Approximate Weighted Matching On Emerging Manycore and Multithreaded Architectures

Mahantesh Halappanavar[1], John Feo[1], Oreste Villa[1], Antonino Tumeo[1], and Alex Pothen[2],

Abstract

Graph matching is a prototypical combinatorial problem with many applications in high performance scientific computing. Optimal algorithms for computing matchings are challenging to parallelize. Approximation algorithms are amenable to parallelization and are therefore important to compute matchings for large scale problems. Approximation algorithms also generate nearly optimal solutions that are sufficient for many applications. In this paper we present multithreaded algorithms for computing half-approximate weighted matching on state-of-the-art multicore (Intel Nehalem and AMD Magny-Cours), manycore (Nvidia Tesla and Nvidia Fermi), and massively multithreaded (Cray XMT) platforms. We provide two implementations, the first uses shared work queues and is suited for all platforms and the second implementation, based on dataflow principles, exploits special features available on the Cray XMT.

Using a carefully chosen dataset that exhibits characteristics from a wide range of applications, we show scalable performance across different platforms. In particular, for one instance of the input, an R-MAT graph (RMAT-G), we show speedups of about 32 on 48 cores of an AMD Magny-Cours, 7 on 8 cores of Intel Nehalem, 3 on Nvidia Tesla and 10 on Nvidia Fermi relative to one core of Intel Nehalem, and 60 on 128 processors of Cray XMT. We demonstrate strong as well as weak scaling for graphs with up to a billion edges using up to $12,800$ threads. We avoid excessive fine-tuning for each platform and retain the basic structure of algorithm uniformly across platforms. An exception is the dataflow algorithm designed specifically for the Cray XMT. To the best of our knowledge, this is the first such large-scale study of the half-approximate weighted matching problem on multithreaded platforms.

Driven by the critical enabling role of combinatorial algorithms such as matching in scientific computing and the emergence of informatics applications, there is a growing demand to support irregular computations on current and future computing platforms. In this context, we evaluate the capability of emerging multithreaded platforms to tolerate latency induced by irregular memory access patterns, and to support fine-grained parallelism via light-weight synchronization mechanisms. By contrasting the architectural features of these platforms against the Cray XMT, which is specifically designed to support irregular memory-intensive applications, we delineate the impact of these choices on performance.

I. Introduction

A matching M in a graph $G = (V, E)$ is a subset of edges such that no two edges in M are incident on the same vertex. In other words, a matching is a pairing of vertices between the two endpoints of matched edges. Based on its objective function, there are several variants of the matching problem. For example: a *maximum* (cardinality) matching maximizes the number of matched edges in M; a *maximum weighted* matching maximizes the sum or product of the weights of matched edges; and a *bottleneck* matching has an objective function to maximize the minimum weight of a matched edge in M. The algorithms can be further classified as *optimal* or *approximate*. In this paper, we are interested in the weighted matching problem with an objective function to maximize the sum of weights of matched edges. In particular, we focus on the $^1/_2$-approximate weighted matching problem that guarantees a solution that is at least half the weight of a solution computed by an optimal algorithm. Formally, given a graph $G = (V, E)$ with a weight function $w : E \rightarrow \mathbf{R}^+$, where V is a set of vertices, E is a set of edges, and each edge has a positive real number as its weight, and a matching $M \subseteq E$, the weight of the matching is given by $w(M) = \sum_{e \in M} w(e)$, where $e \in E$ is an edge. We present a serial $^1/_2$-approx algorithm [1], referred to as the *Locally-Dominant Algorithm*, in Section III-A. Parallel implementations of this algorithm are presented in Section IV. A novel implementation based on the dataflow principles is presented in Section IV-B. The dataflow algorithm addresses some of the limitations of the algorithm based on shared work queues.

Matching is a fundamental combinatorial problem with many applications in scientific computing. For instance: weighted as well as unweighted matchings are used in the solution of sparse linear systems to place large matrix elements on or close to the diagonal [2], [3], [4]; weighted matchings are used in the computation of sparse basis for the null space or column space of under-determined matrices [5]; maximum matching is used in the computation of block-triangular form a matrix (BTF) [6]; approximate weighted matchings are used in multi-level graph algorithms for partitioning and clustering during the coarsening phase [7]; weighted matchings are used in image processing using shape contexts [8]. In addition to scientific computing, matching also plays a key enabling role in bioinformatics, network switch design, web technologies, etc. These applications drive the need for efficient parallel implementations of matching algorithms. The need is justified not only by the emergence of large-scale problems where time-to-solution is important, but also by the stagnation in the performance of serial processing [9].

While desirable, there are considerable challenges in implementing graph algorithms on traditional distributed-memory parallel processors and has been well documented in scientific literature [10], [11]. Some of the key challenges are: (i) efficient distribution of work among processors is determined by the structural and numerical properties of the input. Load balanced partitioning of work is particularly challenging in emerging applications in informatics [12]; (ii) a lack of coarse-grained

[1]Pacific Northwest National Laboratory. [2] Purdue University. Email:{Mahantesh.Halappanavar@pnl.gov, John.Feo@pnl.gov, Oreste.Villa@pnl.gov, Antonino.Tumeo@pnl.gov, apothen@purdue.edu}

parallelism in many graph algorithms leads to fine-grained synchronization resulting in poor utilization of resources; and (iii) a small amount of computation per communicated word makes amortization of the communication cost difficult. Fine-grained synchronization can also lead to low computation per communicated word. As a consequence of these performance limiting challenges, shared-memory platforms provide an attractive alternative for implementing and executing graph algorithms [13], [14]. However, efficient implementation of graph algorithms on shared-memory platforms is also challenging. Some of the challenges are similar to those encountered on distributed-memory platforms: irregular memory-access patterns that result in poor utilization of system resources; low amounts of work per accessed word of memory that make amortization of memory access cost difficult; and the need for fine-grained synchronization between threads that lead to poor utilization of system resources.

