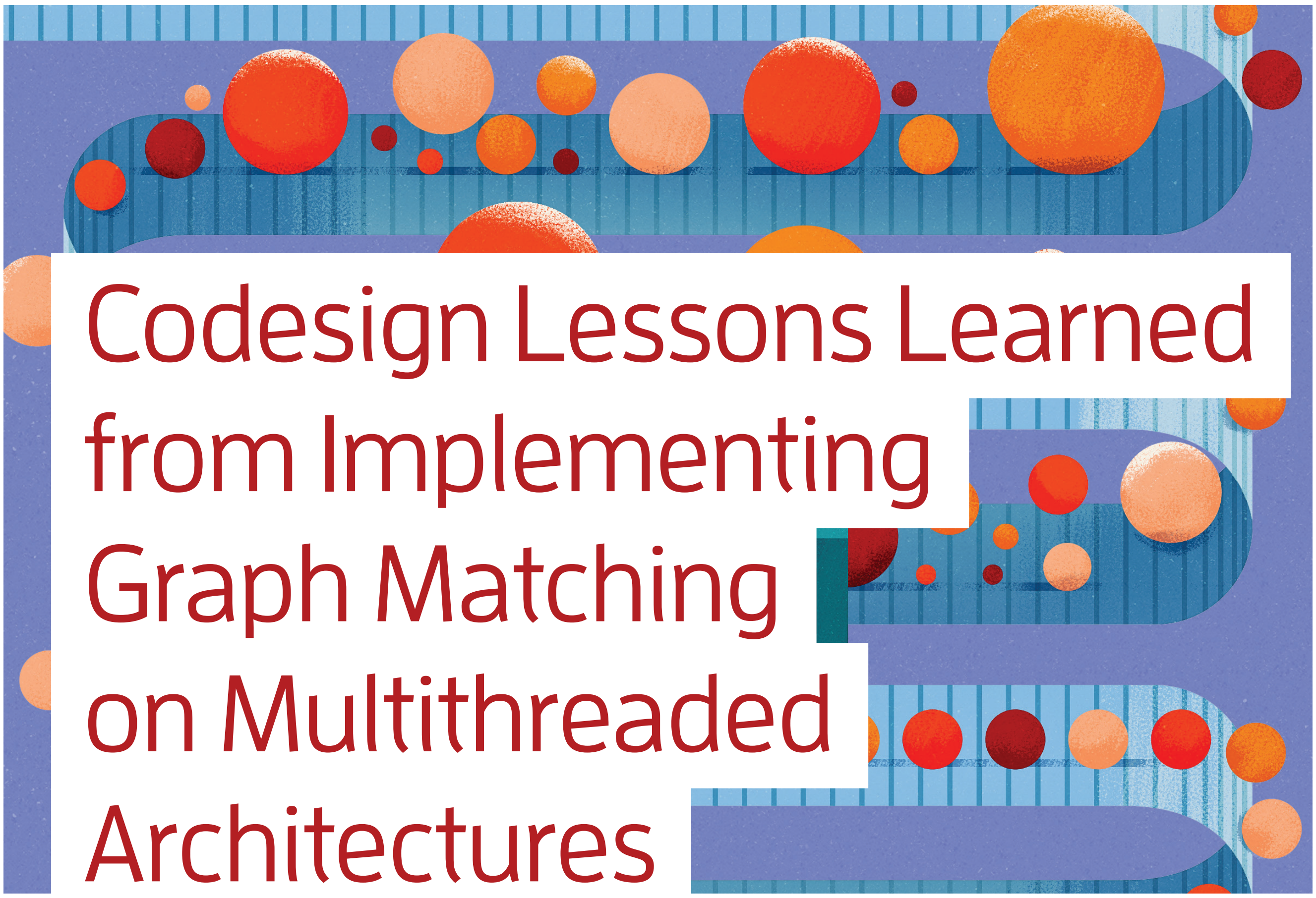


Codesign Lessons Learned from Implementing Graph Matching on Multithreaded Architectures

Mahantesh Halappanavar, Pacific Northwest National Laboratory
Alex Pothen, Purdue University
Ariful Azad, Lawrence Berkeley National Laboratory
Fredrik Manne, University of Bergen
Johannes Langguth, Simula Research Laboratory
Arif Khan, Purdue University

Executing irregular, data-intensive workloads on multithreaded architectures can result in performance losses and scalability problems. Codesigning algorithms and architectures can realize high performance on irregular applications. A codesign study reveals four key lessons learned from implementing matching algorithms on various platforms.

Improving the performance of irregular applications in graph analytics, data science, network science, and similar areas on multithreaded architectures is a challenge because these applications demonstrate unpredictable workloads and data-access patterns. Codesigning algorithms and architectures to accommodate irregular, data-intensive workloads is one effective way to tackle this problem. Using graph matching

TABLE 1. Characteristics of approximate and optimal matching algorithms.*

Algorithm	Search type	Serial time complexity	Parallelization strategy	Architecture preference
Half-approximate matching in general graphs				
Greedy	Local	$O(m \log n)$	Not concurrent	Few fast threads
Locally dominant (weighted)	Local	$O(m\Delta)$	Block of vertices per thread, queue	Moderate number of threads
Suitor (weighted)	Local	$O(m\Delta)$	Block of vertices per thread, lock synchronization	Massive multithreading
Maximum-cardinality matching in bipartite graphs				
Pothen–Fan	Vertex-disjoint alternating DFSs	$O(mn)$	One DFS per thread (coarse-grained)	Moderate number of threads
MS–BFS	Vertex-disjoint alternating BFSs	$O(mn)$	One vertex per thread (fine-grained)	Massive multithreading
Hopcroft–Karp	Both alternating BFS and DFS	$O(mn^{1/2})$	Both fine- and coarse-grained	Heterogeneous
Push–Relabel	Label-guided FIFO search	$O(mn)$	One vertex per thread (fine-grained)	Heterogeneous
Karp–Sipser (half-approximate)	Local	$O(m)$	Block of vertices per thread; synchronize using atomics	Massive multithreading

* m: number of edges; n: number of vertices; Δ: maximum degree of a vertex; BFS: breadth-first search; DFS: depth-first search; FIFO: first in, first out.

as a case study, our work explores the interplay between algorithm design and architectural features.

