Scalable Multi-threaded Community Detection in Social Networks

Jason Riedy David A. Bader
College of Computing
Georgia Institute of Technology
Atlanta, GA, USA

Henning Meyerhenke
Institute of Theoretical Informatics
Karlsruhe Institute of Technology
Karlsruhe, Germany

***Abstract*—The volume of existing graph-structured data requires improved parallel tools and algorithms. Finding communities, smaller subgraphs densely connected within the subgraph than to the rest of the graph, plays a role both in developing new parallel algorithms as well as opening smaller portions of the data to current analysis tools. We improve performance of our parallel community detection algorithm by 20% on the massively multithreaded Cray XMT, evaluate its performance on the next-generation Cray XMT2, and extend its reach to Intel-based platforms with OpenMP. To our knowledge, not only is this the first massively parallel community detection algorithm but also the only such algorithm that achieves excellent performance and good parallel scalability across all these platforms. Our implementation analyzes a moderate sized graph with 105 million vertices and 3.3 billion edges in around 500 seconds on a four processor, 80-logical-core Intel-based system and 1100 seconds on a 64-processor Cray XMT2.**

I. Introduction

Graph-structured data inundates daily electronic life. Its volume outstrips the capabilities of nearly all analysis tools. The Facebook friendship network has over 800 million users each with an average of 130 connections [1]. Twitter boasts over 140 million new messages each day [2], and the NYSE processes over 300 million trades each month [3]. Applications of analysis range from database optimization to marketing to regulatory monitoring. Much of the structure within these data stores lies out of reach of current global graph analysis kernels.

One such useful analysis kernel finds smaller communities, subgraphs that locally optimize some connectivity criterion, within these massive graphs. These smaller communities can be analyzed more thoroughly or form the basis for multi-level algorithms. Previously, we introduced the first massively parallel algorithm for detecting communities in massive graphs [4]. Our implementation on the Cray XMT scaled to massive graphs but relied on platform-specific features. Here we both improve the algorithm and extend its reach to OpenMP systems. Our new algorithm scales well on two generations of Cray XMTs and on Intel-based server platforms.

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 [5]. Another definition [6] 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 [7] and also to simplifying large organizational structures [8] and metabolic pathways [9].

The next section briefly touches on related work, including our prior results. Section III reviews

the high-level parallel agglomerative community detection algorithm. Section IV dives into details on the data structures and algorithms, mapping each to the Cray XMT and Intel-based OpenMP platforms' threading architectures. Section V shows that our current implementation achieves speed-ups on real-world data of up to 29.6× on a 64-processor Cray XMT2 and 13.7× on a four processor, 40-physical-core Intel-based platform. On this uk-2007-05 web crawl graph with 105 million vertices and 3.3 billion edges, our algorithm analyzes the community structure in around 500 seconds on a four processor, 80-logical-core Intel-based system and 1100 seconds on a 64-processor Cray XMT2.

II. Related Work

Graph partitioning, graph clustering, and community detection are tightly related topics. A recent survey by Fortunato [10] covers many aspects of community detection with an emphasis on modularity maximization. Nearly all existing work of which we know is sequential, with a very recent notable exception for modularity maximization on GPUs [11]. Many algorithms target specific contraction edge scoring or vertex move mechanisms [12]. Our previous work [4] established 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 [13], and we replace the queue with a weighted graph matching. Here we improve our algorithm, update its termination criteria, and achieve scalable performance on Intel-based platforms.

Zhang *et al*. [14] recently proposed a parallel algorithm that identifies communities based on a custom metric rather than modularity. Gehweiler and Meyerhenke [15] proposed a distributed diffusive heuristic for implicit modularity-based graph clustering.

Work on sequential multilevel agglomerative algorithms like [16] focuses on edge scoring and local refinement. Our algorithm is agnostic towards edge scoring methods and can benefit from any problem-specific methods. Another related approach for graph clustering is due to Blondel *et al*. [17]. However, it does not use matchings and has not been designed with parallelism in mind. Incorporating refinement into our parallel algorithm is an area of active work. The parallel approach is similar to existing multilevel graph partitioning algorithms that use matchings for edge contractions [18], [19] but differs in optimization criteria and not enforcing that the partitions must be of balanced size.