In order to address these challenges, we focus on three key hardware-software co-design features: (i) hardware multithreading to tolerate latencies arising not only from memory operations but also from synchronization between threads, (ii) dynamic, rather than static, load-balancing through scheduling and shared work queues, and (iii) extended memory semantics using tag bits to enable fine-grained synchronization. While we focus on these three features, we note that there are several other components that are needed to deliver scalable performance such as: runtime support, programming models, and efficient atomic memory operations. In order to evaluate these features effectively we provide results on five shared-memory platforms are are chosen to cover a broad spectrum of features – traditional multicore platforms with deep cache hierarchies, emerging manycore platforms (GPUs), and a non-traditional massively multithreaded cache-less architecture. These platforms are discussed in Section III-B, and the experimental results are provided in Section V.

Coinciding with the manycore revolution in computer architecture is a revolution in informatics applications that is being enabled by the emergence of large-scale real-world data from scientific experiments as well as Internet phenomena such as social networks and connectivity of web pages. While the graphs arising in scientific computing tend to be fairly regular in structure, those arising in emerging applications such as social network analysis tend to be characterized by power law degree distribution and small world phenomenon of short average distance between any two vertices in a graph as well as dense clustering within groups of vertices. In order to represent the characteristics of these graphs we experiment with a carefully chosen set of synthetic graphs generated using the R-MAT algorithm and a few instances from real-world data. The details are provided in Section III-C.

The key contributions of this paper are:

1) A detailed discussion of multithreaded implementations of the $^1/_2$-approx weighted matching algorithm including a novel dataflow-based implementation on the Cray XMT.
2) Presentation of scalable experimental results on emerging multicore (AMD Magny-Cours, Intel Nehalem), manycore (Nvidia Tesla and Nvidia Fermi), and massively multithreaded (Cray XMT) platforms.
3) Presentation of insight from considering algorithm design, input characteristics, and hardware features in tandem.

To the best of our knowledge, for the $^1/_2$-approx weighted matching problem: this is the first large-scale study on diverse shared-memory parallel platforms, first implementation on GPUs, and the first work to consider the interplay of algorithm design and hardware features using large-scale graphs of up to a billion edges carefully generated and selected from real-world data. In the past, only a limited set of results have only been presented on distributed-memory platforms [15] and shared-memory platforms [1]. The input in these experiments has been limited to a few classes of graphs.

II. Related Work

Matching is an important combinatorial problem and has been extensively studied by researchers. Detailed discussions on matching can be found in [16], [17], [18]. Approximation algorithms for matching have also been studied extensively. Avis proposed a greedy $^1/_2$-approx algorithm for weighted matching based on *sorting*. Edges are sorted in a non-increasing order of their weights. A maximal (an approximation guarantee of half for cardinality) matching is then computed by repeatedly adding the current heaviest edge to the matching and removing all its neighbors from the graph [19]. A strict global order for processing edges renders this approach unsuitable for parallel implementation. Preis proposed a $^1/_2$-approx algorithm for weighted matching based on *searching* instead of sorting [20]. Preis' algorithm starts from an arbitrary edge and processes its neighbors. If one of its neighbors is heavier, then it progresses along that edge. The search stops when a *locally-dominant* edge, an edge heavier than all its neighbors, is reached. Preis' algorithm computes a $^1/_2$-approx matching in linear time. Preis' algorithm, as proposed, relies on graph searches and complex book-keeping and is unsuitable for parallel implementation. However, Preis' work was seminal and laid the foundation for subsequent research by others. In particular, the work of Drake and Hougardy that followed the work of Preis is noteworthy for its simplicity. Their $^1/_2$-approx algorithm for weighted matching is based on the simple idea of growing paths. Two temporary sets of matched edges are maintained. The algorithm starts the search from an arbitrary vertex and selects the heaviest neighbor. This neighbor is added to the first set of matching and the vertex (and all edges incident on it) is deleted from the graph. The search continues from other endpoint of the matched edge, but this time the heaviest edge is added to the second set of matched edges. The search continues until all the vertices in the graph have been visited. The heavier of the two sets is returned as the final solution. Although simple, this algorithm is also unsuitable for parallel implementation [21].

Hoepman developed a distributed variant of Preis' algorithm assuming one vertex per processor [22]. Manne and Bisseling adapted this algorithm for a practical distributed-memory implementation with multiple vertices (subgraph) per processor [1]. Building on Manne and Bisseling's work, efficient distributed-memory parallel algorithm and scalable implementation was described by Halappanavar [15] and Catalyurek *et al.* [23]. In this paper, we provide multithreaded implementations of the same algorithm that we refer to as the locally-dominant algorithm in the paper.

A class of optimal algorithms for computing matchings are based on the technique of *augmentation*, in which a matching computed at an intermediate step is augmented by using special paths known as augmenting paths [16]. Matching algorithms based on the technique of augmentation perform searches in a graph either in a breadth-first or a depth-first manner. In this context, we cite related work on breadth-first search, st-connectivity, and all-pairs shortest-path computations. Several shared-memory, as well as distributed-memory, implementations exist for these graph kernels. Some of the important references include the seminal work of Bader and Madduri on Cray MTA-2 (a predecessor of XMT) [24], Nieplocha *et al.* on XMT [14], Cong and Bader on shared-memory platforms [13], Agarwal *et al.* on emerging multicore platforms [25], Yoo *et al.* on BlueGene/L platform [26], and Hendrickson and Berry [11] on cross-architecture comparisons and computational challenges.

