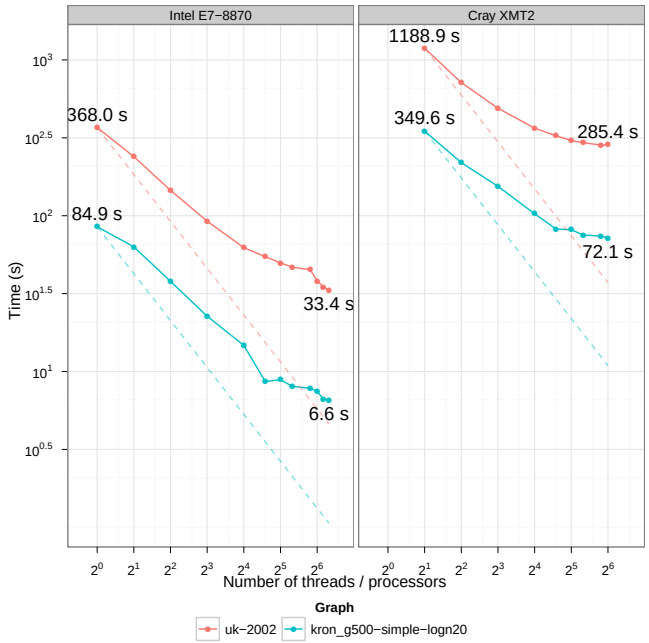

FIGURE 1. Execution time against allocated OpenMP threads or Cray XMT processors per platform and graph. The best single-processor and overall times are noted in the plot. The dashed lines extrapolate perfect speed-up from the time on the least number of processors.

4.3. Time and parallel speed-up. Figure 1 shows the execution time as a function of allocated OpenMP thread or Cray XMT processor separated by platform and graph. Figure 2 translates the time into speed-up against the best single-thread execution time on the Intel-based platform. The execution times on a single XMT2 processor are too large to permit speed-up studies on these graphs. The results are the best of three runs maximizing modularity with our parallel variant of the Clauset, Newman, and Moore heuristic until the communities contain at least half the edges in the graph. Because fewer possible contractions decrease the conductance, minimizing conductance requires three to five times as many contraction steps and a proportionally longer time.

Maximizing modularity on the 105 million vertex, 3.3 billion edge `uk-2007-05` requires from 496 seconds to 592 seconds using all 80 hardware threads of the Intel E7-8870 platform. The same task on the Cray XMT2 requires from 2 388 seconds to 2 466 seconds.

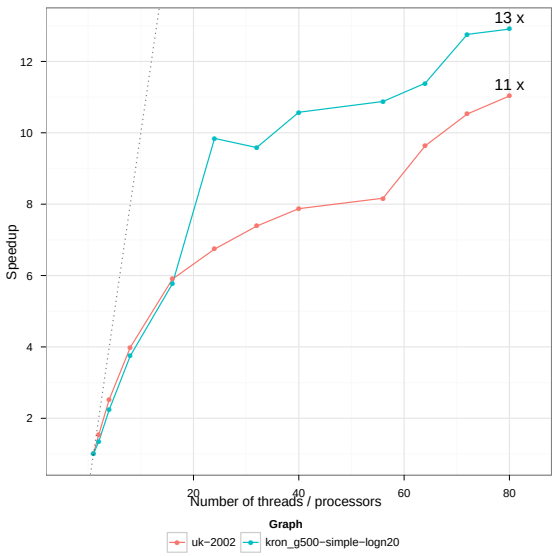

FIGURE 2. Parallel speed-up relative to the best single-threaded execution. The best achieved speed-up is noted on the plot. The dotted line denotes perfect speed-up matching the number of processors.

4.4. Community quality. Computing communities quickly is only good if the communities themselves are useful. Full details are in the workshop report [**32**]. Figure 3 shows the results from two different modularity-maximizing heuristics and one conductance-minimizing heuristic. The real-world `uk-2002` graph shows non-trivial community structure, but the artificial `kron_g500-simple-logn20` lacks such structure [**33**]. There appears to be a significant trade-off between modularity and conductance.