The first reported results on parallel matching date back to the first DIMACS (Center for Discrete Mathematics and Theoretical Computer Science) Implementation Challenge held in 1990–1991. One of the studies by Martin Brady and his colleagues compared the performance of auction algorithms on platforms such as the WaveTracer Zephyr, MasPar MP-1, and Silicon Graphics IRIS 4d/340 DVX. They described their results as a "little short of disastrous." At that time, they argued that the ideal platform would be a machine with "relatively few numbers of powerful processors, support for block transfers, lots of memory, and a high processor-memory bandwidth to this memory."[1] Modern multicore and many-core platforms have some of these ideal characteristics, which we exploit in our design and implementations.

As part of this work, we also propose matching as a better benchmark than other kernels, such as breadth-first search (BFS), to evaluate architectures and algorithm design. Although matching subsumes BFS, it also exposes characteristics such as fine-grained synchronization and execution serialization. Using two variants of the matching problem, several algorithms, and implementations on diverse platforms, here we share (often counterintuitive) lessons learned from implementing irregular applications on parallel (multithreaded) architectures. Specifically, we present four key codesign lessons that we have found to be critical for high performance when implementing matching algorithms on various platforms.

GRAPH MATCHING AND MATCHING ALGORITHMS

Matching is a prototypical combinatorial problem with numerous applications in science and engineering. Given a graph $G = (V, E)$, a matching M is a subset of independent edges—that is, no two edges in M share a common vertex. This problem has applications in combinatorial scientific computing,[2–4] network alignment,[5] fault tolerance,[6] optical network switching,[7] image processing,[8] data privacy,[9] genome sequencing,[10] and computational immunology.[11]

The matching problem has several variants, depending on the objective function and the graph type. A *maximum matching* is one with the maximum number of matched edges. A maximum (edge) weight matching maximizes the sum (or product) of the weight of the matched edges. (Researchers have also developed vertex-weighted matchings,[12] but we will not consider those here.) Algorithms that compute optimal matchings are conceptually simpler if the graph is bipartite. Approximation algorithms that guarantee a matching that is within some constant fraction of the optimal matching have lower time complexities, are amenable to parallelization, and are simpler to implement.

In this article, we focus on two problems: half-approximate weighted matchings in general graphs and maximum-cardinality matchings in bipartite graphs. Table 1 summarizes the characteristics of each. A half-approximate weighted matching guarantees a solution that is at least half the weight of a maximum weight matching. (Usually it also has at least half the cardinality of a maximum-cardinality matching.) Approximation algorithms not only provide fast initialization in optimal algorithms, but they often suffice in many iterative algorithms where we use matching to improve the current solution.[5,13]

Augmentation is a fundamental technique used in exact-matching algorithms. A path in a graph is a sequence of edges such that any two consecutive edges on the path share a common vertex, and the vertices in the path are unique. An alternating path is a path that has alternate matched and unmatched edges. An alternating path that starts and ends with an unmatched vertex is an augmenting path. Given an augmenting path, the cardinality of a matching can be increased by one by exchanging the matched edges with unmatched edges along this path. A straightforward algorithm for computing a maximum-cardinality matching iteratively finds an augmenting path relative to the current matching and then augments along such a path. A matching has maximum cardinality if and only if there is no augmenting path relative to it. Broadly, the search for augmenting paths can be breadth-first or depth-first. In contrast to a maximum-cardinality matching, a *maximal matching* is one that cannot be augmented by adding a new edge to it. Several book-length expositions of matching provide further information.[14–16]

MATCHING IS A PROTOTYPICAL COMBINATORIAL PROBLEM WITH NUMEROUS APPLICATIONS IN SCIENCE AND ENGINEERING.

In our previous and ongoing work, we test matching algorithms on several generations of a diverse set of multithreaded architectures such as the AMD Opteron, the Intel Xeon and Intel Xeon Phi, the Cray XMT, and Nvidia GPUs. Collectively, these architectures represent a broad spectrum of capabilities: clock speeds ranging from 0.5 GHz (XMT) to about 3.0 GHz (Xeon); hardware multithreading ranging from none (Opteron) to 128 threads per processor (XMT); cache hierarchies ranging from none/flat (XMT) to three levels (Xeon); memory interconnect generations ranging from DDR1 (XMT) to GDDR5 (Tesla K40); advanced features on Intel and AMD architectures (such as branch prediction and speculative execution) to simple XMT designs; and control structures ranging from fully autonomous multiple instruction, multiple data (MIMD) processors (Opteron) to a 32-way single instruction, multiple data (SIMD) (Tesla). Different platforms employ different techniques for performance. For example, the XMT uses massive multithreading to tolerate latency from memory operations, whereas the Xeon employs deep cache hierarchies, advanced branch prediction, and two-way multithreading to tolerate latencies. The contrasting features in these platforms offer a rich environment to study how performance is influenced by hardware features and algorithm design.

LESSON 1: APPROXIMATION

Fast-approximation algorithms play an important role in matching. Exact algorithms for edge-weighted matching search for long augmenting paths or transmit information along long paths and are inherently sequential. Slow convergence to optimal solutions with reduced work in later iterations also negates the gains from parallelization. However, approximation algorithms restrict the search to short augmenting paths and thereby to local neighborhoods, thus avoiding the long tail in convergence to optimality.

As an example of a half-approximation algorithm for edge-weighted matching, the greedy algorithm considers edges in nonincreasing order of weights. In each iteration, it adds a current heaviest edge to the matching and removes all other edges that share the endpoints of the matched edge. The algorithm iterates until the graph is empty. Because this algorithm needs to sort edges by weight and process edges in this order, it doesn't have much concurrency.