Several papers on implementation of graph algorithms on general purpose graphics processing units (GPUs) exist. Foremost among them is the work of Harish and Narayanan [27]. Subsequent work is presented by Katz and Kider [28], Luo *et al.* [29], and most recently by Hong *et al.* [30]. In the work of Harish and Narayanan we observe that the cost of breadth-first search is $O(|V| \cdot L + |E|)$, where L represents the number of levels (the longest distance from the source to any vertex), and $|V|$ and $|E|$ are the number of vertices and edges respectively. While an efficient implementation of breadth-first search has a cost of $\Theta(|V| + |E|)$, the implementation of Harish and Narayanan is negatively impacted by architectural limitations of older generation of GPUs. The implementation of Harish and Narayanan, and a similar approach of others, is therefore suitable only for instances with small values of L. In contrast, our GPU implementation of approximate matching uses an efficient approach that is similar to implementation on other multithreaded platforms.

III. Preliminaries

We provide relevant background information in this section. The information is organized into three subsections on: serial algorithm for the $^1/_2$-approx weighted matching problem, hardware platforms, and the dataset used for experiments.

A. Locally-Dominant Algorithm for $^1/_2$-approx Weighted Matching

We now describe a generic queue-based implementation of the $^1/_2$-approx algorithm for weighted matching. The algorithm was proposed by Manne and Bisseling [1] as a serial variant of Hoepman's algorithm [22]. We provide the details in Algorithm 1, which takes as input a graph $G = (V, E)$ and returns as output a matching M. In order to simplify the description of parallel algorithms, we divide the execution into two phases: Phase-1 and Phase-2.

Algorithm 1 Serial Queue-based Implementation. *Input*: graph $G = (V, E)$. *Output*: A matching M represented in vector mate. *Data structures*: a queue, Q, that consists of unprocessed matched vertices; a vector candidate of size $|V|$ that contains the id of the current-heaviest neighbor of each vertex.

```
1:  procedure SERIAL-QUEUE(G(V, E), mate)
2:      for each u ∈ V do                                          ▷ Initialization
3:          mate[u] ← ∅
4:          candidate[u] ← ∅
5:      Q ← ∅
6:      for each u ∈ V do                                          ▷ Phase-1
7:          PROCESSVERTEX(u, Q)
8:      while Q ≠ ∅ do                                             ▷ Phase-2
9:          u ← FRONT(Q)
10:         Q ← Q \ {u}
11:         for each v ∈ adj(u) do
12:             if candidate[v] = u then                           ▷ Process v only if u is it's candidate mate
13:                 PROCESSVERTEX(v, Q)
```

In Phase-1, each vertex $v \in V$ is processed by making a call to Procedure PROCESSVERTEX, which is given in Algorithm 2. In Procedure PROCESSVERTEX, for a given vertex s, all its neighbors are scanned to find the current-heaviest neighbor that has not already been matched. When duplicate weights exist, it is important to break ties consistently to prevent deadlocks. We use vertex indices, which are guaranteed to be unique, to break ties consistently. The identity of the heaviest neighbor for each vertex is stored in vector candidate (Line 8 in Algorithm 2). After setting the candidate mate for vertex s, say to vertex t, we check if the candidate mate for t is also set to s: candidate[candidate[s]] = s. If true, we have found a *locally-dominant* edge in $e_{s,t}$. We add this edge to M (Lines 10 and 11), and the two vertices s and t to the queue (Line 12). Some of the vertices might end up not having any candidates available to match.

Algorithm 2 ProcessVertex

```
1: procedure PROCESSVERTEX(s, Q)
2:     max_wt ← −∞
3:     max_wt_id ← ∅
4:     for each t ∈ adj(s) do
5:         if (mate[t] = ∅) AND (max_wt < w(e_{s,t})) then          ▷ Use vertex id to break ties
6:             max_wt ← w(e_{s,t})
7:             max_wt_id ← t
8:     candidate[s] ← max_wt_id
9:     if candidate[candidate[s]] = s then                          ▷ Found a locally-dominant edge
10:        mate[s] ← candidate[s]
11:        mate[candidate[s]] ← s
12:        Q ← Q ∪ {s, candidate[s]}                                ▷ Only need to process vertices in the queue.
```

The execution of Phase-2 begins when every vertex has been processed via a call to Procedure PROCESSVERTEX, which happens at the completion of Phase-1. In Phase-2, we iterate until the queue becomes empty (Line 8 in Algorithm 1). Note that at least one edge (the heaviest edge in G) will get matched in Phase-1, and therefore, Q is nonempty if M is nonempty. During each iteration of the **while** loop on Line 10, we process vertices matched in previous iterations while adding new vertices to the queue that become eligible as edges get matched. Note that we only need to process vertices for which the candidate was set to one of the matched vertices (Line 12 in Algorithm 1). This is achieved by adding the newly matched vertices to the queue and checking if any of their unmatched neighbors point to them. If so, those neighbors will have to find new candidates for matching. The algorithm will terminate when the queue becomes empty.The matching is stored in vector mate. An index s of this vector represents vertex s, and the vertex $t = \mathsf{mate}[s]$ represents the other endpoint of the matched edge $e_{s,t}$.

An indication of the amount of work completed up to a certain stage in the execution of Algorithm 2, and the amount of work at a given step of execution, is given by the size of queue (Q) with respect to time. At the end of Phase-1, we expect several vertices (at least two) to be added to the queue. In practice, we observe a large percentage of edges being matched in Phase-1. Let us consider the end of Phase-1 as the first iteration. The iterations of the while loop on Line 8 in Algorithm 1 will be considered as the subsequent iterations with the following modification. During each iteration of the while loop, all the elements of Q are processed. Let newly matched vertices be added to a temporary queue. At the end of the iteration, elements of the temporary queue are moved to Q. The size of Q provides an indication of the number of edges that get matched at each iteration (or a step of the algorithm). The queue sizes for three different inputs are illustrated in Figure 1(c). We observe that the queue size decreases roughly by half after each iteration, and has the largest size at for iteration 1 (at the end of Phase-1).