5. Related Work

Graph partitioning, graph clustering, and community detection are tightly related topics. A recent survey by Fortunato [**12**] covers many aspects of community detection with an emphasis on modularity maximization. Nearly all existing work of which we know is sequential and targets specific contraction edge scoring mechanisms. Many algorithms target specific contraction edge scoring or vertex move mechanisms [**14**]. Our previous work [**31, 30**] established and extended the first parallel agglomerative algorithm for community detection and provided results on the Cray XMT. Prior modularity-maximizing algorithms sequentially maintain and update priority queues [**8**], and we replace the queue with a weighted graph matching. Separately from this work, Fagginger Auer and Bisseling developed a similar modularity-optimizing clustering algorithm [**10**]. Their algorithm uses more memory, is more synchronous, and targets execution on GPUs. Fagginger Auer and Bisseling's algorithm performs similarly to ours and includes an interesting star detection technique.

Zhang *et al.* [**37**] recently proposed a parallel algorithm that identifies communities based on a custom metric rather than modularity. Gehweiler and Meyerhenke [**13**]

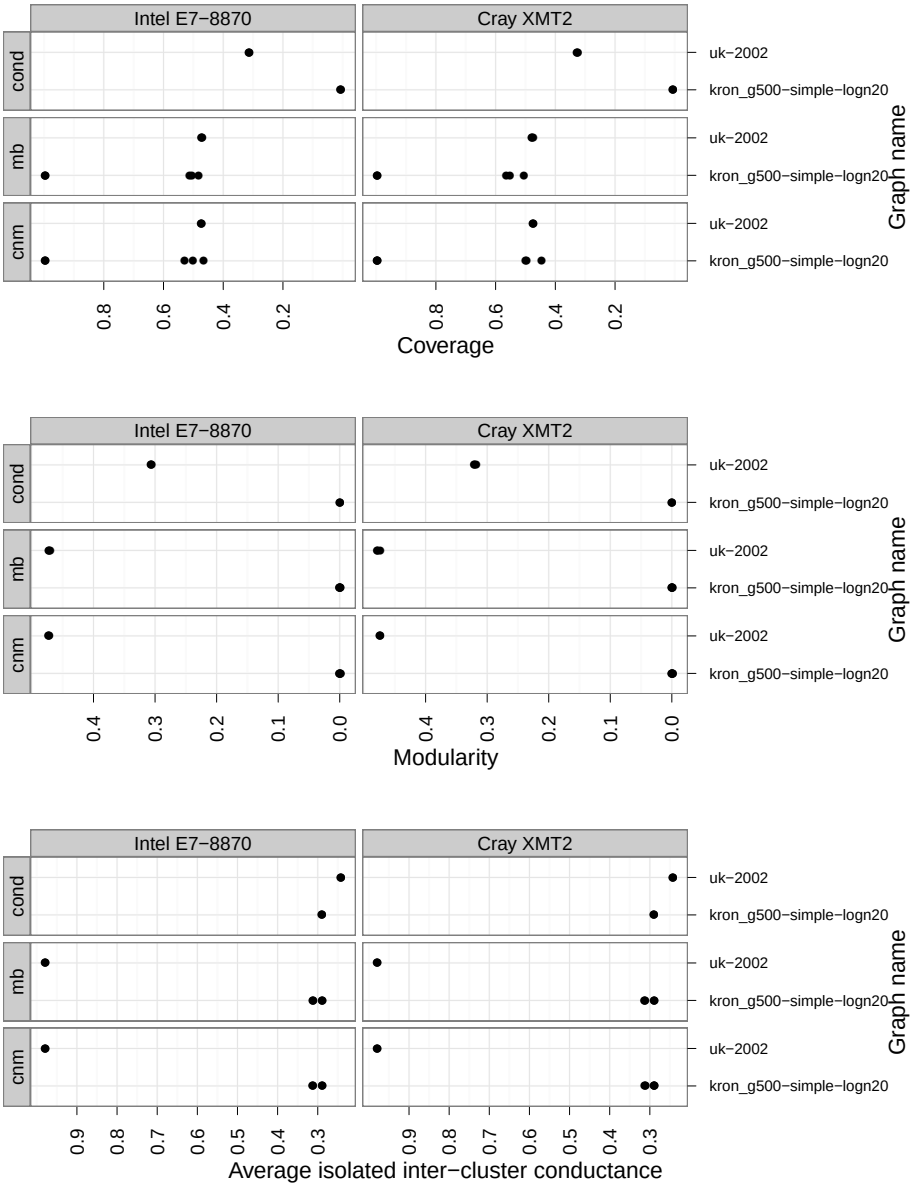

FIGURE 3. Coverage, modularity, and average conductance for the two graphs. The graphs are split vertically by platform and horizontally by scoring method. Here "cnm" and "mb" are the Clauset-Newman-Moore and McCloskey-Bader modularity maximizing heuristics, and "cond" minimizes the conductance.

proposed a distributed diffusive heuristic for implicit modularity-based graph clustering. Classic work on parallel modular decompositions [**24**] finds a different kind of module, one where any two vertices in a module have identical neighbors and somewhat are indistinguishable. This could provide a scalable pre-processing step that collapses vertices that will end up in the same community, although removing the degree-1 fringe may have the same effect.

Work on sequential multilevel agglomerative algorithms like [**23**] focuses on edge scoring and local refinement. Our algorithm is agnostic towards edge scoring methods and can benefit from any problem-specific methods. The Cray XMT's word-level synchronization may help parallelize refinement methods, but we leave that to future work.

6. Observations