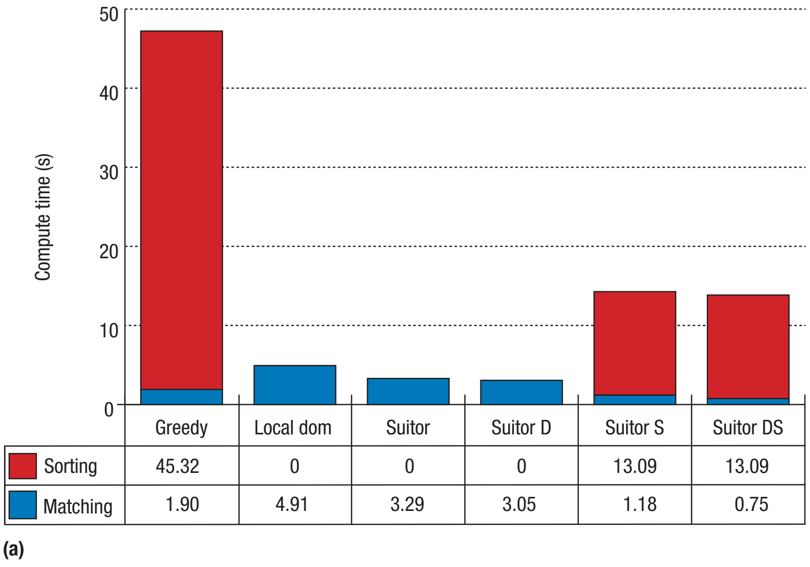
Robert Preis observed that we can obtain a half-approximation algorithm without sorting by iteratively matching locally dominant edges—that is, an edge at least as heavy as all other edges that share its endpoints.[17] This algorithm can be implemented in time linear in the number of edges. It has increased concurrency relative to the greedy algorithm. Fredrik Manne and Rob Bisseling adapted this algorithm for a parallel implementation on a distributed-memory platform,[18] and Mahantesh Halappanavar and his colleagues subsequently adapted it to multithreaded architectures.[2]

The locally dominant edge algorithm can be implemented by having each vertex set a pointer to its heaviest unmatched neighbor. The two endpoints of a locally dominant edge point to each other. This requires breaking ties in edge weights consistently, for example, by selecting the lower-numbered vertex. The algorithm adds the locally dominant edges to the matching and updates the pointers of each unmatched vertex to its next-heaviest neighbor, if any. The algorithm iterates until there are no vertices to process. In a parallel context, the search for the next-heaviest neighbor of each vertex can be done concurrently. The number of vertices eligible to be matched in a given iteration represents the amount of concurrent work, which decreases as the algorithm progresses. This number decreases approximately by half per iteration for several classes of graphs.[2] A shared work queue is maintained to identify the vertices that need to be matched in the next iteration. Different threads working in parallel synchronize to add new vertices to this queue. The total number of iterations, the amount of concurrent work per iteration, and hardware support for atomic operations affect this algorithm's performance.

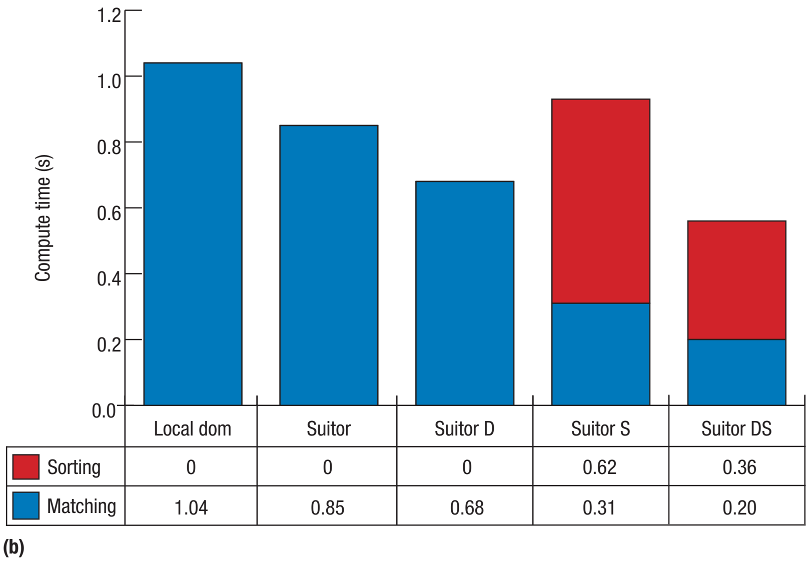
We recently introduced the Suitor algorithm, which increases the concurrency available and reduces the number of edges traversed by the locally dominant edge algorithm.[19] In this algorithm, each vertex proposes to its heaviest neighbor and keeps track of the best offer it receives from a neighbor, which is the weight of the edge that joins them. When a vertex v proposes to a neighbor w, we say that v is the suitor of w. When a vertex v considers proposing to a neighbor w, it examines the best offer w currently has. If this weight is higher than the weight v has to offer, then v's proposal cannot succeed, and hence it moves on to its next-heaviest neighbor. If this weight is less than the weight v has to offer, then v annuls the proposal from the current suitor of w (say x), proposes to w, and adds x to a queue of vertices that need to extend a proposal in the future. Because multiple vertices could seek to update the current best offer of a particular vertex and the shared queue of vertices that need to be processed again, we have to synchronize these location updates using locks. Even with the heavy-weight synchronization performed through locks, the Suitor algorithm performs better than the locally dominant edge algorithm in parallel because it substantially reduces the number of edges that need to be searched in the algorithm.

