Multithreaded Algorithms for Maximum Matching in Bipartite Graphs

Ariful Azad[1], Mahantesh Halappanavar[2], Sivasankaran Rajamanickam[3], Erik G. Boman[3], Arif Khan[1], and Alex Pothen[1],

E-mail: {aazad,khan58,apothen}@purdue.edu, mahantesh.halappanavar@pnnl.gov, and {srajama,egboman}@sandia.gov

[1] Purdue University [2] Pacific Northwest National Laboratory [3] Sandia National Laboratories

Abstract*—We design, implement, and evaluate algorithms for computing a matching of maximum cardinality in a bipartite graph on multicore and massively multithreaded computers. As computers with larger numbers of slower cores dominate the commodity processor market, the design of multithreaded algorithms to solve large matching problems becomes a necessity. Recent work on serial algorithms for the matching problem has shown that their performance is sensitive to the order in which the vertices are processed for matching. In a multithreaded environment, imposing a serial order in which vertices are considered for matching would lead to loss of concurrency and performance. But this raises the question: *Would parallel matching algorithms on multithreaded machines improve performance over a serial algorithm?

We answer this question in the affirmative. We report efficient multithreaded implementations of three classes of algorithms based on their manner of searching for augmenting paths: breadth-first-search, depth-first-search, and a combination of both. The Karp-Sipser initialization algorithm is used to make the parallel algorithms practical. We report extensive results and insights using three shared-memory platforms (a 48-core AMD Opteron, a 32-core Intel Nehalem, and a 128-processor Cray XMT) on a representative set of real-world and synthetic graphs. To the best of our knowledge, this is the first study of augmentation-based parallel algorithms for bipartite cardinality matching that demonstrates good speedups on multithreaded shared memory multiprocessors.

I. Introduction

We design, implement, and evaluate five parallel algorithms for computing a matching of maximum cardinality in a bipartite graph on multicore and massively multithreaded computers. As multicore machines dominate the commodity processor market, and the size of problems continue to increase, there is need for parallel algorithms for solving important combinatorial problems such as matching in graphs. However, achieving good performance and speedup on combinatorial problems is a challenge due to the large number of data accesses relative to computation, the poor locality of the data accesses, irregular nature of the concurrency available, and fine-grained synchronization required. An earlier study on parallel (weighted) matching algorithms [1] reported poor speedups, but said ideal platforms for such algorithms would have "relatively few numbers of powerful processors, support for block transfers, lots of memory, and a high processor-memory bandwidth to this memory". We show that good speedups are achievable on emerging multicore machines via shared memory multithreaded algorithms that are designed here.

Matching has several applications in computer science, scientific computing, bioinformatics, information science, and other areas. Our study of bipartite maximum matching is motivated by applications to solving sparse systems of linear equations, computing a decomposition known as the block-triangular form (BTF) of a matrix [2], and aligning graphs or networks.

The rest of this paper is organized as follows. In Section II we describe three classes of parallel algorithms for computing maximum matchings in bipartite graphs, and the initialization algorithms used in this context. A short section on related work is included in Section III. The next Section IV describes the three machines we implement our parallel algorithms on. The final Section V describes an extensive set of computational experiments and results on a test set of sixteen problems.

Our major contributions include the following: To the best of our knowledge, this is the first work to provide speedups for (cardinality) matching algorithms on multithreaded platforms; this is the first description of three classes of augmentation-path based multithreaded parallel matching algorithms; this is also the first description of a multithreaded Karp-Sipser matching initialization algorithm, which computes maximum or close to maximum matchings on many problems; we provide insights by comparing the different algorithm classes across multithreaded architectures and input graph characteristics; and from a study of a massively multithreaded multiprocessor, we are able to provide insight into performance on newly emerging architectures.

II. Algorithms for Maximum Matching in Bipartite Graphs

In this section we describe five parallel algorithms for computing maximum cardinality matchings in bipartite graphs. All of them compute matchings by finding at each iteration of the algorithm a set of vertex-disjoint augmenting paths in parallel. They differ in the nature of the set of augmenting paths, how the augmenting paths are computed, and what graph is used to compute the augmenting paths. We provide descriptions of two parallel algorithms in some detail, and briefly sketch how the other algorithms are related to them.

We need some definitions before we can describe the algorithms. Given a graph $G = (V, E)$, a matching $M \subseteq E$ is a set of edges such that no two edges in M are incident on

the same vertex. The maximum matching problem is one of maximizing the number of edges in M. In this paper, we focus on matchings in bipartite graphs, $G = (X \cup Y, E)$, where the vertex set $V = X \cup Y$ is partitioned into two disjoint sets such that every edge connects a vertex in X to a vertex in Y. We denote $|V|$ by n and $|E|$ by m. Given a matching M in a bipartite graph $G = (V, E)$, an edge is matched if it belongs to M, and unmatched otherwise. Similarly, a vertex is matched if it is the endpoint of a matched edge, and unmatched otherwise. An alternating path in G with respect to a matching is a path whose edges are alternately matched and unmatched. An augmenting path is an alternating path which begins and ends with unmatched edges. By exchanging the matched and unmatched edges on this path, we can increase the size of the matching by one. We refer the reader to a book on matching algorithms [3] for additional background on matching.

The first algorithm we describe, Parallel Pothen-Fan (PPF+), is derived from a serial algorithm with $O(nm)$ time complexity proposed and implemented in 1990 by Pothen and Fan [2], which is currently among the best practical serial algorithms for maximum matching. The second algorithm, Parallel Hopcroft-Karp (PHK), is based on a serial algorithm proposed by Hopcroft and Karp [4] in 1973, with asymptotic time complexity $O(n^{1/2}m)$. In the next two subsections we describe five parallel algorithms for finding a maximal set of vertex-disjoint augmenting paths using DFS or BFS or both. These algorithms are listed in Table I, where we also include two initialization algorithms (Greedy and Karp-Sipser) that are used to make the exact matching algorithms faster.

Class	Parallel Algorithms	Serial Complexity
DFS based	Vertex-disjoint DFS (PDDFS)	$O(nm)$
	Pothen-Fan (DFS with lookahead) (PPF+)	$O(nm)$
BFS based	Vertex-disjoint BFS (PDBFS)	$O(nm)$
BFS and DFS based	Hopcroft-Karp (PHK)	$O(\sqrt{n}m)$
	Relaxed Hopcroft-Karp (PRHK)	$O(\sqrt{n}m)$
1/2-approx	Greedy Maximal (PG)	$O(m)$
	Karp-Sipser (PKS)	$O(m)$

TABLE I
SUMMARY OF ALGORITHMS IMPLEMENTED.

We point out that the graph searches for augmenting paths have a structure different from the usual graph searches: if the search is begun from an unmatched vertex in X, when a matched vertex $v \in Y$ is reached in a search, the only vertex we reach from v is the *mate* of v, the vertex matched to v. The search for all neighbors continues from the mate, which is again a vertex in X.

A. DFS-based Algorithms

In this subsection, we describe the PPF+ algorithm, whose pseudo-code is provided in Algorithm 1. The algorithm begins with an initial matching obtained from an initialization algorithm that will be discussed in a following subsection. The algorithm makes use of DFS with lookahead to compute, at each iteration, a maximal set of vertex-disjoint augmenting paths. (A maximal set with respect to a property is one to which we cannot add elements and maintain the property; it is not necessarily a set of maximum cardinality with the property.) A maximal set of vertex-disjoint augmenting paths can be discovered in $O(m)$ time by doing DFS's from each unmatched vertex in the set X (or Y), and the matching can be augmented by several edges at a time.

The idea of the lookahead mechanism in DFS is to search for an unmatched vertex in the adjacency list of a vertex u being searched before proceeding to continue the DFS from one of u's children. If the lookahead discovers an unmatched vertex, then we obtain an augmenting path and can terminate the DFS. If it is not found, the lookahead does not add significantly to the cost of the DFS, since it can be implemented in $O(m)$ time for the entire algorithm. Intuitively, lookahead is doing one level of BFS from a vertex before continuing with the DFS. In practice, Pothen and Fan found that lookahead helps a DFS-based algorithm finds shorter augmenting paths faster, and led to a serial algorithm for matching that was practically faster than the Hopcroft-Karp algorithm, in spite of the latter's superior asymptotic time complexity.