III. 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 [5], [20], [21] (internal connectedness) or minimizing conductance (normalized edge cut) [22]. Here we summarize the algorithm and break it into primitive operations. Section IV 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 [10]. We will not discuss the metrics in detail here; more details are in the references above and our earlier work [4].

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 is one 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 [23]. Any positive-weight matching suffices for optimizing community metrics. Some community metrics, including modularity [24], form NP-complete optimization problems, so additional work improving the matching may not produce better results. Our approach follows existing parallel algorithms [25], [26]. 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 IV.

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 spirit of the 10th DIMACS Implementation Challenge rules [27], Section V'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 $O(|E_c|)$ [23] where E_c is the edge set of the current community graph, each iteration of our algorithm's inner 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 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 [28] and similar parallel implementations [11].

IV. Mapping the Agglomerative Algorithm to Threaded Platforms

Our implementation targets two multithreaded programming environments, the Cray XMT[29] and OpenMP [30], 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, 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 assist automatic parallelization of a wider variety of data dependent loops.

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.

A. Graph representation

We use the same core data structure as our earlier work [4] 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 earlier 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 beginning and ending indices into the edge array for each vertex. In a traditional sparse matrix compressed format, the entries adjacent to vertex $i + 1$ follows 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.

B. Scoring and matching

Each edge's score is an independent calculation for our metrics. An edge $\{i, j\}$ requires its weight, the self-loop weights for i and j, and the total weight of the graph. Parallel computation of the scores is straight-forward, 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 maximal matchings.

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 works 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. We induce a total ordering by considering first score and then the vertex indices. 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 [25], [26] and also produces a maximal matching with weight (total score) within a factor of two 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 unless they fit one of a few textual patterns. 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.

C. Graph contraction

Contracting the agglomerated community graph requires from 40% to 80% of the execution time. Our previous implementation was 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 prior implementation used a technique due to John T. Feo where edges are associated to linked lists by a hash of the vertices. After relabeling an edge's vertices to their new vertex numbers, the associated linked list is searched for that edge. If it exists, the weights are added. If not, the edge is appended to the list. This needs only $|E| + |V|$ additional storage but relies heavily on the Cray XMT's full/empty bits and ability to chase linked lists efficiently. The amount of locking and overhead in iterating over massive, dynamically changing linked lists rendered a similar implementation on Intel-based platforms using OpenMP infeasible.

Our new implementation uses an additional $|E|$ space. After relabeling the vertex endpoints and re-ordering their storage according to the hashing, we 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.

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.

V. Parallel Performance

We evaluate parallel performance on a wide range of threaded hardware including two generations of Cray XMT systems and three different Intel-based systems. We use two graphs, one real and one artificial, 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. We focus on multithreaded performance and leave evaluation of the resulting communities across many metrics to future work. Smaller graphs' resulting modularities appear reasonable compared with results from a different, sequential implementation in SNAP [31].

Our current implementation achieves speed-ups on artificial data of up to 24.8× on a 64-processor Cray XMT2 and 16.5× on a four processor, 40-physical-core Intel-based platform. Smaller real-world data achieves smaller speed-ups. Comparing similar runs of our current Cray XMT implementation with our earlier work shows around a 20% improvement. Our earlier OpenMP implementation executed too slowly to evaluate.

A. Evaluation platforms

The Cray XMT used for these experiments is located at Pacific Northwest National Lab and contains 1 TiB of 533 MHz DDR RAM and 128 Threadstorm processors running at 500 MHz. These 128 processors support over 12 000 hardware

Processor	# proc.	Max. threads/proc.	Proc. speed
Cray XMT	128	100	500MHz
Cray XMT2	64	102	500MHz
Intel E7-8870	4	20	2.40GHz
Intel X5650	2	12	2.66GHz
Intel X5570	2	8	2.93GHz

TABLE I
PROCESSOR CHARACTERISTICS FOR OUR TEST PLATFORMS.

Graph	$\|V\|$	$\|E\|$	Reference
rmat-24-16	15 580 378	262 482 711	[32], [33]
soc-LiveJournal1	4 847 571	68 993 773	[34]
uk-2007-05	105 896 555	3 301 876 564	[35]

TABLE II
SIZES OF GRAPHS USED FOR PERFORMANCE EVALUATION.

thread contexts. The next generation Cray XMT2 is located at the Swiss National Supercomputing Centre (CSCS). Its 64 updated processors also run at 500 MHz but support four times the memory density for a total of 2 TiB on half the nodes of PNNL's XMT. The improvements also include additional memory bandwidth within a node, but exact specifications are not yet officially available.