The Suitor algorithm has several variants. When the suitor of a vertex is dislodged, we can search for a vertex for it to propose to immediately or add it to the end of a queue to be processed in the next iteration. The former process is called *eager update*, and the latter is called *delayed update*. If the neighbor list of each vertex is sorted in nonincreasing order of weights, then we can search this list once from highest to lowest weight during the course of the algorithm. Once a vertex in the adjacency list of a vertex is an ineligible partner, then it will continue to be so for the remainder of the algorithm. The sorting here involves a number of local neighborhood lists, not the entire edge list, so it has more concurrency. The sorted variant of the Suitor algorithm has $O(n + m \log \Delta)$ worst-case time complexity. In practice, the sorted algorithm is preferable when the matching needs to be computed a number of times in the context of an iterative algorithm.

Figure 1 shows the serial and parallel runtimes for these algorithms. The serial results in Figure 1a show that the locally dominant edge algorithm has superior performance relative to the greedy algorithm, but the Suitor algorithms perform better than the locally dominant edge algorithm. The results show that the Suitor algorithm benefits from the reduced work and increased concurrency. The cost of sorting in a serial context hurts the performance of the Suitor variants, but these costs are reduced in a parallel context because neighbor lists are sorted in parallel (see Figure 1b). (Input is a recursive matrix [R-MAT][20] graph with 1 million vertices and 67 million edges.) The algorithm is further improved by delaying the search for partners for vertices whose proposals get annulled until the next iteration. (Such vertices are unlikely to be matched in the final approximate matching, and this feature also



	Greedy	Local dom	Suitor	Suitor D	Suitor S	Suitor DS
🟥 Sorting	45.32	0	0	0	13.09	13.09
🟦 Matching	1.90	4.91	3.29	3.05	1.18	0.75



	Local dom	Suitor	Suitor D	Suitor S	Suitor DS
🟥 Sorting	0	0	0	0.62	0.36
🟦 Matching	1.04	0.85	0.68	0.31	0.20

FIGURE 1. Relative performance of different edge-weighted approximation algorithms. (a) Serial performance on Intel Xeon E5-2670 running at a base frequency of 2.6 GHz (maximum turbo frequency of 3.3 GHz) and (b) parallel performance on an Intel Xeon Phi coprocessor 5120D running at 1 GHz, using 240 threads on 60 cores. Input is an recursive matrix (R-MAT) graph with 1 million vertices and 67 million edges. The four variants of the Suitor algorithm include the base algorithm and those augmented with delayed updates (D), sorted adjacency lists (S), and both features (DS). The sorting time is shown in red for the algorithms that sort their edges or adjacency lists, and the matching time is shown in blue.

better exploits spatial and temporal locality in the memory accesses.)

The cost of synchronizations among the large number of threads is an architectural feature relevant to approximation algorithms. The greedy algorithm cannot effectively support many threads because of the ordering in which edges should be processed. The locally dominant edge algorithm has high synchronization costs because it inserts and deletes vertices to be matched in a shared work queue. Shared queues can also cause memory hotspots. The Suitor algorithm supports many more threads because a vertex can extend a proposal to a neighbor speculatively since it can be annulled by another vertex. The algorithm also reduces the number of proposals a vertex makes by keeping track of the best offer each neighbor has currently. Sorting the adjacency lists ensures that a vertex needs to search its adjacency list only once, and delaying the processing of vertices whose proposals have been annulled reduces cache misses.

LESSON 2: DATAFLOW DESIGN

We now consider the design of algorithms for massively multithreaded machines such as the Cray XMT. The XMT platform builds a global address space from physically distributed memory modules associated with processors. The address space itself is built using a hardware hashing mechanism that maps data randomly to the memory modules in blocks of 64 bytes to minimize conflicts and hotspots.[21] The latencies from memory accesses are tolerated using massive (128-way) multithreading. The XMT has special tag bits to denote whether memory locations are full or empty, and utilizes extended memory semantics to support such reads and writes. We redesigned matching algorithms using dataflow principles to avoid work queues on the XMT and developed a novel dataflow algorithm that performs and scales better than the queue-based implementation.[2]

Figure 2 shows the scalability of the locally dominant edge and Suitor algorithms on the XMT. The Suitor algorithm's implementation is facilitated by fine-grained synchronization using the full/empty tag bit and extended memory semantics. On the XMT, we observed up to seven times improvement in performance for the Suitor algorithm over the locally dominant edge algorithm; it also enjoys better scalability. The benefits of dataflow design, exemplified by the Suitor algorithm, are evident from its superior performance across different multithreaded platforms where synchronization was implemented using OpenMP locks because there is no explicit hardware support for synchronization.[19] Even with the use of such a heavyweight synchronization mechanism, the Suitor algorithm has superior performance relative to the queue-based locally dominant edge algorithm.

The key architectural feature enabling dataflow algorithms is hardware support for fine-grained synchronization. The work in alternating BFS-based parallel matching algorithms is fine-grained, with a significant need for synchronization among threads executing concurrently. Although hardware-supported atomic memory operations were useful in some cases, the support for fine-grained synchronization enabled the redesign of algorithms that yielded substantial performance improvements. The dataflow design overcomes load balancing, memory contention, and hotspotting problems. We observed that the design not only improves the overall performance but also results in better scalability of the dataflow algorithms.