We call each iteration of **repeat until** block in Algorithm 1 a *phase* of the algorithm. Each phase is executed in parallel by spawning a set of threads. Each thread begins a DFS with lookahead from a currently unmatched vertex to search for an augmenting path. As each augmenting path is found, each thread augments the current matching, and then proceeds to search for an augmenting path from the next available unmatched vertex in the phase. The augmenting paths found in a phase need to be vertex-disjoint so that the matching at the beginning of a phase can be augmented using all of them. One way to enforce the vertex-disjointedness constraint is to let all threads do their DFS's without synchronization and synchronize when we update the matching. This could cause many of the augmenting paths to be discarded since after augmentation by one of the paths, the matching changes, and the other augmenting paths might no longer remain augmenting with respect to the current matching. The discarded work causes the total work in the parallel version to be significantly greater than in the serial version. Thus, a better approach is to make sure that independent DFS traversals in an iteration do not visit the same vertices, which is accomplished by a thread-safe implementation of DFS with lookahead that guarantees the vertex-disjoint property. While each DFS is inherently sequential, there are many DFS traversals that can be performed simultaneously in parallel.

From one iteration to the next, the direction in which the adjacency list is searched for unmatched vertices can be switched from the beginning of the list to the end of the list, and vice versa. Duff et al. [5] call this *fairness*, and found that this enhancement leads to faster execution times. In the PPF+ algorithm we have employed fairness.

Algorithm 2 describes the pseudo-code for the multithreaded DFS with lookahead, DFS-LA-TS. The algorithm has

two steps, a lookahead step, and a DFS step.

The lookahead step makes uses of a pointer $lookahead[u]$ to the next unseen vertex in the adjacency list of u. The lookahead pointer ensures that the entire adjacency list of u is scanned only once in all the phases during the lookahead step. The **for all** loop (line 2) searches the unscanned vertices in the adjacency list of the vertex u for an unmatched vertex. If it finds such a vertex, the algorithm returns the unmatched vertex found. If all vertices in the adjacency set of u are matched, then the algorithm proceeds to the DFS step, by executing the second **for all** loop (line 7), and in that loop, making a recursive call to the algorithm.

The vertex-disjoint property of the set of augmenting paths is achieved by using *atomic memory operations* for updating the entry for each vertex in a $visited$ array. This ensures that only the first thread to reach a vertex can use it in an augmenting path. As a generic operator `Atomic_Fetch_Add` performs the requested addition operation in a thread-safe (atomic) manner, and returns the original value prior to the requested arithmetic operation. The operator ensures that only one thread reads the original value of 0, thus indicating that this is the first thread to read it; all other threads will see a nonzero value, which indicates that the variable has already been accessed by another thread.

Even though DFS-LA-TS is expressed as a recursive algorithm for brevity in describing it here, we use a $stack$ mechanism (an array of size at most n, where n is the number of vertices) within each thread to implement the DFS. Note that in spite of the stack being private to each thread we do not require an array of size n for each thread. In any phase of the matching algorithm, no vertex will get visited more than once, and so it cannot be in the stack of multiple threads. The amount of memory required for the stack of all the threads is at most n. We can resize our stack dynamically to achieve this. We should note that when memory is not a bottleneck with a sufficiently small number of threads, the implementation with a stack of size n for *each* thread does perform better and we use this for our performance results. The lookahead is implemented with an array of size $O(n)$. The cost of the lookahead is $O(m)$ for the entire algorithm as in the serial case. We maintain the maximum matching as an array of size n. As the paths are vertex disjoint, we can augment the matching in parallel as no two threads will touch the same entries of the matching array.

The parallel disjoint DFS (PDDFS) matching algorithm is similar to the PPF+ algorithm, but it does not make use of the lookahead mechanism in the latter. In each phase of the PDFS algorithm, we do a a DFS traversal from each unmatched vertex and enforce the vertex-disjoint property of augmenting paths as in the PPF+ algorithm. The call to DFS-LA-TS in the PPF+ algorithm is replaced here by DFS-TS. The pseudo-code for DFS-TS is presented in Algorithm 3.

B. BFS-based Algorithms

The Hopcroft-Karp algorithm [4] finds a *maximal* set of *shortest* vertex-disjoint augmenting paths and augments along each path simultaneously. They showed that by doing so, the number of augmentation phases could be bounded by $O(n^{1/2})$ thereby providing faster asymptotic time complexity than an algorithm that augments the matching by one augmenting path at a time. The other major difference is that the Hopcroft-Karp (HK) algorithm uses a breadth-first search (BFS) to construct a layered graph, and the maximal set of augmenting paths are found within this layered graph. The BFS stops at the first level where an unmatched vertex is discovered, as we are looking for shortest augmenting paths. The BFS does not return the augmenting paths, instead it returns the layered graph. In a second step, the HK algorithm uses a vertex disjoint DFS traversal on the layered graph from all the unmatched vertices in the last level of the BFS to find the maximal set of shortest length vertex-disjoint augmenting paths.

The layered graph is defined as follows. Let M be a current matching in a bipartite graph $G(X \cup Y, E)$, and let $X_0 \subseteq X$ and $Y_0 \subseteq X$ be the set of unmatched vertices. The layered graph $LG(L, E_{LG}, K)$ with respect to a matching M is a subgraph of G whose vertex set is arranged into levels $L_0, L_1, ...L_K$, with $L_0 = X_0$; $L_K \subseteq Y_0$. The level L_1 is the set of vertices in Y adjacent to vertices in L_0; if there are unmatched vertices in L_1, then $K = 1$. If not, L_2 is the set of vertices in X that are matched to vertices in L_1. More generally, for even i, L_i consists of vertices in X matched to vertices in L_{i-1}; for odd i, L_i consists of all vertices in Y adjacent to vertices in L_{i-1} that do not belong to earlier levels. In the last layer L_K (K is odd, and these vertices belong to Y) we need keep only the unmatched vertices since we use the layered graph to find augmenting paths beginning from vertices in L_K. The edges in the layered graph, E_{LG}, join a vertex in one layer to a vertex in the next layer; other edges of G can be discarded.

The parallel Hopcroft-Karp (PHK) algorithm is described in Algorithm 4. It first constructs the layered graph in parallel by simultaneous BFS's from the unmatched vertices in one side (the vertex set X here). The details of this construction will be discussed later. In the second step, it uses Algorithm 3 to do parallel disjoint DFS searches from the unmatched vertices in the last layer of the layered graph to find vertex-disjoint augmenting paths. As before, we call each iteration of the parallel **repeat-until** block a phase of the PHK algorithm. Since the augmenting paths are vertex disjoint, the matching at the beginning of a phase can be augmented by all of the augmenting paths discovered during this phase.

The parallel BFS-based layered graph construction is described in Algorithm 5. It constructs the layered graph two levels at a time. The next level L_{k+1} consists of all vertices in Y not belonging to earlier levels that can be reached from vertices in level L_k. If there is an unmatched vertex in level L_{k+1}, we are done. Otherwise, the following level L_{k+2} consists of all vertices in X that are matched to vertices in L_{k+1}.

Each thread picks a vertex u in the current level L_k and assigns its neighbors to the level L_{k+1} and the vertices matched to the latter to the level L_{k+2}. Each thread creates its local list of vertices in the two succeeding levels. Once all

threads create their local lists, they are concatenated to obtain the levels L_{k+1} and L_{k+2}. Then the threads synchronize, and can begin to construct the next two levels.

As before, we use *atomic memory operations* in our parallel layer construction algorithm (Algorithm 5) to ensure that each vertex is added to only one of the local lists of the levels. We do not need any synchronization while adding a vertex or the corresponding edge to the layered graph. There is one synchronization point per pair of levels in the layered graph in addition to the atomic operations to mark the $visited$ array.

We have chosen to construct the layered graph explicitly in our Hopcoft-Karp algorithms, although since the work of Duff and Wiberg on serial matching algorithms, others have chosen to work with an implicitly defined layered graph that uses the original graph data structures. The explicit construction is more expensive than an implicit representation of the layered graph, but the explicit layered graph speeds up the search for augmenting paths by DFS. We have compared the Hopcoft-Karp algorithms here with others that work with an implicitly defined layered graph, and found that their relative performance is problem-dependent, so there was no clear winner among the two strategies.