The cost of Algorithm 1 is given by $O(|V| + |E| \cdot \Delta)$, where Δ is the maximum degree in G. The worst case happens when a vertex points to all of its neighbors unsuccessfully, and in order to determine the current heaviest neighbor needs to check its entire list of neighbors. However, the runtime can be improved to $\Theta(|V| + |E|)$ if the adjacency list for each vertex is maintained in a non-increasing order of edge weights. Under such an assumption, the current heaviest neighbor of a vertex can be computed in constant time. In practice, we observe that the benefits from ordered adjacency lists is about 2 relative to unordered lists (Figure 5). However, the cost of sorting adjacency lists is considerably more than matching. Therefore, sorting is not a viable optimization strategy. In this paper, we use sorted lists only to demonstrate its effectiveness and to simplify the explanation of the dataflow algorithm (Algorithm 4) presented in Section IV-B.

Performance of approximation algorithms can be measured based on quality and runtime of the algorithms. Quality can be measured in terms of the cardinality and weight of approximate matchings with respect to optimal matchings. Approximation algorithms have been demonstrated to compute high quality matchings in a fraction of the time taken by optimal algorithms [15]. When compared to other approximation algorithms, the Locally-Dominant algorithm performs better in both quality and runtime. Experimental results comparing different approximation algorithms are provided in [15]. The details of parallel implementations of the Locally-Dominant Algorithm are provided in Section IV, and experimental results are provided in Section V.

B. Hardware Platforms

The five platforms selected in this study – Opteron, Nehalem, Tesla, Fermi, and XMT – represent a broad spectrum of capabilities: clock speeds ranging from 0.5 GHz (XMT) to about 3.0 GHz (Nehalem); hardware multithreading ranging from none (Opteron) to 128 threads per processor (XMT); cache hierarchies ranging from none/flat (XMT) to three levels (Nehalem); generation of memory interconnect ranging from DDR1 (XMT) to GDDR5 (Fermi); advanced architectural features such as branch prediction and speculative execution on modern architectures to simple features of the XMT; and control structures ranging from fully autonomous (MIMD) processors (Opteron) to a 32-way SIMT (Fermi). Different platforms employ different techniques for performance. For example, while XMT uses massive multithreading to tolerate latency from memory operations, Nehalem employs deep cache hierarchies, advanced branch prediction, and two-way multithreading to tolerate latencies. While some architectures are extremes, others are more balanced. For example, compare Nehalem with two-way HyperThreading and

three levels of caches to the XMT that has 128-way threading and a cache-less memory structure. These contrasting hardware features therefore offer a rich environment to compare performance impact of different features with respect to each other. Key architectural features of these five platforms are summarized in Table I. Considering the scope and length of this paper we provide brief descriptions as applicable to irregular applications in general and graph algorithms in particular. References for further details are also provided in the table.

Platform:	Opteron 6176 SE (Magny-Cours)	Xeon E5540 (Nehalem)	Tesla C1060 (Tesla)	Tesla C2050 (Fermi)	ThreadStorm-2 (XMT)
Processing Units					
Clock (GHz)	2.30	2.53(base)	1.3	1.15	0.5
Sockets	4	2	30 SMs	14 SMs	128
Cores/socket	12	4	8 SPs/SM	32 SPs/SM	1
Threads/core	1	2	–	–	128
Total threads	48	16	240 SPs	448 SPs	16,384
Interconnect	HyperTransport-3	QPI	PCIe2	PCIe2	Seastar2
Interconnect (GB/s)	25.6	25.6	8	8	4.8
Memory System					
Cache structure	L1/L2/L3†	L1/L2/L3†	–	L1/L2†	cache-less
L1 (KB)/core:Inst/Data	64/64	32/32	–	48/SM*	–
L2 (KB)/core	512	256	–	768/SM	–
L3 (MB)/socket	12	8	–	–	–
Memory/socket (GB)	64	12	4	3	8
Total memory (GB)	256	24	3	3	1,024
Peak Bandwidth (GB/s)	42.7 (DDR3)	25.6 (DDR3)	102 (GDDR3)	144 (GDDR5)	86.4 (DDR1)
Software					
C Compiler	GCC 4.1.2	Intel 11.1	NVCC (CUDA 4.0)	NVCC (CUDA 4.0)	Cray C 6.5.0
Flags	-O3	-fast	-O3	-O3	-par
Thread scheduling	static	static	dynamic	dynamic	block-dynamic
Reference	[31]	[32]	[33]	[34]	[35]

TABLE I

Key Architectural Features: A SUMMARY OF KEY FEATURES OF THE PLATFORMS USED IN THIS PAPER.† SHARED CACHE. * 64 KB OF MEMORY IS CONFIGURED AS 48 KB OF L1 CACHE AND 16 KB AS SHARED MEMORY.

The AMD Opteron platform (**Magny-Cours**) is a 4-socket 12-core system with 256 GB of system memory. Each 12-core socket consists of two 6-core dies on a single package (multi-chip module) with separate memory controllers. The sockets are interconnected using HyperTransport-3 technology. Given the large size and non-uniform memory access (NUMA) costs it is important to consider allocation of memory and how software threads get pinned to physical cores.

The Intel Xeon platform (**Nehalem**) is a 2-socket 4-core system with 24 GB of system memory. Each core supports Simultaneous MultiThreading (SMT), allowing 2 threads to share the instruction pipeline. This feature is commercially known as HyperThreading. In order to assess the benefits of HyperThreading, we pin two threads to the same core using the capabilities provided by the Intel compiler. In addition to hyper-threading, Nehalem provides several advanced architectural features such as a new cache coherency protocol (MESIF), out-of-order execution, sophisticated branch prediction, and high bandwidth interconnect (QPI). Accordingly, we observe superior single-thread (serial) performance on this platform.