The Intel-based server platforms are located at Georgia Tech. One has four ten-core Intel Xeon E7-8870 processors running at 2.40GHz with 30MiB of L3 cache per processor. The processors support Hyper-Threading, 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. Two additional servers cluster have two six-core Intel Xeon X5650 processors running at 2.66 GHz with 12MiB of L3 cache each and two four-core Intel Xeon X5570 processors at 2.93 GHz with 8MiB L3 cache. They have 24 GiB and 48 GiB of 1 067 MHz DDR3 RAM, respectively. Both support two Hyper-Threads per core.

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. We show results per-processor on the XMT and per-thread for OpenMP. 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. Table I summarizes the processor counts, threads per processor, and speeds for our test platforms.

B. Test graphs

We evaluate on two moderate-sized graphs. Excessive single-processor runs on highly utilized resources are discouraged, rendering scaling studies using large graphs difficult. Table II shows the graphs' names and number of vertices and edges.

Our first graph is an artificial R-MAT [32], [33] graph derived by sampling from a perturbed Kronecker product. R-MAT graphs are scale-free and reflect many properties of real social networks but are known not to possess significant community structure [36]. We generate an R-MAT graph with parameters $a = 0.55$, $b = c = 0.1$, and $d = 0.25$ and extract the largest component. An R-MAT generator takes a scale s, here 24, and edge factor f, here 16, as input and generates a sequence of $2^s \cdot f$ edges over 2^s vertices, including self-loops and repeated edges. We accumulate multiple edges within edge weights and then extract the largest connected component.

Our second graph is an anonymous snapshot of the LiveJournal friendship network from the Stanford Large Network Dataset Collection [34]. The graph is relatively small but is rich with community structures. There are no self-loops or multiple edges, and all weights initially are set to one. The final, large graph was generated by a web crawl of English sites in 2007 [35]. This graph is too large for the smaller Intel-based platforms and triggers platform bugs on the Cray XMT, so we consider it only on the larger Intel E7-8870 and updated Cray XMT2.

C. 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, and Table III scales the time by the number of edges to show the peak processing rate. Figure 2 translates the

Platform	soc-LiveJournal1	rmat-24-16	uk-2007-05
X5570	3.89×10^6	1.83×10^6	
X5650	4.98×10^6	2.54×10^6	
E7-8870	6.90×10^6	5.86×10^6	6.54×10^6
XMT	0.41×10^6	1.20×10^6	
XMT2	1.73×10^6	2.11×10^6	3.11×10^6

TABLE III
THE PEAK PROCESSING RATE ACHIEVED IN EDGES PER SECOND OF THE INPUT GRAPH OVER THE FASTEST TIME. OUR IMPLEMENTATION SCALES TO LARGE DATA SETS.

time into speed-up against the best single-thread or single-processor execution time. Figure 3 shows the larger uk-2007-05 graph's data. Note that the uk-2007-05 graph uses 32-bit integers for vertex labels on the Intel-based platform to fit in memory.

Comparing the Cray XMT to the Cray XMT2 shows substantial performance improvements in the new generation. While details are not publicly available, we suspect increased memory bandwidth related to wider memory paths and an upgrade from slow DDR memory are responsible. The variation in time on the Cray XMT2 appears related to finding different community structures and is a consequence of our non-deterministic algorithm. Monitoring execution shows that the XMT compiler under-allocates threads in portions of the code, leading to burst of poor processor utilization. Explicitly annotating the loops with thread counts may increase performance.

Trading the earlier list-chasing implementation for our new algorithm with bursts of sequential access shows good performance on Intel-based platforms. Within the Intel-based platforms we see speed-ups past physical cores and into logical cores with Hyper-Threading. The 2.40GHz E7-8870 achieves better single-core performance than the 2.93GHz X5570 but lower performance than the 2.66GHz X5650. The X5570 is an earlier generation of processor with a less advanced memory controller that supports fewer outstanding transactions. This data is insufficient to see if a single, slower E7-8870's additional cores can outperform the faster X5650's fewer cores. With twice the processors, the E7-8870-based system performs more than twice as quickly as the X5650 on the artificial R-MAT graph but only slightly faster on the real-world soc-LiveJournal1 graph.