We have implemented two additional parallel BFS-based algorithms. One of them is Parallel Relaxed Hopcroft-Karp (PRHK), which continues to construct the layered graph to levels beyond the shortest augmenting path length. Beyond augmentation by a maximal set of vertex-disjoint shortest augmenting paths, we search for longer augmenting paths from unmatched vertices in the relaxed layered graph and use these paths to augment the matching. Another algorithm is based on Parallel Disjoint BFS (PDBFS), which does not use a layered graph construction at all, but performs vertex-disjoint BFS's in the original graph from unmatched vertices in X to find unmatched vertices in Y, and augments the matching directly using these augmenting paths.

C. Parallel Initialization Algorithms

Many iterations of one of the parallel algorithms described earlier might be needed to find a matching of maximum cardinality. The number of iterations needed can be reduced substantially by using an initialization algorithm to compute an initial matching in $O(m)$ time. In many cases, the initialization algorithms are good enough to find all or a large fraction of the maximum cardinality of a matching. Langguth *et al.* [6] experimented with various initialization algorithms, and reported that the Karp-Sipser algorithm was the best performer for quality and fast execution. Duff *et al.* [5] also found that this algorithm is among the best initialization algorithms in terms of the cardinality of the matchings found. An initialization algorithm that finds a larger cardinality matching, usually though not always, leads to faster computation of the maximum cardinality matching.

The Parallel Karp-Sipser algorithm is described in Algorithm 6. All vertices with initial degree one in the set X are stored in Q_1 and processed first in parallel. Degrees of the vertices adjacent to matched Y vertices are updated, and if a new vertex of current degree one in X is discovered, it is processed immediately. After processing Q_1 other unmatched vertices in X from queue Q_* are processed in parallel. The degree update and immediate processing of newly discovered degree one vertices are done synchronously and described in a separate routine MATCHANDUPDATE in Algorithm 7.

Note that, before a degree one vertex u is discovered in dynamic degree update by a thread, a higher degree vertex v can be picked by another thread. Hence the processing order of the parallel implementation can deviate from that of a serial implementation, and consequently when the number of threads increases, the matching size is expected to decrease.

We have also implemented a Greedy initial matching algorithm that finds a maximal matching (i.e., a matching such that its size cannot be increased any further without changing some of its matched edges to unmatched edges). The Parallel Greedy algorithm examines, in parallel, (unmatched) vertices u belonging to one of the vertex sets X, and does an atomic fetch and add operation to lock an unmatched vertex $v \in Y$ in u's adjacency list, and then matches the two vertices u and v. We omit a detailed description.

III. RELATED WORK

The literature on parallel matching algorithms spans several models of parallel computers: parallel random access machine (PRAM), coarse grained multicomputer (CGM), bulk synchronous parallel (BSP), and the massively parallel computer (MPP). However, much of the work is theoretical in nature, e.g., Karpinski and Rytter [7].

Considerable interest in parallel algorithms has been observed in recent time with work on approximation as well as optimal algorithms on shared and distributed memory platforms. Langguth *et al.* [8] describe their work on parallelizing the Push-Relabel algorithm [9] for bipartite maximum matching on a distributed-memory platform. Although they conclude that strong scaling or speedup is difficult to achieve, good parallel performance justifies the effort to parallelize for memory-scaling reasons. Patwary *et al.* [10] have implemented a parallel Karp-Sipser algorithm on a distributed memory machine using an edge partitioning of the graph rather than a vertex partitioning. Setubal [11] has implemented the push-relabel algorithm on a shared memory machine obtaining a speed-up of two to three on twelve processors.

Parallel algorithms for weighted matching have also been studied. There is an extensive literature on parallelization of auction algorithms [12], [13], [14]. A general conclusion from the work on auction algorithms is that they are suitable for large dense problems. Recently Sathe and Schenk have reported speedups of 32 on a 1024 processor Cray XE6 for a parallel auction algorithm [15].

Several experimental results were presented as part of the first DIMACS Implementation Challenge held in 1990—1991. The results showed performance of augmenting-path based as well as auction based algorithms on platforms such as WaveTracer Zephyr, MasPar MP-1, and Silicon Graphics IRIS 4D/340 DVX. Of particular interest are the results of Brady *et*

al. [1], who compare the performance of auction algorithms on the different parallel architectures mentioned above. Brady *et al.* describe their results as a "little short of disastrous". However, we believe their conclusions on an "ideal" platform for auction-like algorithms still hold: "relatively few numbers of powerful processors, support for block transfers, lots of memory and a high processor-memory bandwidth to this memory". After two decades, modern multicore processors feature some of these characteristics.

Unlike optimal algorithms, approximation algorithms for matching are more amenable to parallelization. In particular, half-approximation algorithms for weighted matching have been explored by Manne and Bisseling [16], Halappanavar [17], and Çatalyürek *et al.* [18] on distributed-memory platforms. Speedups on up to $16,000$ processors for a class of graphs was demonstrated by Catalyürek *et al.* Recently, Halappanavar *et al.* [19] explored the half-approximation algorithm on several multithreaded platforms including general purpose graphics processing units. They demonstrated good speedups for several classes of graphs.

In designing our parallel algorithms we have started from the best implementations of serial algorithms. From this perspective, Duff *et al.* [5] is the closest to our work. In addition to introducing a new variant algorithm, Duff *et al.* conducted an extensive empirical study of bipartite maximum matching algorithms in serial. They compared eight algorithms based on augmenting paths, ranging from simple BFS and DFS based algorithms to sophisticated Pothen-Fan (PF), Hopcroft-Karp (HK) and their variants. They used three different methods for greedy initializations: simple greedy, Karp-Sipser, and minimum degree. They implemented all these algorithms consistently in a single library. Duff *et al.* found that overall PF with fairness (PF+) and Hopcoft-Karp with the Duff-Wiberg modification (HKDW) were the two best methods. Duff *et al.* also studied the effect of random permutations of the input, and observed that this may significantly impact performance. They point out in the conclusion section that this poses a great challenge for parallel matching algorithms, since parallel tasks may be executed in non-deterministic order.

IV. Experimental Platforms

The three platforms selected for this study—Opteron, Nehalem and XMT—represent two state-of-the-art multicore architectures as well as a non-traditional massively multithreaded architecture. Together, the platforms represent a broad spectrum of capabilities with respect to clock speeds, ranging from 0.5 GHz (XMT) to about 2.6 GHz (Nehalem); hardware multithreading, ranging from none (Opteron) to 128 threads per processor (XMT); cache hierarchies, ranging from none (XMT) to three levels (Nehalem and Opteron); and memory interconnects, ranging from DDR1 (XMT) to DDR3 (Nehalem). The XMT is cache-less, and uses massive multithreading to tolerate the large latency to memory by having some threads that are ready to execute while others wait for outstanding memory accesses. The Nehalem and the Opteron depend on moderate multithreading, cache hierarchies and lower latencies to memory.

A few key features of the three platforms and references to detailed information are provided in Table II. For the Opteron, while the theoretical peak bandwidth to memory is 42.7 GB/s, speeds in practice are limited to 28.8 GB/s due to the speed of Northbridge links. The base clock frequency of Nehalem is 2.266 GHz, and the maximum turbo frequency is 2.666 GHz.

The parallel algorithms were implemented on the Nehalem and the Opteron using OpenMP primitives, and on the Cray XMT using pragmas specific to the XMT. Additional implementation details will be mentioned in the next section.

V. Experimental Results

We present experimental results in this section on a set of test problems whose sizes, degree distributions, and descriptions are included in Table III. The problems are divided into two classes: unsymmetric and rectangular problems form the first subset of eight problems, and the remaining problems are symmetric (these are separated by some vertical space). The problems are taken from various sources: the Florida sparse matrix collection, the RMAT graph generator [23] (ER22, good22, bad22), and a graph generated in a network alignment problem by a belief propagation algorithm (scbp). We begin with a discussion of the performance of greedy initialization algorithms. Next we provide experimental results on sensitivity of the performance to the non-determinism in parallel executions, and show that the variance in runtime, while large for some algorithms, is small relative to the speedups that we observe to make experimental comparisons meaningful. We then proceed to present strong scaling results for the best performing algorithms, comparing their performance across algorithms, problem classes, and architectures. We discuss the moderately multithreaded Opteron and Nehalem first, before discussing results on the massively multithreaded XMT.