Tesla is the codename for the graphics processing unit (GPU) used in Tesla C1060 boards targeted for general computational purposes. The basic unit of computation on Tesla is a Streaming Multiprocessor (SM) that consists of: an Instruction Unit, 8 Streaming Processors (SPs), 2 Special Function Units (SFUs), a Double Precision Unit and a scratchpad memory (local memory) of 16 KB. Streaming Processors are the Arithmetic-Logic units (ALUs) that perform basic single precision and integer computations. The Instruction Unit fetches an instruction for a group of 32 threads, called a *warp*, which is then executed in a Single Instruction Multiple Thread (SIMT) fashion for multiple clock cycles using various resources of an SM. An instruction executes for a minimum of 4 clock cycles when simple single precision operations are used. Each SM maintains up to 32 warps (1024 threads) that are simultaneously active. Multiple SMs are integrated in a single GPU and connected, through a set of 64-bit wide GDDR3 memory controllers, to an external random access memory (named global memory) located on the board. In Tesla C1060, 30 SMs (for a total of 240 SPs) are interconnected through 8 memory controllers to 4 GB of global memory at a speed of 800 MHz. Thus, obtaining an overall memory bus width of 512 bits and reaching a peak bandwidth of 102 GB/s. When a warp issues a long latency operation to the global memory, another active warp is switched on the execution resources of the SM to cover this latency.

A GPU is programmed as an accelerator with its own memory pool, so the data needs to be moved to its global memory before and after the computation. This procedure is performed through a PCI-Express 2.0 bus that allows a peak bandwidth of 8 GB/s in both the directions. This bus may present a significant bottleneck for the overall performance of a GPU-accelerated algorithm. For obtaining high performance on Tesla, it is important to *coalesce* memory accesses (i.e., performing multiple operations with a single transaction). In Tesla, the maximum size of a transaction is 64 bytes. To coalesce memory operations,

memory accesses by threads in a half-warp (memory operations are issued for a group of 16 threads) should be sequential and aligned. If these rules are not respected, multiple memory transactions get issued that reduce the effective bandwidth available. Thus, irregular applications are challenging to implement on Tesla.

Fermi is the latest generation of NVIDIA GPUs. In comparison to previous generation of GPUs, Fermi retains general architectural principles and programming model. However, there are several significant changes compared to Tesla. Principal differences are in the organization of SMs and memory hierarchy. The SMs in Fermi are composed by 32 SPs, 4 SFUs and 2 Instruction Units. The Instruction Units fetch instructions from two different warps (still composed of 32 threads) which are then simultaneously issued on a group of 16 SPs for a minimum of 2 clock cycles. However, when double precision operations are issued, only one warp can be active at a given point of time. The number of warps-in-flight for each SM has been increased to 48. Each SM includes 64 KB of on-chip memory which can be configured as 48 KB of shared memory (scratchpad) and 16 KB of L1 cache, or as 16 KB of shared memory and 48 KB of L1 cache. The L1 caches of different SMs are not cache coherent. The SMs in Fermi are connected to a 768 KB L2 cache. The line size for L1 and L2 caches is 128 bytes. While the rules for memory accesses remain similar to Tesla, the size of each memory transaction has been increased to 128 bytes to match the line size of L1 and L2 caches. Caches have the potential to improve the performance of uncoalesced accesses to memory. Subsequent accesses to unaligned data already loaded in one of the caches are serviced at the throughput of the respective caches. In addition, L2 plays an important role when atomic memory operations are issued. In this paper, we use a Tesla C2050 board equipped with a Fermi processor comprising of 14 SMs (for a total of 448 SPs). The 14 SMs are connected to six 64-bit memory controllers for a total bus-width of 384 bits. The C2050 hosts 3 GB of GDDR5 memory at 1.5 GHz, which allows a peak bandwidth of 144 GB/s. While Fermi supports Error Correcting Code (ECC) for all levels of the memory hierarchy, it also provides the ability to disable ECC through a software switch. The software infrastructure on Fermi allows setting the sizes of L1 caches and local memories. It also allows the user to bypass L1 caches altogether. However, L2 caches cannot be bypassed since all memory transactions pass through them. We show performance differences arising from disabling ECC and L1 caches in Figure 4. In summary, Fermi is a superior alternative to Tesla for irregular algorithms such as the matching algorithm. Accordingly, we observe considerable improvements in performance.

The Cray **XMT** platform used in this paper consists of 128 Threadstorm (MTA2) processors interconnected using a high bandwidth 3D torus network (Cray SeaStar2). Each MTA2 processor consists of 128 thread streams (hardware threads) and a very long instruction pipeline (VLIW) with 21 stages. In contrast to SMT on the Nehalem, MTA2 uses *interleaved* scheduling of threads – at each cycle, an instruction is chosen from a different thread that is ready. MTA2 uses massive multithreading as a means to tolerate latency from memory operations as well as latencies generated from synchronization of threads. In contrast to a multilevel cache hierarchy on traditional platforms, the memory system on XMT is a flat cacheless system. The virtual global address space is built from physically distributed memory modules of 8 GB of DDR-1 memory on each processor, resulting in a total memory of 1 TB from 128 processors. Memory accesses are made uniform by using a hardware hashing mechanism that maps data randomly to memory modules in block sizes of 64 bytes. Each word (64 bits) of memory has special tag bits that are used to provide light-weight synchronization among threads. Details of extended memory semantics using these bits are presented in Section IV-B. The average latency of a memory access is 600 cycles with a worst-case latency of 1000 cycles. A 128 processor system has a sustained bandwidth of 86.4 GB/s. Unlike Nehalem, XMT lacks advanced architectural features such as branch prediction. It relies on multithreading to tolerate latencies from different sources including branches. A slow clock speed coupled with a long instruction pipeline results in a considerably slow serial performance. However, relatively better performance of XMT for irregular problems demonstrates the effectiveness of multithreading and large memory bandwidth over faster clock speeds and deeper cache hierarchies.