The reproducible drop in performance on the X5650 at full processor count on soc-LiveJournal1 may be related to finding a different community structure. While still under investigation, there may be an unfortunate locking interaction within the matching phase. A similar but smaller issue occurs on the Cray XMT with soc-LiveJournal1 above 64 processors.

Except for the one case on the X5650, the best performance on Intel-based platforms always occurred at full utilization. The smaller size of soc-LiveJournal1 provides insufficient parallelism for large processor counts on the XMTs, but the larger uk-2007-05 shows scaling similar to the artificial rmat-24-16.

VI. OBSERVATIONS

Our improved parallel agglomerative community detection algorithm demonstrates high performance, good parallel scalability, and good data scalability on both the Cray XMT series and more common Intel-based platforms using OpenMP. We achieved high performance by carefully choosing data structures and algorithms that map well to both environments. Reducing linked list walking, locking, and synchronization improves performance on both the Cray XMT and Intel-based architecures. 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 [37] or possibly cloud-based implementations through environments like Pregel [38]. The performance trade-offs for graph algorithms between these different environments and architectures remains poorly understood.

ACKNOWLEDGMENTS

This work was supported in part by the Pacific Northwest National Lab (PNNL) Center

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

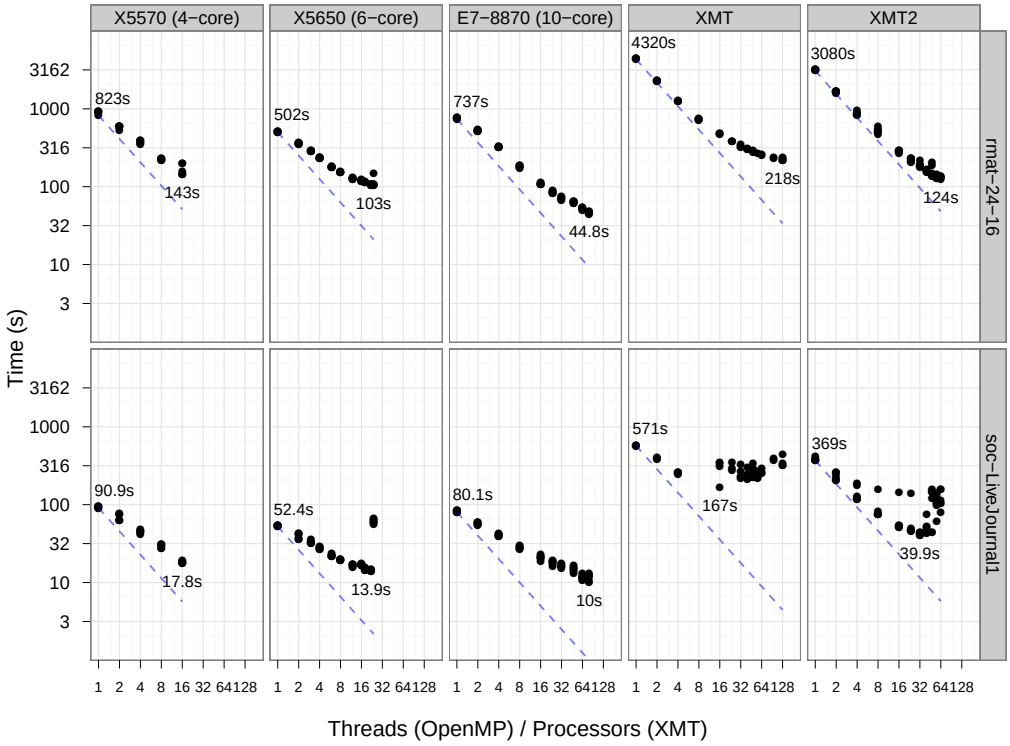

Fig. 1. Execution time against allocated OpenMP threads or Cray XMT processors per platform and graph. The best single-processor and overall times are noted in the plot. The blue dashed line extrapolates perfect speed-up from the best single-processor time.

for Adaptive Supercomputing Software for Multi-Threaded 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. Attendees of the 10th DIMACS Implementation Challenge and anonymous reviewers from MTAAP and within Oracle provided helpful comments.