We now consider some details that are relevant for multithreaded implementations. First, in the DFS-based matching algorithms (which search for multiple augmenting paths in parallel), we used dynamic scheduling for load balancing the threads. In BFS-based matching algorithms (which compute vertices in the layered graph two levels at a time), we used static or block-dynamic scheduling for low overhead costs. The Karp-Sipser initialization algorithm performed best with dynamic scheduling, while the Greedy algorithm did best with static scheduling. Second, threads can be pinned to specified cores to prevent overheads from thread migration. When there are eight cores on a socket, compact thread pinning (filling threads one socket after another) provides good scaling for up to eight threads, whereas scatter thread-pinning (distributing threads across sockets in a round-robin manner) is better for a moderate numbers (up to twelve) of threads. For larger numbers of threads, both perform equally well. We used compact thread pinning in our experiments. Third, two of the machines have non-uniform memory access costs since physically separate memory banks are associated with each socket. Interleaved memory allocation (memory allocated

Platform:	Opteron 6176 SE (Opteron)	Xeon X7560 (Nehalem)	ThreadStorm-2 (XMT)
Processing Units			
Clock (GHz)	2.30	2.266	0.5
Sockets	4	4	128
Cores/socket	12	8	1
Threads/core	1	2	128
Total threads	48	64	16,384
Memory System			
L1 (KB)/core:Inst/Data	64/64	32/32	–
L2 (KB)/core	512	256	–
L3 (MB)/socket	12	24	–
Memory/socket (GB)	64	64	8
Peak Bandwidth (GB/s)	42.7 (DDR3) per socket	34.1 (DDR3) per socket	86.4 (DDR1) per system
Software			
C Compiler	GCC 4.1.2	Intel 11.1	Cray C 6.5.0
Flags	-O3	-fast	-par
Reference	[20]	[21]	[22]

TABLE II
SUMMARY OF ARCHITECTURAL FEATURES OF PLATFORMS USED IN THIS PAPER.

evenly across sockets) yields faster run times for larger numbers of threads, although this can slow down performance on one or two threads.

A. Quality and Impact of Initialization Algorithms

We measure the quality of a matching M obtained by an initialization algorithm by the ratio of the cardinality of the matching $|M|$ to the cardinality of a maximum matching in that graph. A high quality initialization algorithm reduces the work that an exact matching algorithm needs to do, and usually, but not always, reduces the overall runtime. Initialization has an even larger impact on the run time of a parallel algorithm, by reducing the number of iterations that an exact matching algorithm needs to perform.

The quality of the Karp-Sipser initialization is better than the Greedy initialization for all the test problems we have considered. We show results for a representative example, `ER22`, in Figure 1. Karp-Sipser computes matchings that are better than Greedy by 6% for this graph. Since the graph has random connectivity, it has poor spatial and temporal cache locality in the matching computation. The small difference in the quality of the matching decreases run times for the PRHK algorithm (the fastest among our algorithms for this problem) by a factor of four or more, as can be seen from the right figure in Fig. 1. However, for very sparse graphs with good spatial locality, the decrease in run times is not as significant. An example is the `USA-Roadmap` problem, where the Karp-Sipser algorithm obtains an initial matching with cardinality that is 15% greater than the Greedy algorithm, but where the improvement in run times is only a factor of 1.5 for the PPF+ algorithm on an Opteron (again, this is the fastest algorithm for this problem).

Fig. 1 also demonstrates the influence of concurrency on the quality of matchings. The number of threads executing on the XMT is several orders larger than those on AMD, and therefore, we observe that the quality of matching computed on the XMT is lower than that on the AMD for the Karp-Sipser algorithm. Note that we request 100 threads per processor on the XMT, and the runtime determines the number of threads that get allocated. We expect (and observe) twenty to hundred threads executing concurrently over the entire execution in a dynamic fashion. When the number of threads exceeds the number of vertices with current degree one, then vertices of higher degree are matched at random, and consequently the cardinality of matching decreases. The difference in quality is more pronounced for the Karp-Sipser algorithm than for the Greedy matching algorithm.

For all the input graphs in our study, Greedy initialization is faster than the Karp-Sipser algorithm by a factor of two to four. But the total runtime of an exact matching algorithm when initialized with the Karp-Sipser algorithm is almost always smaller than the combination with Greedy initialization due to the improved quality that Karp-Sipser provides. This agrees with earlier findings for initializations in serial matching algorithms [5], [6]. Hence, from now on, we consider only the Karp-Sipser initialization with exact matching algorithms. For many problems, after initialization with Karp-Sipser, only few unmatched vertices remain to match with an exact matching algorithm (Karp-Sipser obtained a maximum matching for 16 out of 100 problems in the test set of [5]). In order to enable a meaningful comparison of exact algorithms, we have chosen input graphs such that after greedy initialization a significant number of unmatched vertices still remains for an exact algorithm to match.

B. Sensitivity of Runtimes

By randomly permuting the vertices, Duff *et al.* [5] have shown that there is a significant variation in the runtimes of serial matching algorithms when unmatched vertices are processed in different orders. This raises a serious issue

Name	$\|X\|$	$\|Y\|$	$\|E\|$	Degree Max.	Degree Avg.	Degree Std. Dev.	Description
amazon0312	400,727	400,727	3,200,440	10	7.99	3.07	Amazon product co-purchasing network
hamrle3	1,447,360	1,447,360	5,514,242	6	3.81	1.55	Circuit simulation matrix
cit-patents	3,774,768	3,774,768	16,518,948	770	4.38	7.78	Citation network among US Patents
scbp	154,974	342,684	20,883,500	38,636	135	441	Network alignment of Library of Congress terms
GL7d19	1,911,130	1,955,309	37,322,725	121	19.5	2.85	Combinatorial problems
ER22 (RMAT)	4,199,304	4,199,304	67,080,546	59	16.0	10.1	Erdos-Renyi Random graph
good22 (RMAT)	4,199,304	4,199,304	67,080,546	965	16.0	23.8	Random power law graph
bad22 (RMAT)	4,199,304	4,199,304	67,080,546	26,832	15.8	87.3	Random power law graph with wider degree distribution
roadNet-CA	1,971,281	1,971,281	5,533,214	12	2.81	1.01	California street networks
kkt_power	2,063,494	2,063,494	12,771,361	96	7.08	7.40	Optimal power flow, nonlinear optimization (KKT)
hugetrace-0000	4,588,484	4,588,484	13,758,266	3	2.99	0.02	Frames from 2D Dynamic Simulations
as-skitter	1,696,415	1,696,415	22,190,596	35,455	13.1	137	Internet topology graph
coPapersDBLP	540,486	540,486	30,491,458	3,299	56.4	66.2	Citation Networks in DBLP
hugetrace-0020	16,002,413	16,002,413	47,997,626	3	2.99	0.02	Frames from 2D Dynamic Simulations
USA-Roadmap	23,947,347	23,947,347	57,708,624	9	2.41	0.93	USA street networks
delaunay_n24	16,777,316	16,777,316	100,663,202	26	6.00	1.34	Delaunay triangulations of random points in the plane

TABLE III
DESCRIPTION OF A SET OF 16 TEST PROBLEMS FROM THE UNIVERSITY OF FLORIDA COLLECTION AND OTHER SOURCES. THE FIRST EIGHT PROBLEMS ARE NONSYMMETRIC OR RECTANGULAR; THE OTHERS ARE SYMMETRIC.