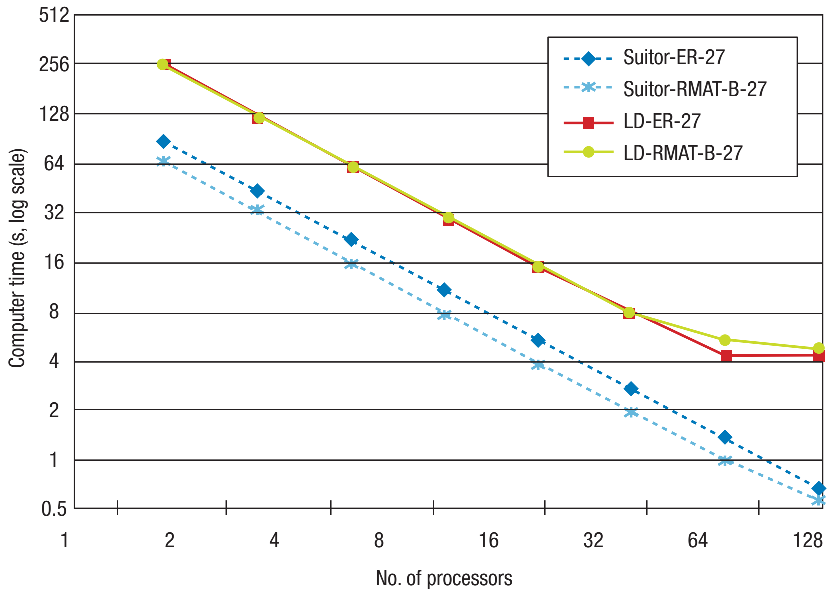
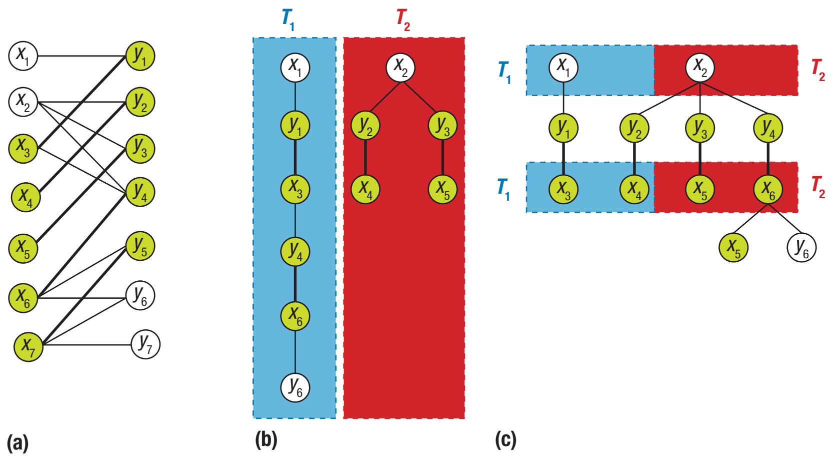


FIGURE 2. Scalability of the locally dominant edge and Suitor algorithms on a Cray XMT for two types of R-MAT graphs (134 million vertices and over a billion edges). The Suitor algorithm is two to seven times faster than the locally dominant (LD) algorithm and scales better.

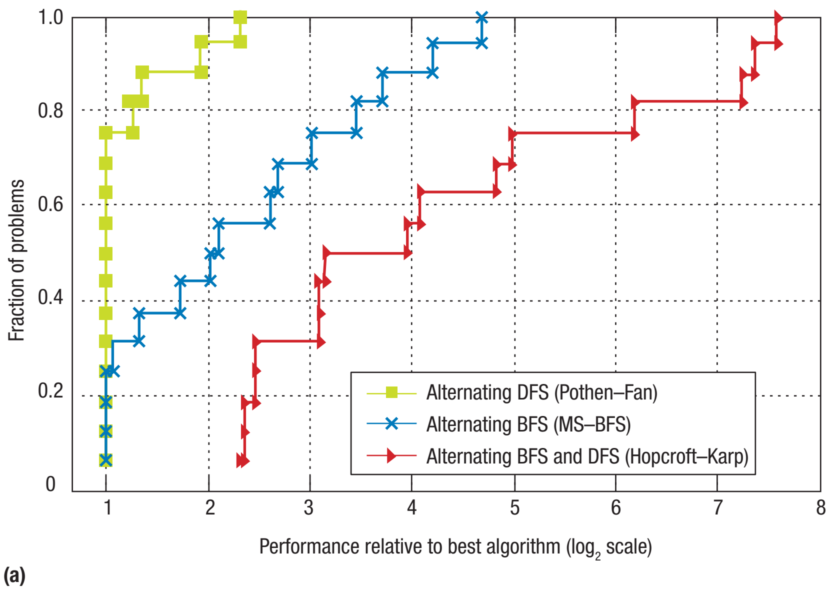


T_1
T_2
x_1 x_2 x_3 x_4 x_5 x_6 x_7
y_1 y_2 y_3 y_4 y_5 y_6 y_7
(a)
(b)
(c)

FIGURE 3. Algorithm search style. (a) For the initial matching, two threads (T_1 and T_2) are used to augment a maximal matching. (b) Another approach uses alternating depth-first search (DFS) to spawn separate threads for each tree. (c) In contrast, alternating breadth-first search (BFS)–based algorithms can use all threads to explore the neighborhood of vertices in the current level. They then synchronize the threads before proceeding to the next level of the BFS forest.

LESSON 3: SEARCH STYLE

Augmentation-based algorithms search for augmenting paths and augment the current matching until such paths no longer exist. Augmenting paths can be constructed using either an alternating BFS or an alternating depth-first search (DFS), starting either from a single vertex or from all unmatched vertices. Figure 3 illustrates the construction of these paths using alternating BFS and alternating DFS from multiple source vertices. Alternating BFS and DFS means that we search for all neighbors at every other level and then search for a matched edge in the other levels. Because starting from a single vertex has limited concurrency, we did not give it further consideration. The searches