Our algorithm and implementation, the first published parallel algorithm for agglomerative community detection, extracts communities with apparently high modularity or low conductance in a reasonable length of time. Finding modularity-maximizing communities in a graph with 105 million vertices and over 3.3 billion edges requires a little over eight minutes on a four processor, Intel E7-8870 based server. Our implementation can optimize with respect to different local optimization criteria, and its modularity results are comparable to a state-of-the-art sequential implementation. As a twist to established sequential algorithms for agglomerative community detection, our parallel algorithm takes a novel and naturally parallel approach to agglomeration with maximum weighted matchings. That difference appears to reduce differences between the CNM and MB edge scoring methods. The algorithm is simpler than existing sequential algorithms and opens new directions for improvement. Separating scoring, choosing, and merging edges may lead to improved metrics and solutions. Our implementation is publicly available[1].

Outside of the edge scoring, our algorithm relies on well-known primitives that exist for many execution models. Much of the algorithm can be expressed through sparse matrix operations, which may lead to explicitly distributed memory implementations through the Combinatorial BLAS [**7**] or possibly cloud-based implementations through environments like Pregel [**19**]. The performance trade-offs for graph algorithms between these different environments and architectures remain poorly understood.

Besides experiments with massive real-world data sets, future work includes the extension of the algorithm to a streaming scenario. In such a scenario, the graph changes over time without an explicit start or end. This extension has immediate uses in many social network applications but requires algorithmic changes to avoid costly recomputations on large parts of the graph.

7. Acknowledgments

This work was supported in part by the Pacific Northwest National Lab (PNNL) Center for Adaptive Supercomputing Software for MultiThreaded Architectures (CASS-MT), NSF Grant CNS-0708307, and the Intel Labs Academic Research Office for the Parallel Algorithms for Non-Numeric Computing Program. We thank PNNL and the Swiss National Supercomputing Centre for providing access to Cray XMT systems. We also thank reviews of previous work inside Oracle.