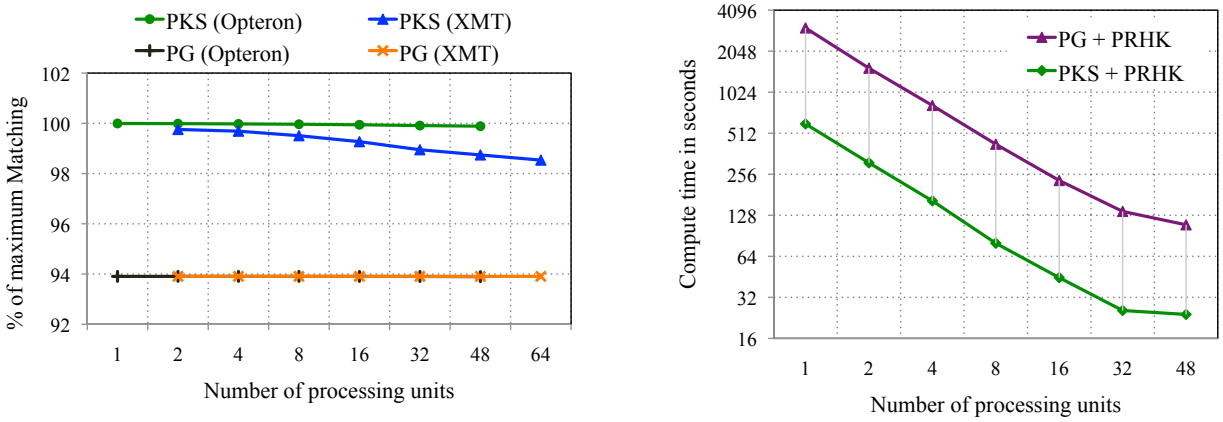


Fig. 1. **Quality of Initial Matchings and Impact on Runtimes:** The subfigure on the left shows the fraction of the maximum matching computed in the initialization step by the Greedy (PG) and Karp-Sipser (PKS) algorithms. Processing units are cores (one thread per core) for AMD Opteron and number of processors (20 to 100 interleaved threads per processor) for Cray XMT. The subfigure on the right shows the run time of the Relaxed Hopcroft-Karp algorithm on an AMD Opteron. Both figures are for the random graph ER22 generated from RMAT.

for parallel algorithms: In a multithreaded context, different executions of an algorithm are likely to process vertices to match in different orders. We cannot assign threads to specific vertices or impose an order in which threads must execute without severe loss of concurrency. Hence we conduct an experiment to determine the variability in run times of our parallel matching algorithms.

We measure the parallel sensitivity (ψ) of an algorithm as the ratio of the standard deviation (σ) of runtimes from N runs, to the mean of runtimes (μ): $\psi = \frac{\sigma}{\mu} \times 100$. We report the parallel sensitivity of three algorithms on Opteron for three input graphs in Fig. 2, based on ten repetitions of the same algorithm for the same number of threads.

The first observation we make is that the variations in run times are higher for the DFS-based algorithms (less than 25% for most problems) relative to the BFS-based algorithms (less than 5% for most problems). In contrast to DFS-based algorithms, the BFS-based algorithms generally have many vertices in a level where each thread performs a relatively small amount of work. Thus, the impact of load imbalance among the threads is smaller. A second observation we make is that as the number of threads increases, DFS-based algorithms become even more sensitive. Load balancing becomes difficult with a larger number of threads, especially when the tasks have varying work loads. Third, incorporating fairness into the PPF+ algorithm makes it faster, but also makes it more sensitive to non-determinism. This is in contrast with serial algorithms [5], where fairness had a stabilizing effect. Finally, we note that the sensitivities are still small relative to the speedups that we observe, so that parallelization is worth the effort. In light of these results, for all subsequent experiments in this paper we report results from the best performing run from a set of three independent runs. Given a small margin of variance, we make conclusions by ignoring sensitivity where

relevant, and do not make conclusions when the margin of difference is small.

C. Scalability of Maximum Matching Algorithms

We experimented with five different maximum matching algorithms on three different architectures. Initially we discuss results for the Nehalem and the Opteron processors with a relatively small number of threads in a system; we will consider the massively multithreaded Cray XMT towards the end of this discussion.

We show the scaling of two of the parallel algorithms, Parallel Pothen-Fan (PPF+) and Parallel Relaxed-HK (PRHK) algorithms for three problems on the AMD Opteron platform in Fig. 3. The serial runtimes are from Pothen-Fan with fairness (PF+) and HKDW algorithms; the latter is a variant of the Hopcroft-Karp algorithm due to Duff and Wiberg, when after finding shortest augmenting paths in the layered graph, a disjoint DFS is done from the remaining unmatched vertices before the next layered graph is constructed. These results are obtained from the implementations of Duff *et al.* [5] of the two serial algorithms also run on the same platform. We believe their implementations are among the best open-source, serial implementations currently available.

We make a number of observations from the data presented here.

The first observation is that we obtain significant speedups from the parallel implementations of different matching algorithms. This is demonstrated from the results on three problems shown in Fig. 3, as well as those that will be presented soon. We observe nearly linear speedups for the `amazon` and `cit-patents` problems on the Opteron for the PPF+ algorithm. The number of iterations needed by a DFS-based algorithm decreases for these problems when there is a moderate number of threads. We also note that for problems such as `cit-patent`, our algorithm on one thread runs faster than the Duff *et al*. code; we do not understand the reason for this at this time.

Our second observation links the performance of algorithms to the characteristics of input. The relative performance of DFS-based and BFS-based parallel algorithms is dependent on the structure of a given problem. For many problems, we see that DFS-based algorithms are faster and scale better, as can be seen in results from `GL7d19` and `cit-patents`. For these problems, the average number of edges traversed in an iteration to discover vertex-disjoint augmenting-paths is considerably smaller for DFS than BFS. The `amazon` problem features degree distribution characteristics of a scale-free graph, and here, BFS-based algorithms are faster due to the availability of a large number of short vertex-disjoint augmenting-paths that can be identified quickly. For scale-free graphs, when the number of threads increase, DFS-based algorithms are able to discover a large number of short vertex-disjoint augmenting-paths, and these algorithms tend to become as fast as the BFS-based algorithms.

Our third observation in on the general behavior of different algorithmic approaches. In general, BFS-based algorithms require fewer iterations than DFS-based algorithms to compute a maximum matching. However, BFS-based algorithms have a greater cost per iteration. This is due to two reasons: one is that they tend to search more edges in each iteration, and the second is caused by the finer-grained level-based synchronization in the algorithm compared to the path-based synchronization in DFS-based algorithms. Thus, while the latter run faster on a per-iteration basis, they have to run many more iterations to complete. The results in Fig. 3 show that quite often the product of the number of iterations and the average time per iteration, which is the total time taken by the algorithm, is in favor of the parallel DFS-based algorithms, especially when lookahead is employed. However, this is also influenced by the structure of the graph. We have also noticed that the number of iterations in a BFS-based algorithm is relatively independent of the number of threads. In contrast, for many problems the number of iterations decreases linearly with increasing number of threads for DFS-based algorithms. This is because for a large number of threads, DFS is capable of finding many short vertex-disjoint augmenting-paths much the same way that BFS does.

D. Comparison Across Different Platforms

We now provide details and observations on performance across platforms. In Fig. 4, we present scalability of the PRHK and PPF+ algorithms on the three platforms. On both Nehalem and Opteron, with a relatively small number of threads, we obtain good scaling for three graphs with different characteristics: `USA-Roadmap` is a sparse graph with good locality, `Copaper-DBLP` is a scale-free graph, and `ER22` is a sparse random graph with poor locality characteristics. On the first of these, the PRHK algorithm runs the fastest on the XMT, and scales well with increasing number of threads. Note that the PPF+ algorithm runs faster on Nehalem compared to Opteron, and is slowest on the XMT for the latter two problems.

Fig. 5 shows the performance of two DFS-based algorithms and three BFS-based algorithms on our sixteen test problems, on 16 processing units of the Opteron and the XMT. Note that on the Opteron, the DFS-based PPF+ is clearly the fastest algorithm for this test suite: it is faster than all other algorithms for 60% of the problems. It is followed by the simpler PDDFS and PDBFS algorithms, both of which outperform the Hopcroft-Karp variants.