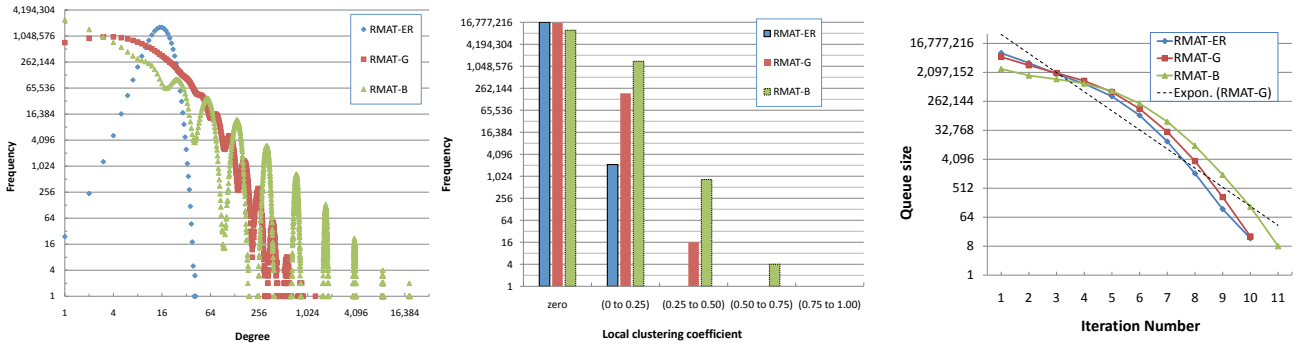
C. Datasets

Our dataset comprises of three instances of synthetic graphs and three instances of real-world data. Synthetic graphs are generated using the recursive matrix multiplication (R-MAT) algorithm proposed by Chakrabarti and Faloutsos [36]. Different classes of graphs can be generated by varying the four input parameters for probability in the R-MAT algorithm. While a large number of combinations are possible, we find that the following three input parameters cover a broad range of characteristics: (i) **RMAT-ER:** $(0.25,\ 0.25,\ 0.25,\ 0.25)$, (ii) **RMAT-G:** $(0.45,\ 0.15,\ 0.15,\ 0.25)$, and (iii) **RMAT-B:** $(0.55,\ 0.15,\ 0.15,\ 0.15)$. RMAT-ER stands for a class of Erdős-Rényi random graphs. The other two classes of graphs (RMAT-G and RMAT-B) have properties similar to real-world graphs with skewed normal distributions for vertex degrees and small world phenomenon of short average distance between pairs of vertices. The size of the graph generated by R-MAT algorithm is specified as a power of two, where the number of vertices is given by 2^{SCALE} and the number of edges is some multiple of the number of vertices (we use $8\times$ for our experiments). For the input parameter SCALE, we use a range of values from 23 to 27 on different platforms. The different sizes and key properties of the graphs used for experiments are summarized in Table II. The table represents the numbers for graphs generated on the Cray XMT using our own implementation of the R-MAT algorithm. However, for experiments on the Magny-Cours platform, we generate the graphs in memory using SNAP (Small-world Network Analysis and Partitioning) toolkit version 0.3 [37]. While the number of vertices remain the same, there are minor differences in the number of edges and connectivity due to differences in the pseudo-random numbers used. When cross comparisons between

Graph	Scale	No. Vertices	No. Edges	Max. Deg.	Variance	% Isolated vertices
RMAT-ER	23	8,388,608	67,108,782	39	16.00	0
	24	16,777,216	134,217,654	42	16.00	0
	25	33,554,432	268,435,385	41	16.00	0
	26	67,108,864	536,870,837	48	16.00	0
	27	134,217,728	1,073,741,753	43	16.00	0
RMAT-G	23	8,388,608	67,081,539	999	390.25	2.11
	24	16,777,216	134,181,095	1,278	415.72	2.33
	25	33,554,432	268,385,483	1,489	441.99	2.56
	26	67,108,864	536,803,101	1,800	469.43	2.81
	27	134,217,728	1,073,650,024	2,160	497.88	3.06
RMAT-B	23	8,388,608	66,738,577	26,949	6,834.34	29.22
	24	16,777,216	133,658,229	38,143	8,085.64	30.81
	25	33,554,432	267,592,474	54,974	9,539.17	32.34
	26	67,108,864	535,599,280	77,844	11,213.79	33.87
	27	134,217,728	1,071,833,624	111,702	13,165.52	35.37

TABLE II

Synthetic Graphs: DETAILS OF THE GRAPHS USED FOR EXPERIMENTS. GRAPHS WITH SCALE ≤ 24 WERE GENERATED ON THE XMT, STORED IN A TEXT FILE, AND USED ON ALL THE PLATFORMS. LARGER GRAPHS WITH SCALE ≥ 25 WERE GENERATED SEPARATELY ON MAGNY-COURS AND XMT. THE DETAILS SHOWN HERE ARE FOR THOSE GENERATED ON THE XMT. THEY VARY BY A SMALL MARGIN FROM THOSE GENERATED ON MAGNY-COURS.



(a) Degree distribution (b) Distribution of local clustering coefficient (c) Queue-Size

Fig. 1. **Characteristics of Data** (SCALE $= 24$)**:** (a) Degree distribution of the three RMAT graphs. The last datapoint for RMAT-B with degree=38143 and frequency=1 is not shown. The plot is on a log-log scale. (b) Distribution of local clustering coefficients of the three RMAT graphs. (c) Queue sizes during the execution of the algorithm. Iterations are as defined in Section III-A. The black dashed trend line (exponential) indicates that the amount of work roughly decreases by half after each iteration.

architectures are made, the graphs are generated on the XMT, saved to disk in plain text format, and reused on the other platforms. A positive integer weight is associated with each edge. For our experiments we assigned random weights ranging from zero to number of vertices.