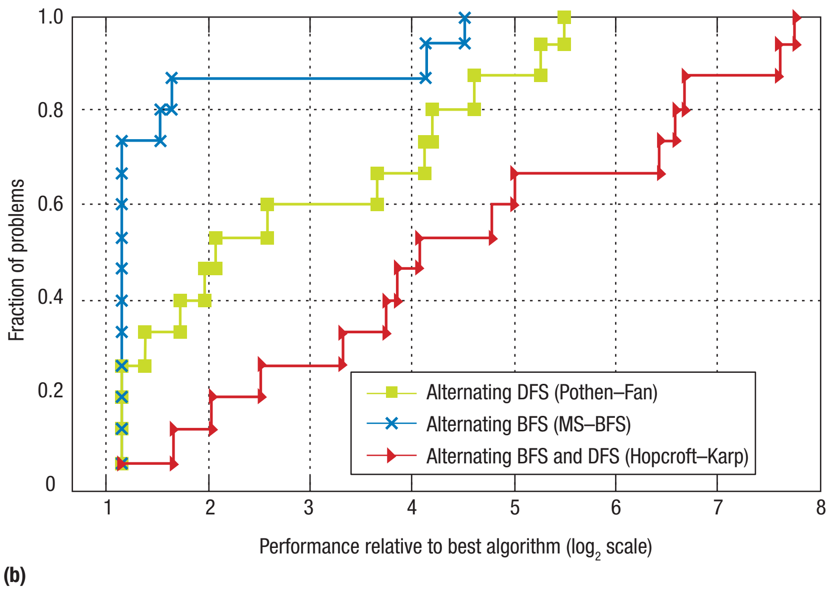


FIGURE 4. Performance profiles of the three best algorithms for maximum-cardinality matching: (a) AMD Opteron and (b) Cray XMT. The results are shown for 16 graphs. The fraction of input problems is plotted on the y-axis, and the runtime relative to the best performing algorithm (in $\log_2$ scale) is plotted on the x-axis. The Pothen–Fan algorithm performed best on the Opteron, and an alternating BFS algorithm was the winner on the XMT.

from multiple source vertices have to be vertex-disjoint in order to find multiple augmenting paths. However, choosing between BFS and DFS approaches is a key decision that is influenced by architectural features and the characteristics of the graph we are matching.

In our work with these approaches,[22] we found that DFS-based approaches perform better on traditional multicore platforms, but BFS-based approaches are faster on the XMT, which has a slower clock and massive numbers of threads. Figure 4 provides our results using 16 graphs on the Opteron and XMT. Although some BFS-based approaches (such as Hopcroft–Karp) have superior worst-case time complexity, the DFS-based approaches emerged as practical winners across multiple studies conducted on large sets of graphs. Hence, we found that the worst-case computational complexity of matching algorithms is not a good predictor of parallel performance. Hannah Bast and colleagues have shown that $O(m \log n)$ is the expected time complexity for maximum cardinality matching in Erdos–Renyi random graphs with specified average degree.[23] (See earlier work for further details.[24])

In our work on bipartite maximum matching, we implemented five augmentation-based algorithms in conjunction with two initialization schemes on multicore and XMT platforms[12] and the Push–Relabel algorithm with several heuristics on multicore platforms.[25] We adapted several lessons from the extensive study in which Iain Duff and his colleagues compared serial algorithms.[4] We implemented a pure alternating BFS-based approach and two pure DFS-based approaches, one of which is Pothen–Fan. The remaining two schemes were hybrid breadth-first and depth-first approaches: parallel Hopcroft–Karp and parallel relaxed Hopcroft–Karp. The Hopcroft–Karp implementations involve the elaborate construction of level graphs to track the multiple vertex-disjoint augmenting paths. In the first part, alternating BFSs are initiated from multiple unmatched vertices leading to the construction of a level graph ending at the discovery of the first unmatched vertex. Subsequently, DFSs are initiated from the other end to find vertex-disjoint augmenting paths in the level graph. The relaxed version of the algorithm comes from relaxing the restriction that the augmenting paths should be the shortest in the original Hopcroft–Karp algorithm.

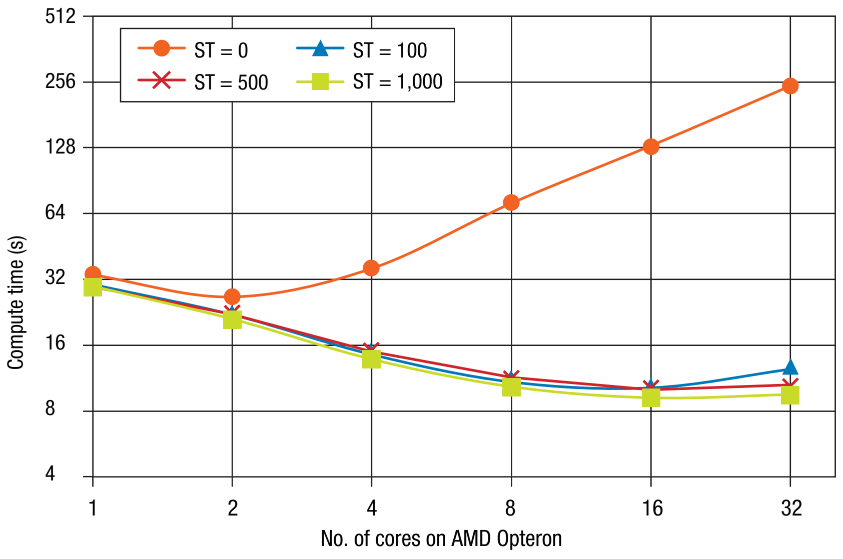


FIGURE 5. Runtimes of the Push–Relabel algorithm for maximum-cardinality matching. We averaged 36 input graphs run on the AMD Opteron system. ST indicates the number of remaining unmatched vertices when the switch to serial computing was made.

Given the overhead required to construct the level graphs, it is no surprise that the Hopcroft–Karp algorithms were less competitive. Because of its simplicity and efficient techniques, the parallel Pothen–Fan algorithm emerged as a clear winner for both serial and parallel approaches on multicore platforms with a modest number of cores. Our Pothen–Fan algorithm implementation used DFS to construct vertex-disjoint alternating-path search trees using the look-ahead technique—that is, it performed one level of BFS at every alternate level to identify unmatched vertices. We further implemented the fairness technique from the study by Iain Duff and his colleagues[4] that alternates the direction in which the adjacency lists of matched vertices are explored in every other iteration.