BFS-based algorithms are more naturally suited to the XMT architecture, which uses a large number (up to hundred) of threads per processor to hide the large memory latencies. This is seen in the performance profiles of four different algorithms on the Cray XMT for the sixteen test problems in the subfigure on the right in Fig. 5. The best performing algorithm on the XMT is the parallel disjoint BFS (PDBFS) algorithm, which is the best algorithm for 65% of the problems. It is followed by PPF+, and then the HK variants. In level-synchronized BFS, there is a large number of vertices in each level that the large number of threads can process, and BFS does a better job than DFS in keeping the lengths of augmenting paths. On

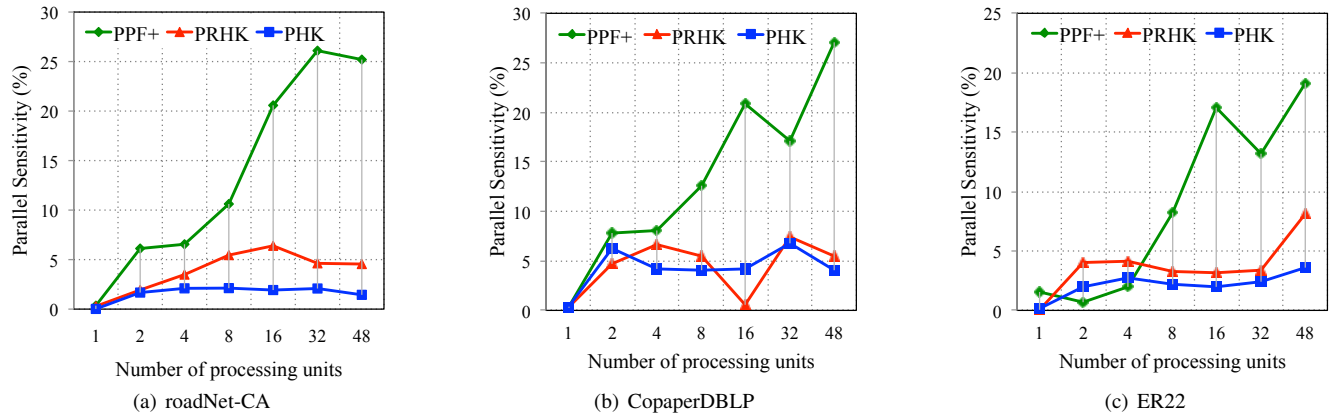


Fig. 2. **Sensitivity of maximum matching algorithms:** Sensitivity of different maximum matching algorithms with parallel Karp-Sipser initialization; runs are on Opteron.

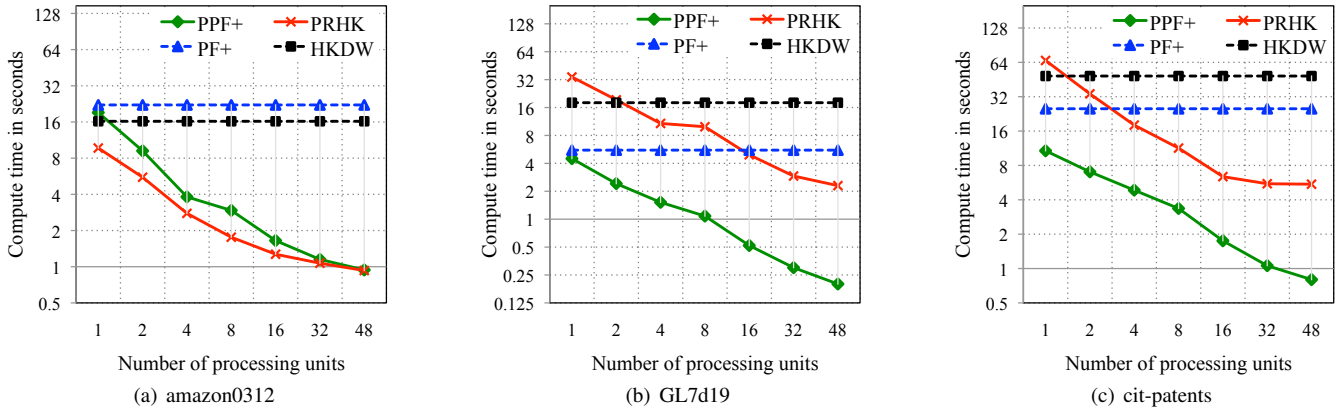


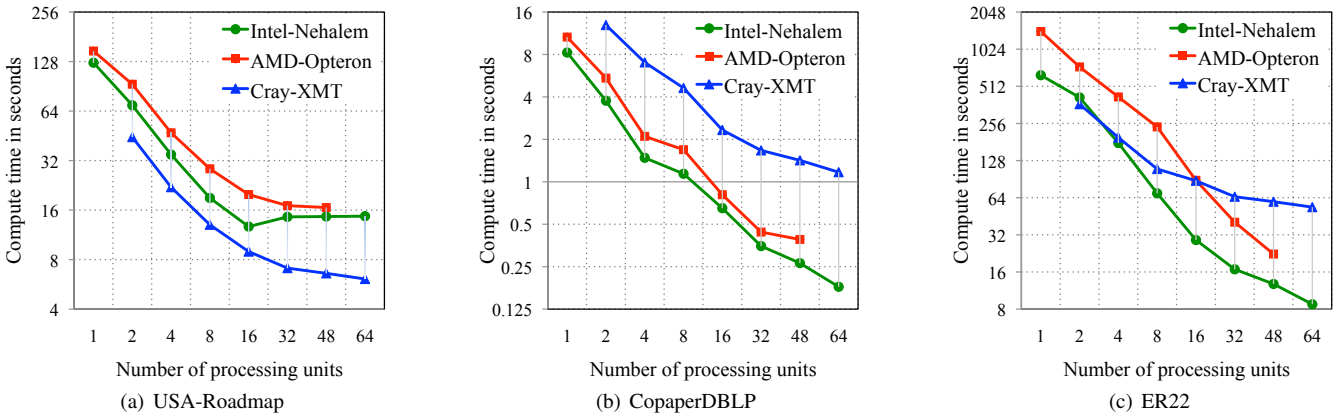
Fig. 3. **Scaling of maximum matching algorithms:** The scaling of exact matching algorithms with parallel Karp-Sipser algorithm as an initializer on Opteron for three problems. Serial runtimes of PF+ and HKDW are from the implementation of Duff *et al.* [5].

the XMT which has hardware-based synchronization, the costs of synchronizing the threads at the end of each level is also relatively small compared to the Nehalem and the Opteron. The DFS-based algorithms have a much bigger variation in the lengths of augmenting paths, but each iteration can be implemented faster than BFS-based algorithms, due to the relatively lower synchronization costs (at the granularity of paths rather than levels), and the short augmenting paths that lookahead helps DFS-based algorithms to find. However, the last few iterations of a DFS-based algorithm have few vertices to match, and the augmenting paths now are relatively long, and such iterations cannot make effective use of the massive number of threads on the XMT. Consequently, these iterations serialize on the XMT, and the parallel DFS-based algorithms perform poorly.

In comparing the performance of the XMT against the two other architectures, a few other features should be borne in mind: The XMT can compute matchings in much larger graphs, which are beyond the memory limitations of the Nehalem and the Opteron, and the larger problems will yield better performance on the XMT since they can take better advantage of the massive multithreading. The XMT has the slowest clock among these three processors, and the newer XMT-2 systems with a faster and larger memory system will potentially yield better relative performance against the other two architectures.

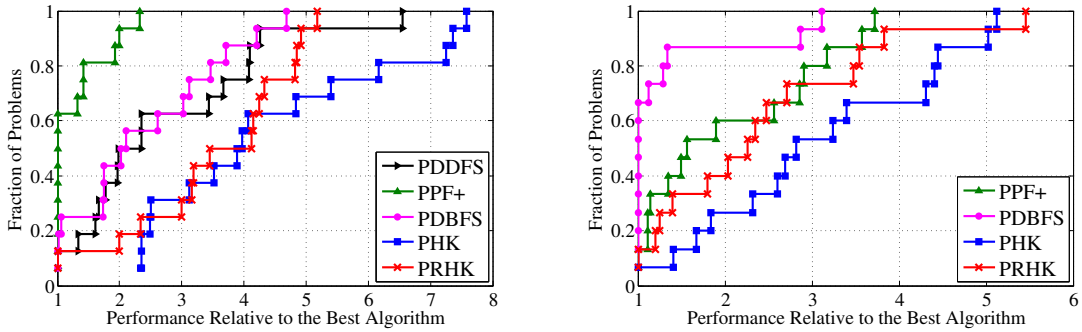
VI. Conclusions

We have evaluated the performance of five augmenting path-based algorithms for computing maximum cardinality matchings in bipartite graphs on three multithreaded architectures across a collection of test graphs with different structures. Our findings are reasonably consistent with the work of Langguth *et al.* [6] and Duff *et al.* [5] that compared related algorithms on serial architectures. We find that initialization algorithms are critical to the performance of matching algorithms, and that a DFS-based algorithm, PPF+, and a variant of a BFS-based algorithm, PDBFS, are among the best performers. We also find that different classes of algorithms perform best on different classes of architectures. DFS-based algorithms have a long 'tail' when there are a number of iterations that find few, long, augmenting paths; hence these algorithms perform poorly on the massively multithreaded Cray XMT, on which execution serializes at this stage. However, these



(a) USA-Roadmap (b) CopaperDBLP (c) ER22

Fig. 4. **Comparing two parallel algorithms on different architectures:** Scaling of the PRHK algorithm on the USA-Roadmap graph and the PPF+ algorithm on CopaperDBLP and ER22 graphs, all with parallel Karp-Sipser algorithm as the initializer.