[1] `http://www.cc.gatech.edu/~jriedy/community-detection/`

References

1. R. Andersen and K. Lang, *Communities from seed sets*, Proc. of the 15th Int'l Conf. on World Wide Web, ACM, 2006, p. 232.
2. D.A. Bader and J. McCloskey, *Modularity and graph algorithms*, Presented at UMBC, September 2009.
3. D.A. Bader, H. Meyerhenke, P. Sanders, and D. Wagner, *Competition rules and objective functions for the 10th DIMACS Implementation Challenge on graph partitioning and graph clustering*, `http://www.cc.gatech.edu/dimacs10/data/dimacs10-rules.pdf`, September 2011.
4. J.W. Berry., B. Hendrickson, R.A. LaViolette, and C.A. Phillips, *Tolerating the community detection resolution limit with edge weighting*, CoRR **abs/0903.1072** (2009).
5. B. Bollobás, *Modern graph theory*, Springer, July 1998.
6. Ulrik Brandes, Daniel Delling, Marco Gaertler, Robert Görke, Martin Hoefer, Zoran Nikoloski, and Dorothea Wagner, *On modularity clustering*, IEEE Trans. Knowledge and Data Engineering **20** (2008), no. 2, 172–188.
7. Aydn Buluç and John R Gilbert, *The Combinatorial BLAS: design, implementation, and applications*, International Journal of High Performance Computing Applications **25** (2011), no. 4, 496–509.
8. A. Clauset, M.E.J. Newman, and C. Moore, *Finding community structure in very large networks*, Physical Review E **70** (2004), no. 6, 66111.
9. Facebook, *Fact sheet*, February 2012, `http://newsroom.fb.com/content/default.aspx?NewsAreaId=22`.
10. B. O. Fagginger Auer and R. H. Bisseling, *Graph coarsening and clustering on the GPU*, Tech. report, 10th DIMACS Implementation Challenge - Graph Partitioning and Graph Clustering, Atlanta, GA, February 2012.
11. S. Fortunato and M. Barthélemy, *Resolution limit in community detection*, Proc. of the National Academy of Sciences **104** (2007), no. 1, 36–41.
12. Santo Fortunato, *Community detection in graphs*, Physics Reports **486** (2010), no. 3-5, 75 – 174.
13. Joachim Gehweiler and Henning Meyerhenke, *A distributed diffusive heuristic for clustering a virtual P2P supercomputer*, Proc. 7th High-Performance Grid Computing Workshop (HGCW'10) in conjunction with 24th Intl. Parallel and Distributed Processing Symposium (IPDPS'10), IEEE Computer Society, 2010.
14. Robert Görke, Andrea Schumm, and Dorothea Wagner, *Experiments on density-constrained graph clustering*, Proc. Algorithm Engineering and Experiments (ALENEX12), 2012.
15. Jaap-Henk Hoepman, *Simple distributed weighted matchings*, CoRR **cs.DC/0410047** (2004).
16. P. Konecny, *Introducing the Cray XMT*, Proc. Cray User Group meeting (CUG 2007) (Seattle, WA), CUG Proceedings, May 2007.
17. Andrea Lancichinetti and Santo Fortunato, *Limits of modularity maximization in community detection*, Phys. Rev. E **84** (2011), 066122.
18. S. Lozano, J. Duch, and A. Arenas, *Analysis of large social datasets by community detection*, The European Physical Journal - Special Topics **143** (2007), 257–259.
19. Grzegorz Malewicz, Matthew H. Austern, Aart J.C Bik, James C. Dehnert, Ilan Horn, Naty Leiser, and Grzegorz Czajkowski, *Pregel: a system for large-scale graph processing*, Proceedings of the 2010 international conference on Management of data (New York, NY, USA), SIGMOD '10, ACM, 2010, pp. 135–146.
20. Fredrik Manne and Rob Bisseling, *A parallel approximation algorithm for the weighted maximum matching problem*, Parallel Processing and Applied Mathematics (Roman Wyrzykowski, Jack Dongarra, Konrad Karczewski, and Jerzy Wasniewski, eds.), Lecture Notes in Computer Science, vol. 4967, Springer Berlin / Heidelberg, 2008, pp. 708–717.
21. M.E.J. Newman, *Modularity and community structure in networks*, Proc. of the National Academy of Sciences **103** (2006), no. 23, 8577–8582.
22. M.E.J. Newman and M. Girvan, *Finding and evaluating community structure in networks*, Phys. Rev. E **69** (2004), no. 2, 026113.
23. Andreas Noack and Randolf Rotta, *Multi-level algorithms for modularity clustering*, Experimental Algorithms (Jan Vahrenhold, ed.), Lecture Notes in Computer Science, vol. 5526, Springer Berlin / Heidelberg, 2009, pp. 257–268.
24. Mark B. Novick, *Fast parallel algorithms for the modular decomposition*, Tech. report, Cornell University, Ithaca, NY, USA, 1989.