In addition to multithreading, the hardware feature most relevant to optimal algorithms is the availability of a large shared address space for multithreaded platforms. The design and development of efficient graph algorithms for distributed-memory platforms is a difficult problem, especially for algorithms with complex and synchronized search patterns. Graphs arising in modern data-intensive applications such as social networks and data mining are difficult to partition among the processors while reducing communications. Partitioning such graphs on distributed-memory platforms leads to communication bottlenecks, and on multithreaded platforms with nonuniform memory access (NUMA) costs, it can lead to significant performance losses. The Cray XMT addresses this problem with massive multithreading and by having several memory requests in flight during a clock cycle. However, on modern microprocessor architectures, it is necessary to develop novel schemes for latency tolerance beyond multithreading, and, when possible, programmers should carefully design data structures and memory layout schemes to exploit spatial and temporal localities.[26]

LESSON 4: SERIALIZATION AND SENSITIVITY

We now turn our attention to a nonaugmentation-based approach for maximum matching in bipartite graphs and the parallel Push–Relabel algorithm. This algorithm intuitively provides an estimate of the nearest unmatched vertex from a given vertex by maintaining distance labels. The distances provide a lower bound on the length of an alternating path from a given vertex to its nearest unmatched vertex. In our implementation,[25] we adapted techniques from a serial implementation by Kamer Kaya and his colleagues[27] to a parallel context. We compared the performance of this algorithm with the Pothen–Fan algorithm on several multicore platforms. Some of the algorithmic ideas showcased by the Push–Relabel algorithm are also relevant to other algorithms for weighted matching and network flow. Of particular relevance are the heuristics that determine the relabeling frequency, queuing of active vertices, and switch to serial computation.

A salient feature of most optimal matching algorithms is the rapid decrease in available concurrent work as the algorithm progresses. Toward the end of execution, there is a need to process long augmenting paths (longer distances in the Push–Relabel algorithm) that do not have enough parallel work, which reveals the need for fast serial performance. In the case of the Push–Relabel algorithm, we demonstrate the benefit of switching to serial computation at different stages of the execution. Figure 5 shows the average runtime over 36 input graphs and the effect of switching to serial code toward the end of the algorithm. (Additional details are available in earlier work.[25])

Fast serial performance is thus an important hardware feature that significantly benefits the overall graph algorithm performance. Therefore, we believe that emerging heterogeneous architectures that combine different classes of compute units sharing an address space will be important for irregular (graph) algorithms. We aim to explore these architectures for matching algorithms in the near future.

In a serial context, Duff and his colleagues observed that the runtimes of matching algorithms varied significantly when they randomly permuted the vertices and neighbor lists.[4] This raised serious doubts about the utility of parallel algorithms and implementations for maximum matching. Enforcing vertex ordering in a parallel context leads to serialization and lost efficiency. We therefore studied the parallel sensitivity of different algorithms to prove their effectiveness. We defined an algorithm's parallel sensitivity as the ratio of the standard deviation of runtimes from several runs to the mean of runtimes multiplied by 100. Key observations from our work were that alternating DFS-based algorithms were more sensitive than alternating BFS-based algorithms (up to a 25 percent variance for DFS compared with up to a 5 percent variance for BFS algorithms), and fairness that helped stabilize serial algorithms made parallel algorithms more sensitive.

The lessons from our work transcend the matching problem itself and can be applied to several other combinatorial problems and extended to distributed-memory platforms. For instance, we have successfully adapted some of the ideas such as dataflow principles in designing efficient vertex coloring of graphs.[28] The key insight from our work highlights the need for the careful design of algorithms influenced by architectural features and graph characteristics.

A topic of emerging importance that we did not discuss here is designing algorithms for optimal energy and power consumption. This is a relatively new research topic for irregular algorithms that will become important for modern low-power architectures.[29]

ACKNOWLEDGMENTS

This work was supported in part by the US Department of Energy (the CSCAPES Institute DE-FC02-08ER25864, DE-FG02-13ER26135, and DE-AC02-05CH11231), the US National Science Foundation (CCF-1218916), an IBM Fellowship, and the Center for Adaptive Supercomputing Software (CASS) at Pacific Northwest National Laboratory (PNNL). PNNL is operated by Battelle for the US Department of Energy under contract DE-AC05-76RL01830.

REFERENCES

1. M. Brady et al., "The Assignment Problem on Parallel Architectures," *Network Flows and Matching: First DIMACS Implementation Challenge*, D.S. Johnson and C.C. McGeoch, eds., Am. Mathematical Soc., 1993, pp. 469–517.
2. M. Halappanavar et al., "Approximate Weighted Matching on Emerging Manycore and Multithreaded Architectures," *Int'l J. High Performance Computing Applications*, vol. 26, no. 4, 2012, pp. 413–430.
3. A. Pothen and C.-J. Fan, "Computing the Block Triangular Form of a Sparse Matrix," *ACM Trans. Mathematical Software*, vol. 16, no. 4, 1990, pp. 303–324.
4. I.S. Duff, K. Kaya, and B. Uçar, "Design, Implementation, and Analysis of Maximum Transversal Algorithms," *ACM Trans. Mathematical Software*, vol. 38, no. 2, 2012, article no. 13.
5. A. Khan et al., "A Multithreaded Algorithm for Network Alignment via Approximate Matching," *Proc. Int'l Conf. High Performance Computing, Networking, Storage and Analysis* (SC 12), 2012, article no. 64.
6. A. Nawab et al., "Tolerating Correlated Failures for Generalized Cartesian Distributions via Bipartite Matching," *Proc. 8th ACM Int'l Conf. Computing Frontiers* (CF 11), 2011, article no. 36.
7. Z. Zhu et al., "Fully Programmable and Scalable Optimal Switching Fabric for Petabyte Data Center," *Optics Express*, vol. 23, no. 3, 2015, pp. 3563–3580.
8. S. Belongie, J. Malik, and J. Puzicha, "Shape Matching and Object Recognition Using Shape Contexts," *IEEE Trans. Pattern Analysis and Machine Intelligence*, vol. 24, 2002, pp. 509–522.
9. K.M. Choromanski, T. Jebara, and K. Tang, "Adaptive Anonymity via *b*-Matching," *Advances in Neural Information Processing Systems*, C.J.C. Burges et al., eds., 2013, pp. 3192–3200.
10. P. Medvedev and M. Brudno, "Maximum Likelihood Genome Assembly," *J. Computational Biology*, vol. 16, no. 8, 2009, pp. 1101–1116.
11. S. Pyne et al., "Automated High-Dimensional Flow Cytometric Data Analysis," *Proc. Nat'l Academy of Sciences*, vol. 106, no. 21, 2009, pp. 8519–8524.
12. M. Halappanavar, "Algorithms for Vertex-Weighted Matching in Graphs," PhD dissertation, Dept. of Computer Science, Old Dominion Univ., 2009.
13. G. Karypis and V. Kumar, "Parallel Multilevel *k*-Way Partitioning Scheme for Irregular Graphs," *Proc. Int'l Conf. High Performance Computing, Networking, Storage and Analysis* (SC 96), 1996, article no. 35.
14. L. Lovasz, *Matching Theory*, Elsevier Science, 1986.
15. A. Schrijver, *Combinatorial Optimization: Polyhedra and Efficiency* (*Paths, Flows and Matchings*), vol. A, Springer, 2003.
16. W.J. Cook et al., *Combinatorial Optimization*, John Wiley and Sons, 1998.