(a) AMD Opteron (b) Cray XMT

Fig. 5. **Profile of relative performance of different algorithms:** The relative performance of different algorithms for all our test problems with respect to the best algorithm for a given problem.

algorithms run faster on the multicore machines, the Nehalem and the Opteron, since the tail takes a smaller fraction of the total execution times due to the faster processor speeds. The DFS-based algorithms tend to be more sensitive to the non-determinism from one execution to another, and hence show greater variation in the run times. Unlike the results reported by Duff *et al.* for the serial case, the use of fairness in the PPF+ algorithm makes it more, not less, sensitive to non-determinism in the multithreaded environment. However, we find that this sensitivity is still small relative to the speed-ups that we observe. For many graphs we obtain nearly linear speed-ups. Even superlinear speedups are possible, since the distribution of the number and the lengths of augmenting paths can vary from one algorithm to another, and can vary also with the order in which vertices are considered for matching. Finally, the structure of the graphs influences the performance of multithreaded algorithms. Scale-free graphs have a large number of short augmenting paths, and hence BFS-based algorithms tend to be faster on these problems. Generally, BFS-based algorithms require fewer iterations than DFS-based algorithms, but creating the layered graph in a BFS-based algorithm searches more edges per iteration and has higher synchronization costs than a DFS-based algorithm.

It should be noted that the serial algorithms we have compared against are the best performers from a large set of algorithms developed over the last few decades, with implementations targeted for high performance. In contrast, our first-in-class multithreaded parallel implementations have room for improvement from several optimizations. Among them is allocating memory to portions of the graph that will be processed by a thread in a memory module local to the socket it belongs to. We have also not considered the multithreaded implementation of a fourth class of algorithms, the push-relabel algorithm.

ACKNOWLEDGEMENTS

This work was funded by the Center for Adaptive Super Computing Software - MultiThreaded Architectures (CASS-MT) at the U.S. Department of Energy's (DOE) Pacific Northwest National Laboratory. PNNL is operated by Battelle Memorial Institute under Contract DE-ACO6-76RL01830. Funding was also provided by the DOE through the CSCAPES Institute (grants DE-FC02-08ER25864 and DE-FC02-06ER2775), and NSF grant CCF-0830645. Sandia National Laboratories is a multi-program laboratory managed and operated by Sandia Corporation, a wholly owned subsidiary of Lockheed Martin Corporation, for the DOE's National Nuclear Security Administration under contract DE-AC04-

94AL85000. We acknowledge fruitful discussions with Florin Dobrian, Bora Uçar and John Feo.

REFERENCES

[1] M. Brady, K. Jung, H. Nguyen, R. Raghavan, and R. Subramonian, "The assignment problem on parallel architectures," *Network Flows and Matching. DIMACS*, pp. 469–517, 1993.

[2] A. Pothen and C.-J. Fan, "Computing the block triangular form of a sparse matrix," *ACM Trans. Math. Softw.*, vol. 16, pp. 303–324, December 1990.

[3] R. Burkard, M. Dell'Amico, and S. Martello, *Assignment Problems*. Society for Industrial and Applied Mathematics, 2009.

[4] J. Hopcroft and R. Karp, "A $n^{5/2}$ algorithm for maximum matchings in bipartite graphs," *SIAM J. Comput.*, vol. 2, pp. 225–231, 1973.

[5] I. S. Duff, K. Kaya, and B. Uçar, "Design, analysis and implementation of maximum transversal algorithms," *ACM Transactions on Mathematical Software*, vol. 28, no. 2, pp. 13:1– 13:31, 2011.

[6] J. Langguth, F. Manne, and P. Sanders, "Heuristic initialization for bipartite matching problems," *Journal of Experimental Algorithmics (JEA)*, vol. 15, pp. 1–3, 2010.

[7] M. Karpinski and W. Rytter, *Fast Parallel Algorithms for Graph Matching Problems*. New York, NY, USA: Oxford University Press, Inc., 1998.

[8] J. Langguth, M. M. A. Patwary, and F. Manne, "Parallel algorithms for bipartite matching problems on distributed memory computers," *Parallel Computing*, vol. 37, no. 12, pp. 820–845, 2011.

[9] A. V. Goldberg and R. E. Tarjan, "A new approach to the maximum flow problem," in *Proceedings of the Eighteenth Annual ACM Symposium on Theory of Computing*, ser. STOC '86. New York, NY, USA: ACM, 1986, pp. 136–146.

[10] M. A. Patwary, R. Bisseling, and F. Manne, "Parallel greedy graph matching using an edge partitioning approach," in *Proceedings of the Fourth International Workshop on High-level Parallel Programming and Applications*. ACM, 2010, pp. 45–54.

[11] J. C. Setubal, "Sequential and parallel experimental results with bipartite matching algorithms," University of Campinas, Tech. Rep. IC-96-09, 1996.

[12] D. Bertsekas, "Auction Algorithms for Network Flow Problems: A Tutorial Introduction," *Computational Optimization and Applications*, vol. 1, no. 1, pp. 7–66, 1992.

[13] D. P. Bertsekas and D. A. Castañon, "Parallel synchronous and asynchronous implementations of the auction algorithm," *Parallel Comput.*, vol. 17, pp. 707–732, September 1991.

[14] L. Buš and P. Tvrdík, "Towards auction algorithms for large dense assignment problems," *Comput. Optim. Appl.*, vol. 43, pp. 411–436, July 2009.

[15] M. Sathe and O. Schenk, "Computing auction based weighted matchings in massive graphs," 2011, talk presented at ICIAM 2011.

[16] F. Manne and R. H. Bisseling, "A parallel approximation algorithm for the weighted maximum matching problem," in *Proceedings of the 7th international conference on Parallel processing and applied mathematics*, ser. PPAM'07. Berlin, Heidelberg: Springer-Verlag, 2008, pp. 708–717.

[17] M. Halappanavar, "Algorithms for vertex-weighted matching in graphs," Ph.D. dissertation, Old Dominion University, Norfolk, VA, USA, 2009.

[18] Ü. V. Çatalyürek, F. Dobrian, A. H. Gebremedhin, M. Halappanavar, and A. Pothen, "Distributed-memory parallel algorithms for matching and coloring," in *IPDPS Workshops*, 2011, pp. 1971–1980.

[19] M. Halappanavar, J. Feo, O. Villa, A. Tumeo, and A. Pothen, "Approximate weighted matching on emerging manycore and multithreaded architectures," *International Journal of High Performance Computing Applications (IJHPCA)*, submitted 2011.

[20] "AMD Opteron 6100 Series Processor," available at http://www.amd.com/us/products/embedded/processors/opteron/Pages/opteron-6100-series.aspx.

[21] D. Molka, D. Hackenberg, R. Schone, and M. S. Muller, "Memory performance and cache coherency effects on an Intel Nehalem multiprocessor system," in *PACT '09: Proceedings of the 2009 18th International Conference on Parallel Architectures and Compilation Techniques*. Washington, DC, USA: IEEE Computer Society, 2009, pp. 261–270.

[22] J. Feo, D. Harper, S. Kahan, and P. Konecny, "Eldorado," in *CF '05: Proceedings of the 2nd Conference on Computing Frontiers*. New York, NY, USA: ACM, 2005, pp. 28–34.

[23] D. Chakrabarti and C. Faloutsos, "Graph mining: Laws, generators, and algorithms," *ACM Comput. Surv.*, vol. 38, no. 1, p. 2, 2006.