25. NYSE Euronext, *Consolidated volume in NYSE listed issues, 2010 - current*, March 2011, `http://www.nyxdata.com/nysedata/asp/factbook/viewer_edition.asp?mode=table&key=3139&category=3`.
26. OpenMP Architecture Review Board, *OpenMP application program interface; version 3.0*, May 2008.
27. Robert Preis, *Linear time 1/2-approximation algorithm for maximum weighted matching in general graphs*, STACS 99 (Christoph Meinel and Sophie Tison, eds.), Lecture Notes in Computer Science, vol. 1563, Springer Berlin / Heidelberg, 1999, pp. 259–269.
28. F. Radicchi, C. Castellano, F. Cecconi, V. Loreto, and D. Parisi, *Defining and identifying communities in networks*, Proc. of the National Academy of Sciences **101** (2004), no. 9, 2658.
29. E. Ravasz, A. L. Somera, D. A. Mongru, Z. N. Oltvai, and A.-L. Barabási, *Hierarchical organization of modularity in metabolic networks*, Science **297** (2002), no. 5586, 1551–1555.
30. E. Jason Riedy, David A. Bader, and Henning Meyerhenke, *Scalable multi-threaded community detection in social networks*, Workshop on Multithreaded Architectures and Applications (MTAAP) (Shanghai, China), May 2012.
31. E. Jason Riedy, Henning Meyerhenke, David Ediger, and David A. Bader, *Parallel community detection for massive graphs*, Proceedings of the 9th International Conference on Parallel Processing and Applied Mathematics (Torun, Poland), September 2011.
32. ———, *Parallel community detection for massive graphs*, Tech. report, 10th DIMACS Implementation Challenge - Graph Partitioning and Graph Clustering, Atlanta, GA, February 2012.
33. C. Seshadhri, Tamara G. Kolda, and Ali Pinar, *Community structure and scale-free collections of Erdös-Rényi graphs*, CoRR **abs/1112.3644** (2011).
34. Twitter, Inc., *Happy birthday Twitter!*, March 2011, `http://blog.twitter.com/2011/03/happy-birthday-twitter.html`.
35. Ken Wakita and Toshiyuki Tsurumi, *Finding community structure in mega-scale social networks*, CoRR **abs/cs/0702048** (2007).
36. Dennis M. Wilkinson and Bernardo A. Huberman, *A method for finding communities of related genes*, Proceedings of the National Academy of Sciences of the United States of America **101** (2004), no. Suppl 1, 5241–5248.
37. Yuzhou Zhang, Jianyong Wang, Yi Wang, and Lizhu Zhou, *Parallel community detection on large networks with propinquity dynamics*, Proceedings of the 15th ACM SIGKDD international conference on Knowledge discovery and data mining (New York, NY, USA), KDD '09, ACM, 2009, pp. 997–1006.

E. Jason Riedy, Georgia Institute of Technology, 266 Ferst Drive, Atlanta, Georgia, 30332, USA
E-mail address: `jason.riedy@cc.gatech.edu`

Henning Meyerhenke, Karlsruhe Institute of Technology, Am Fasanengarten 5, 76131 Karlsruhe, Germany
E-mail address: `meyerhenke@kit.edu`

David Ediger, Georgia Institute of Technology, 266 Ferst Drive, Atlanta, Georgia, 30332, USA
E-mail address: `dediger@gatech.edu`

David A. Bader, Georgia Institute of Technology, 266 Ferst Drive, Atlanta, Georgia, 30332, USA
E-mail address: `bader@cc.gatech.edu`