The overall sizes of the three graph variants are similar, but they vary widely with respect to degree distribution and clustering coefficient. The local clustering coefficient of a vertex is given by the ratio of the actual number of edges between its neighbors to the maximum number of edges possible between the neighbors. The clustering coefficient of a graph is the average of local clustering coefficients over all the vertices. Degree distribution and local clustering coefficient for the three variants at SCALE $= 24$ are shown in Figure 1. As expected, RMAT-ER has a mean value of 16 and small variance resembling normal distribution. The other two variants have a large distribution with RMAT-B having a maximum degree of $38,143$. The average clustering coefficient for ER, G, and B are 10^{-7}, 12^{-6}, 34^{-5} respectively. We also observe that the number of isolated vertices (with zero degree) is high for RMAT-B, over 30% of vertices are isolated. In our experiments, we remove duplicate edges and self-loops (this is what leads to a variation in the number of edges between different classes of graphs), but retain isolated vertices. Given the large variation in graph properties, we expect significant impact not only on the parallel performance of algorithms, but also on the behavior of the sequential algorithm (in terms of the number of edges that get matched). In particular, we expect a large variation in the degree distribution in RMAT-B to have an adverse effect on the performance of GPUs caused by load imbalances. We also note that in our experiments we randomly shuffle the vertex indices in the graphs as generated by by R-MAT algorithm. The motivation for this shuffling is to prevent vertices with larger degrees getting concentrated towards vertices with smaller indices (the top-left quadrant of the R-MAT algorithm). This shuffling also makes the synthetic graphs realistic since we do not expect correlation between vertex identities to its degree.

A small set of graphs from real-world data are chosen to enable comparative study of our implementations on the GPUs with existing literature. The properties of real-world dataset are summarized in Table III.

Graph	No. Vertices	No. Edges	Maximum Degree	Average Degree	Variance	% Isolated vertices
cit-patent	3,774,768	33,037,894	793	8.75	110.06	0
soc-live	4,847,571	42,851,237	20,333	17.68	2704.35	0.02
Usa-Roadmap	23,947,347	28,854,312	9	2.41	0.86	0

TABLE III
Real-World Graphs: WERE OBTAINED FROM THE STANFORD LARGE NETWORK DATASET COLLECTION MAINTAINED BY JURE LESKOVEC AND THE COLLECTION FROM DIMACS CHALLENGE-9. DUPLICATE ENTRIES ARE REMOVED AND RANDOM WEIGHTS ARE ASSIGNED TO THE FIRST TWO INSTANCES.

IV. PARALLEL IMPLEMENTATION

In this section we provide details of parallel implementations of the Locally-Dominant Algorithm for $^1/_2$-approx weighted matching that was introduced in Section III-A. We first describe a generic queue-based implementation suitable for all the platforms, and then present a novel dataflow algorithm targeted specifically for the Cray XMT. The dataflow algorithm is motivated by general weaknesses of the shared work-queue approach such as the need for synchronization and variation in the amount of work that can be done in parallel leading to loss of performance.

A. Parallel Queue-based Implementation

The parallel queue-based implementation, illustrated in Algorithm 3, is a parallel implementation of the Locally-Dominant Algorithm introduced in Section III-A. Similar to the serial version, we divide the parallel algorithm into two phases. Phase-1 of the algorithm (Lines 5 – 18 in Algorithm 3) consists of two parallel sections. Each section iterates once over all the vertices in a graph. While the first section (Line 5) finds a candidate mate for each vertex, the second section (Line 15) checks for locally-dominant edges and adds them to the matching. Endpoints of matched edges (matched vertices) are added to a queue (Q_C). Phase-2 of the algorithm (Lines 21 – 27 in Algorithm 3) is executed several times by iterating over all the vertices in the queue until no new edges get matched (the queue becomes empty).

Algorithm 3 Parallel Queue-based Implementation. *Input*: graph $G = (V, E)$. *Output*: A matching M represented in vector mate. *Data structures*: a queue Q_C of vertices for processing in the current step, and a queue Q_N of vertices that will be processed in the next step. Both the queues contain matched vertices; a vector candidate of size $|V|$ that contains the id of the current heaviest neighbor of each vertex.

```
1:  procedure PARALLEL-QUEUE(G(V, E), mate)
2:      Q_C ← ∅
3:      Q_N ← ∅
4:      — — Phase-1 — —
5:      for each u ∈ V in parallel do                          ▷ OpenMP pragmas/CUDA kernels
6:          mate[u] ← ∅
7:          candidate[u] ← ∅
8:          max_wt ← −∞
9:          max_wt_id ← ∅
10:         for each v ∈ adj(u) do
11:             if (mate[v] = ∅) AND (max_wt < w(e_{u,v})) then   ▷ Use vertex ids to break ties.
12:                 max_wt ← w(e_{u,v})
13:                 max_wt_id ← v
14:         candidate[u] ← max_wt_id
15:     for each u ∈ V in parallel do                          ▷ OpenMP pragmas/CUDA kernels
16:         if candidate[candidate[u]] = u then                ▷ Found a locally dominant edge
17:             mate[u] ← candidate[u]
18:             Q_C ← Q_C ∪ {u}                                ▷ Use atomic memory operation
19:
20:     — — Phase-2 — —
21:     while Q_C ≠ ∅ do
22:         for each u ∈ Q_C in parallel do                    ▷ OpenMP pragmas/CUDA kernels
23:             for each v ∈ adj(u) do
24:                 if candidate[v] = u then                   ▷ Process v only if u is its candidate mate
25:                     PROCESSVERTEX(v, Q_N)
26:         Q_C ← Q_N                                          ▷ The new set of matched vertices
27:         Q_N ← ∅
```