Algorithm 1 The Parallel Pothen-Fan Algorithm with fairness.

Input: A graph G. **Output:** A matching M.

procedure PPF+($G(X \cup Y, E)$)
  $M \leftarrow InitMatch(G)$ ▷ E.g., Karp Sipser
  Initialize $lookahead[i]$ to the first neighbor of i ▷ For each vertex: $i \leftarrow 1..n$
  repeat ▷ Each iteration
    $path\_found \leftarrow 0$
    $visited[v] \leftarrow 0$ ▷ For $v \in Y$
    for all unmatched vertices $u \in X$ in `parallel` **do** ▷ Look for augmenting paths in parallel
      $P \leftarrow$ DFS-LA-TS(u) ▷ Find a vertex-disjoint augmenting path from u
      if P found **then**
        $path\_found \leftarrow 1$
        $M \leftarrow M \oplus P$ ▷ Augment the matching with the path found
  until $path\_found = 0$ ▷ Repeat until no augmenting paths exist

Algorithm 2 Find a vertex-disjoint augmenting path using DFS with lookahead in a threadsafe way. **Input:** A graph G, a source vertex u, a matching M, vectors $visited$ and $lookAhead$. **Output:** An augmenting path P if found.

procedure DFS-LA-TS(u)
  for all $v \in adj[u]$ starting at $lookAhead[u]$ **do** ▷ Lookahead step
    $lookAhead[u] \leftarrow$ next neighbor of u ▷ Set to $\emptyset$ if v is the last neighbor
    if v is not matched **then**
      if `Atomic-Fetch-Add`($visited[v], 1) = 0$ **then** ▷ First thread to reach v in the lookahead step
        return v ▷ Treat path ending at v as an augmenting path.
  for all $v \in adj[u]$ **do** ▷ DFS step. For fairness, alternate search directions in odd and even iterations
    if `Atomic-Fetch-Add`($visited[v], 1) = 0$ **then** ▷ First thread to reach v in vertex-disjoint DFS
      $index \leftarrow$ DFS-LA-TS($M(v)$) ▷ Recursive call to continue DFS from mate of v
      if $index$!= $invalid$ **then return** $index$
  return $invalid$

Algorithm 3 Find a vertex-disjoint augmenting path using DFS in a threadsafe way. **Input:** A graph G, a source vertex u, a matching M and a vector $visited$. **Output:** An augmenting path P if found.

procedure DFS-TS(u)
  for all $v \in adj[u]$ **do**
    if `Atomic-Fetch-Add`($visited[v], 1) = 0$ **then** ▷ First thread to reach v
      if v is unmatched **then**
        return v
      else
        $index \leftarrow$ DFS-TS($M(v)$) ▷ Recursive call to continue DFS from mate of v
        if $index$!= $invalid$ **then return** $index$
  return $invalid$

Algorithm 4 The Parallel Hopcroft-Karp Algorithm.

Input: A graph G. **Output:** A maximum matching M.

procedure PHK($G(X \cup Y, E)$)
  $M \leftarrow InitMatch(G)$ ▷ E.g., Karp Sipser
  repeat ▷ Each iteration
    for all $u \in X$ **do**
      $next[u] \leftarrow$ first neighbor of u
    for all $w \in X \cup Y$ **do**
      $visited[w] \leftarrow 0$ ▷ $visited$ is set to *false*.
    $\mathbb{P} \leftarrow \emptyset$
    $(G_L, L_k) \leftarrow$ LAYERED-GRAPH-TS(G, M) ▷ Construct layered graph from all unmatched vertices
    for all $w \in X \cup Y$ **do**
      $visited[w] \leftarrow 0$ ▷ $visited$ is reset to *false* for DFS-TS.
    for all $v \in L_k \setminus V(M)$ in `parallel` **do** ▷ Unmatched vertices in last level
      $P_v \leftarrow$ DFS-TS(v) ▷ Find augmenting paths using DFS in G_L
      $\mathbb{P} \leftarrow \mathbb{P} \cup P_v$
    $M \leftarrow \mathbb{P} \oplus M$
  until $\mathbb{P} = \emptyset$ ▷ No augmenting paths exist

Algorithm 5 Construction of the layered graph in parallel. **Input:** A graph $G(X \cup Y, E)$ and a matching M. **Output:** A layered graph G_L and a set $L_k \subset Y$.

procedure LAYERED-GRAPH-TS($G(X \cup Y, E), M, visited$)
  $\mathbb{P} \leftarrow \emptyset$
  $L_0 \leftarrow$ unmatched vertices in X
  $k \leftarrow 0$
  while $true$ **do** ▷ Level-synchronous BFS to find shortest augmenting paths
    $L_{k+1} \leftarrow \emptyset$ ▷ Will consist of vertices from Y
    $L_{k+2} \leftarrow \emptyset$ ▷ Will consist of vertices from X
    for all $u \in L_k$ in `parallel` **do**
      for all $v \in adj[u]$ **do**
        if `Atomic-Fetch-Add`$(visited[v], 1) = 0$ **then** ▷ First thread to visit vertex v
          Add vertex v to layer $k+1$ locally in a thread ▷ Thread-private vector of vertices
          Add edge $\{u, v\}$ to the local set of edges ▷ Thread-private vector of edges
          if $v \in V(M)$ **then** ▷ If vertex v is matched, add the next layer
            Add vertex $\{M[v]\}$ to layer $k+2$ locally in a thread
            Add edge $\{v, M[v]\}$ to local set of edges
    Concatenate local layers $k+1$ and $k+2$ *from all threads* to L_{k+1} and L_{k+2}
    Concatenate local edges *from all threads* to $E(G_L)$
    if $(L_{k+1} = \emptyset)$ **OR** $(L_{k+1} \setminus V(M) \neq \emptyset)$ **then** ▷ Last level is either empty or we have found an unmatched vertex
      $G_L \leftarrow \{(\bigcup_{0 \leq i \leq k+1} L_i, E(G_L)\}$
      return (G_L, L_{k+1}) ▷ Augmenting paths do not exist if L_{k+1} is empty
    else
      $k = k + 2$ ▷ Proceed to construct the next two levels

Algorithm 6 The Parallel Karp-Sipser Algorithm.

Input: A graph G. **Output:** A *maximal* matching M.

procedure PARALLEL-KARP-SIPSER($G(X \cup Y, E)$)
  $M \leftarrow \emptyset$
  $Q_1 \leftarrow \emptyset$
  $Q_* \leftarrow \emptyset$
  for all $u \in X \cup Y$ in `parallel` **do**
    $visited[u] \leftarrow 0$ ▷ $visited[u]$ is set to *false*.
  for all $u \in X$ in `parallel` **do**
    if $deg[u] = 1$ **then**
      $Q_1 \leftarrow Q_1 \cup \{u\}$ ▷ Add vertices of degree=1 in Q_1 and remaining to in Q_*
    else
      $Q_* \leftarrow Q_* \cup \{u\}$
  for all $u \in Q_1$ in parallel **do**
    MATCHANDUPDATE($G, M, u, visited$) ▷ Match degree=1 vertices, update degrees, and look for new degree=1 vertices
  for all $u \in Q_*$ in parallel **do**
    MATCHANDUPDATE($G, M, u, visited$) ▷ Match a higher degree vertex, update degrees, and look for new degree=1 vertices
  return M

Algorithm 7 Match a vertex if possible and process its neighbors. **Input:** A graph G, a matching M, a source vertex u, and a vector $visited$.

procedure MATCHANDUPDATE($G, M, u, visited$)
  if `Atomic-Fetch-Add`$(visited[u], 1) = 0$ **then** ▷ First thread to visit unmatched u
    for all $v \in adj[u]$ **do**
      if `Atomic-Fetch-Add`$(visited[v], 1) = 0$ **then** ▷ First thread to visit unmatched v
        $M \leftarrow M \cup \{u, v\}$ ▷ Found a mate for u
        for all $w \in adj[v]$ **do**
          if `Atomic-Fetch-Add`$(deg[w], -1) = 2$ **then** ▷ Update the degree of neighbors of v
            MATCHANDUPDATE($G, M, w, visited$) ▷ Recursive call to match the new vertex w of degree=1
          break ▷ Stop search when u gets matched