References

[1] Facebook, Inc., "User statistics," March 2011, http://www.facebook.com/press/info.php?statistics.

[2] Twitter, Inc., "Happy birthday Twitter!" March 2011, http://blog.twitter.com/2011/03/happy-birthday-twitter.html.

[3] 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.

[4] E. J. Riedy, H. Meyerhenke, D. Ediger, and D. A. Bader, "Parallel community detection for massive graphs," in *Proc. 9th International Conference on Parallel Processing and Applied Mathematics*, Torun, Poland, Sep. 2011.

[5] M. Newman and M. Girvan, "Finding and evaluating community structure in networks," *Phys. Rev. E*, vol. 69, no. 2, p. 026113, Feb 2004.

[6] F. Radicchi, C. Castellano, F. Cecconi, V. Loreto, and D. Parisi, "Defining and identifying communities in networks," *Proc. of the National Academy of Sciences*, vol. 101, no. 9, p. 2658, 2004.

[7] D. M. Wilkinson and B. A. Huberman, "A method for finding communities of related genes," *Proceedings of the National Academy of Sciences of the United States of America*, vol. 101, no. Suppl 1, pp. 5241–5248, 2004.

[8] S. Lozano, J. Duch, and A. Arenas, "Analysis of large social datasets by community detection," *The European Physical Journal - Special Topics*, vol. 143, pp. 257–259, 2007.

[9] E. Ravasz, A. L. Somera, D. A. Mongru, Z. N. Oltvai, and A.-L. Barabási, "Hierarchical organization of modularity in metabolic networks," *Science*, vol. 297, no. 5586, pp. 1551–1555, 2002.

[10] S. Fortunato, "Community detection in graphs," *Physics Reports*, vol. 486, no. 3-5, pp. 75 – 174, 2010.

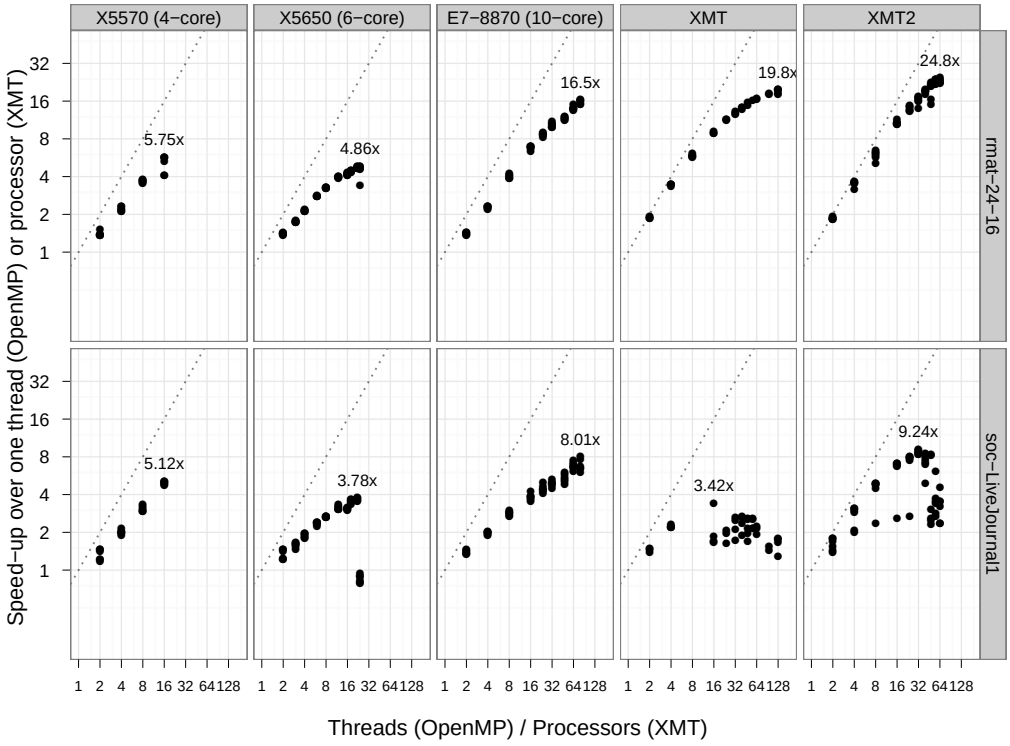

Fig. 2. Parallel speed-up relative to the best single-processor execution. The best achieved speed-up is noted on the plot.The dotted line denotes perfect speed-up matching the number of processors.