17. R. Preis, "Linear Time 1/2-Approximation Algorithm for Maximum Weighted Matching in General Graphs," *Proc. 16th Ann. Conf. Theoretical Aspects of Computer Science* (STACS 99), 1999, pp. 259–269.
18. F. Manne and R.H. Bisseling, "A Parallel Approximation Algorithm for the Weighted Maximum Matching Problem," *Proc. 7th Int'l Conf. Parallel Processing and Applied Mathematics* (PPAM 07), 2007, pp. 708–717.
19. F. Manne and M. Halappanavar, "New Effective Multithreaded Matching Algorithms," *Proc. IEEE 28th Int'l Parallel and Distributed Processing Symp.* (IPDPS 14), 2014, pp. 519–528.
20. D. Chakrabarti and C. Faloutsos, "Graph Mining: Laws, Generators, and Algorithms," *ACM Computing Surveys*, vol. 38, no. 1, 2006, article no. 2.
21. J. Feo et al., "ELDORADO," *Proc. 2nd Conf. Computing Frontiers* (CF 05), 2005, pp. 28–34.
22. A. Azad et al., "Multithreaded Algorithms for Maximum Matching in Bipartite Graphs," *Proc. IEEE 26th Int'l Parallel and Distributed Processing Symp.* (IPDPS 12), 2012, pp. 860–872.
23. H. Bast et al., "Matching Algorithms are Fast in Sparse Random Graphs," *Proc. 21st Ann. Symp. Theoretical Aspects of Computer Science* (STACS 04), LNCS 2996, 2004, pp. 81–92.
24. A. Azad, A. Buluc, and A. Pothen, "A Parallel Tree Grafting Algorithm for Maximum Cardinality Matching in Bipartite Graphs," *Proc. IEEE 29th Int'l Parallel and Distributed Processing Symp.* (IPDPS 15), 2015, pp. 1075–1084.
25. J. Langguth et al., "On Parallel Push–Relabel Based Algorithms for Bipartite Maximum Matching," *Parallel Computing*, vol. 40, no. 7, 2014, pp. 289–308.
26. D. Chavarría-Miranda, M. Halappanavar, and A. Kalyanaraman, "Scaling Graph Community Detection on the Tilera Many-Core Architecture," *Proc. IEEE Int'l Conf. High Performance Computing* (HiPC 14), 2014, pp. 1–11.
27. K. Kaya et al., "Push–Relabel Based Algorithms for the Maximum Transversal Problem," *Computers & Operations Research*, vol. 40, no. 5, 2013, pp. 1266–1275.
28. U. Catalyurek et al., "Graph Coloring Algorithms for Multi-core and Massively Multithreaded Architectures," *Parallel Computing*, vol. 38, nos. 10-11, 2012, pp. 576–594.
29. D. Chavarría-Miranda et al., "Optimizing Irregular Applications for Performance and Energy on the Tilera Many-Core Architecture," *Proc. 12th ACM Int'l Conf. Computing Frontiers* (CF 15), article no. 12.

ABOUT THE AUTHORS

MAHANTESH HALAPPANAVAR is a staff scientist in the Fundamental and Computational Sciences Directorate at the Pacific Northwest National Laboratory. His research interests include the interplay of algorithm design, architectural features, and input characteristics targeting massively multithreaded architectures such as the Cray XMT and emerging multicore and many-core platforms. Halappanavar received a PhD in computer science from Old Dominion University. Contact him at hala@pnnl.gov.

ALEX POTHEN is a professor of computer science at Purdue University. His research interests include combinatorial scientific computing, parallel computing, computational science and engineering, and bioinformatics. Pothen received a PhD in applied mathematics from Cornell University. He is an editor of the *Journal of the ACM* and *SIAM Review*. Contact him at apothen@purdue.edu.

ARIFUL AZAD is a postdoctoral fellow in the Computational Research Division at Lawrence Berkeley National Laboratory. His research interests include parallel computing, computational science and engineering, and bioinformatics. Azad received a PhD in computer science from Purdue University. Contact him at azad@lbl.gov.

FREDRIK MANNE is a professor of computer science at the University of Bergen, Norway. His research interests include parallel algorithms for combinatorial scientific computing. Manne received a PhD in computer science from the University of Bergen. Contact him at manne@ii.uib.no.

JOHANNES LANGGUTH is a postdoctoral fellow at the Simula Research Laboratory in Oslo, Norway. His research interests include computer architecture, parallel algorithms, combinatorial optimization, and high-performance scientific computing on multicore CPUs and GPUs. Langguth received a PhD in computer science from the University of Bergen. Contact him at langguth@simula.no.

ARIF KHAN is a doctoral student in the Department of Computer Science at Purdue University. His research interests includes graph algorithms, bioinformatics, and parallel and high-performance computing. Khan received an MS in computer science from the University of Florida. Contact him at khan58@purdue.edu.