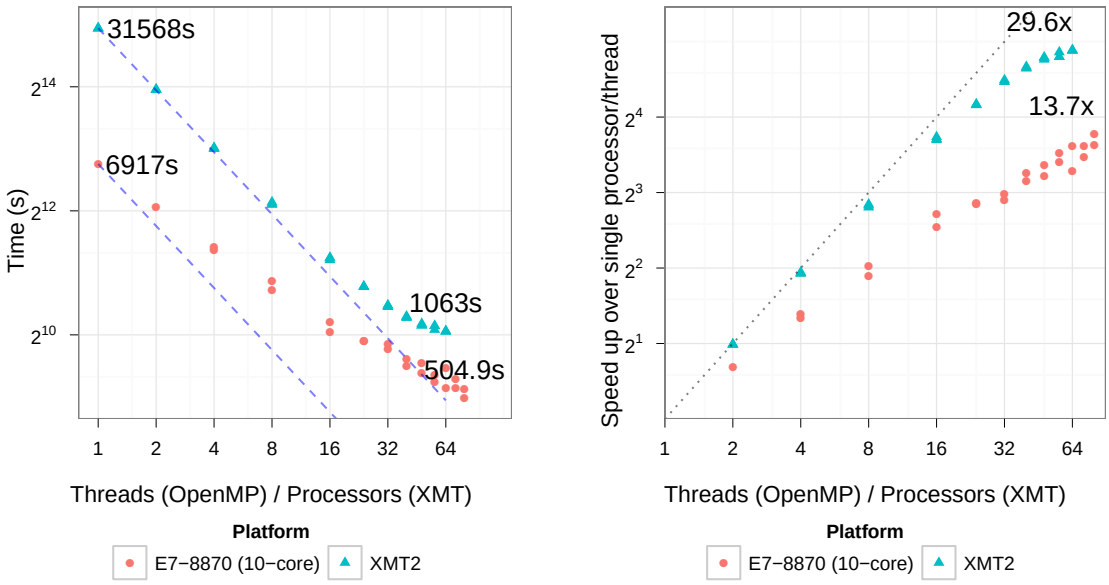

Fig. 3. Parallel speed-up relative to the best single-processor execution on the uk-2007-05 graph.

[11] B. O. F. Auer and R. H. Bisseling, "10th DIMACS implementation challenge: Graph coarsening and clustering on the gpu," 2012, http://www.cc.gatech.edu/dimacs10/papers/[16]-gpucluster.pdf.

[12] R. Görke, A. Schumm, and D. Wagner, "Experiments on density-constrained graph clustering," in *Proc. Algorithm Engineering and Experiments (ALENEX12)*, 2012.

[13] A. Clauset, M. Newman, and C. Moore, "Finding community structure in very large networks," *Physical Review E*, vol. 70, no. 6, p. 66111, 2004.

[14] Y. Zhang, J. Wang, Y. Wang, and L. Zhou, "Parallel community detection on large networks with propinquity dynamics," in *Proceedings of the 15th ACM SIGKDD international conference on Knowledge discovery and data mining*, ser. KDD '09. New York, NY, USA: ACM, 2009, pp. 997–1006.

[15] J. Gehweiler and H. Meyerhenke, "A distributed diffusive heuristic for clustering a virtual P2P supercomputer," in *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.

[16] A. Noack and R. Rotta, "Multi-level algorithms for modularity clustering," in *Experimental Algorithms*, ser. Lecture Notes in Computer Science, J. Vahrenhold, Ed. Springer, 2009, vol. 5526, pp. 257–268.

[17] V. D. Blondel, J.-L. Guillaume, R. Lambiotte, and E. Lefebvre, "Fast unfolding of communities in large networks," *J. Stat. Mech*, p. P10008, 2008.

[18] G. Karypis and V. Kumar, "Multilevel k-way partitioning scheme for irregular graphs," *Journal of Parallel and Distributed Computing*, vol. 48, no. 1, pp. 96–129, 1998.

[19] M. Holtgrewe, P. Sanders, and C. Schulz, "Engineering a scalable high quality graph partitioner," in *Proc. 24th IEEE Intl. Symposium on Parallel and Distributed Processing (IPDPS'10)*, 2010, pp. 1–12.

[20] D. Bader and J. McCloskey, "Modularity and graph algorithms," Sep. 2009, presented at UMBC.

[21] M. Newman, "Modularity and community structure in networks," *Proc. of the National Academy of Sciences*, vol. 103, no. 23, pp. 8577–8582, 2006.

[22] R. Andersen and K. Lang, "Communities from seed sets," in *Proc. of the 15th Int'l Conf. on World Wide Web*. ACM, 2006, p. 232.

[23] R. Preis, "Linear time 1/2-approximation algorithm for maximum weighted matching in general graphs," in *STACS 99*, ser. Lecture Notes in Computer Science, C. Meinel and S. Tison, Eds. Springer Berlin / Heidelberg, 1999, vol. 1563, pp. 259–269.

[24] U. Brandes, D. Delling, M. Gaertler, R. Görke, M. Hoefer, Z. Nikoloski, and D. Wagner, "On modularity clustering," *IEEE Trans. Knowledge and Data Engineering*, vol. 20, no. 2, pp. 172–188, 2008.

[25] J.-H. Hoepman, "Simple distributed weighted matchings," *CoRR*, vol. cs.DC/0410047, 2004.

[26] F. Manne and R. Bisseling, "A parallel approximation algorithm for the weighted maximum matching problem," in *Parallel Processing and Applied Mathematics*, ser. Lecture Notes in Computer Science, R. Wyrzykowski, J. Dongarra, K. Karczewski, and J. Wasniewski, Eds. Springer, 2008, vol. 4967, pp. 708–717.

[27] D. 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," Sep. 2011, http://www.cc.gatech.edu/dimacs10/data/dimacs10-rules.pdf.

[28] K. Wakita and T. Tsurumi, "Finding community structure in mega-scale social networks," *CoRR*, vol. abs/cs/0702048, 2007.

[29] P. Konecny, "Introducing the Cray XMT," in *Proc. Cray User Group meeting (CUG 2007)*. Seattle, WA: CUG Proceedings, May 2007.

[30] *OpenMP Application Program Interface; Version 3.0*, OpenMP Architecture Review Board, May 2008.

[31] D. Bader and K. Madduri, "SNAP, small-world network analysis and partitioning: an open-source parallel graph framework for the exploration of large-scale networks," in *Proc. Int'l Parallel and Distributed Processing Symp. (IPDPS)*, 2008.

[32] D. Chakrabarti, Y. Zhan, and C. Faloutsos, "R-MAT: A recursive model for graph mining," in *Proc. 4th SIAM Intl. Conf. on Data Mining (SDM)*. Orlando, FL: SIAM, Apr. 2004.

[33] D. Bader, J. Gilbert, J. Kepner, D. Koester, E. Loh, K. Madduri, W. Mann, and T. Meuse, *HPCS SSCA#2 Graph Analysis Benchmark Specifications v1.1*, Jul. 2005.

[34] J. Leskovec, "Stanford large network dataset collection," At http://snap.stanford.edu/data/, Oct. 2011.

[35] P. Boldi, B. Codenotti, M. Santini, and S. Vigna, "Ubicrawler: A scalable fully distributed web crawler," *Software: Practice & Experience*, vol. 34, no. 8, pp. 711–726, 2004.

[36] C. Seshadhri, T. G. Kolda, and A. Pinar, "Community structure and scale-free collections of erdös-rényi graphs," *CoRR*, vol. abs/1112.3644, 2011.

[37] A. Buluç and J. R. Gilbert, "The Combinatorial BLAS: design, implementation, and applications," *International Journal of High Performance Computing Applications*, vol. 25, no. 4, pp. 496–509, 2011. [Online]. Available: http://hpc.sagepub.com/content/25/4/496.abstract

[38] G. Malewicz, M. H. Austern, A. J. Bik, J. C. Dehnert, I. Horn, N. Leiser, and G. Czajkowski, "Pregel: a system for large-scale graph processing," in *Proceedings of the 2010 international conference on Management of data*, ser. SIGMOD '10. New York, NY, USA: ACM, 2010, pp. 135–146.