On the multicore platforms, the algorithm is parallelized using OpenMP as the programming model. We use compressed sparse row (CSR) format (data structure) to store graphs in memory. In this format the neighborhood of each vertex is stored contiguously in memory. Thus, the benefit from caching exists when adjacency lists are accessed on platforms with cache

hierarchies. The shared data structures for which the threads need to synchronize accesses are the two queue data structures, Q_C and Q_N. Different threads simultaneously add those vertices to a queue that get matched in a given iteration of the algorithm. One way of implementing synchronous queue operations is to use critical sections. However, critical sections result in severe loss of performance. On the x86 platforms (Magny-Cours and Nehalem) we use an intrinsic atomic operation `__sync_fetch_and_add()` to find a unique position to add to the queue (the tail of the queue) for each thread. A similar atomic operation, `int_fetch_add()`, is provided on the XMT. The use of two queues (Q_C and Q_N) leads to better performance – while from Q_C we efficiently process (read) the vertices matched in the previous iteration, we enqueue (write) vertices matched in the current iteration in Q_N. The threads synchronize at the end of each iteration, and in serial, we assign the contents of Q_N to Q_C (via pointer swap) and set Q_N to empty (Lines 26 and 27 in Algorithm 3).

The use of shared work-queues is similar to the use of queues in a level-synchronous breadth-first search to store and process the frontier. However, given that the vertices that need to be processed in the next iteration are determined based current matching and edge-weights, generic prefetching techniques to exploit caches are not applicable for this algorithm. The amount of parallelism at any given iteration depends on the size of the queue and the parallel efficiency depends on the pattern and speed of atomic operations to add new elements to the queue. The size of the queue as the algorithm progresses for different inputs are captured in Figure 1(c).

CUDA Implementation: Parallel implementation on GPUs is similar to the OpenMP implementation on multicore platforms. Again, we divide the algorithm into two phases and have three GPU kernel routines for Algorithm 3. The first routine assigns the heaviest neighbor for each vertex, the second routine checks if a locally-dominant edge is found, and if so, adds that edge to M and the two endpoints to the queue. The third kernel call combines the functionality of finding the heaviest neighbor and a test for local dominance of an edge. Instead of processing one vertex, as shown in Algorithm 2, we process a set of vertices stored in Q_C. The newly matched vertices are added to Q_N. We use the atomic memory operation `atomicAdd()` on the GPUs. Depending on the size of a queue, the grid dimensions are determined on the host machine that places the kernel call. Within the kernel, each thread processes a vertex based on its thread id. At the beginning, all the data structures are copied to GPU memory and they reside there for the entire duration of the algorithm. After each iteration, only the size of Q_N is sent back to the host machine that decides if a kernel call needs to be made and the sizes of grid dimensions of such a call. Due to a lack of spatial and temporal locality of memory accesses in graphs, achieving high performance on GPUs is challenging.

Three factors affect the performance of our implementation on GPUs: (i) *Load balancing within a warp*: The same instruction is issued for all the 32 threads in a warp, and therefore, the performance of a warp is limited by the slowest thread even when other threads finish their work ahead of time. For example, the thread processing the vertex with the largest degree might be the slowest. An approach to solve this imbalance is to block the work into groups and use independent threads for different blocks. We observed about 20 percent improvement for breadth-first search in RMAT-B with this approach. Since there is a small overhead for blocking, there was no performance penalty for well balanced inputs like RMAT-ER. While such an approach is well suited for straightforward algorithms like breadth-first search, it is not suitable for algorithms like matching that need synchronization between threads that process the adjacency list of a vertex. (ii) *Coalescing memory accesses*: On Tesla, memory accesses by threads of the same half-warp should be aligned and to contiguous blocks in memory. On Fermi, caches reduce overheads of non-sequential accesses only if threads of the same warp access data from the same cache line. However, in our implementation the vertices get added to the queue in a random order, and thus, each thread in a warp will access random memory locations. An approach to address this problem would be maintain the work queues in an order. However, the potential benefits may not be worth the effort to reorder the queues. (iii) *Branch divergence*: Since all the threads in a warp share the same instruction, if any one of these threads in a warp takes a different branch of an `if` condition, all the other threads are forced to wait for its completion. In other words, both the branches of an `if` condition get executed *serially*. The GPU kernel call to process a vertex has several `if` conditions that check if a neighbor has already been matched or not, and if it is the current-heaviest neighbor or not. Restructuring of an algorithm and implementation might alleviate this problem to a certain extent. Future architectural improvements like branch prediction will also have a pronounced impact on performance.

To summarize, solving each of these problems is a significantly hard challenge by itself, and in concert, they determine the overall performance of a GPU implementation. In our experimental results, presented in Figure 4, we observe the adverse impact of load imbalances in RMAT-B relative to RMAT-ER, and Cit-Patent relative to Soc-Live. The compressed data structures used in our implementation alleviate irregular accesses by storing adjacency lists of a vertex in a contiguous block. In particular, the benefits are evident in Phase-1 when vertices are processed in an order, but in Phase-2 vertices are processed in a random order and therefore lead to poor performance. Optimization for branch divergence is challenging.

B. Parallel Dataflow-based Implementation

In this section we provide a novel dataflow-based implementation of the Locally-Dominant Algorithm. The XMT supports light-weight synchronization by associating every word of memory (8 Bytes) with the following additional bits: a full/empty bit, a pointer forwarding bit, and two trap bits. Of particular interest to this study is the full/empty bit. XMT provides several extended memory semantic operations to manipulate the full/empty bit. The four key operations that are sufficient to implement the matching algorithm (and several other graph algorithms) are:

- `purge`: sets the full/empty bit to empty and the value to zero.
- `readff`: reads a memory location only when the full/empty bit is full and leaves the bit full when read finishes.
- `readfe`: reads a memory location only when the full/empty bit is full and leaves the bit empty when read finishes.
- `writeef`: writes to a memory location only if the full/empty bit is empty, and flips the bit to full when the write finishes.

While a "single-producer single-consumer" model can be efficiently implemented using `writeef` and `readfe`, a "single-producer multiple-consumer" model can be implemented using `writeef` and `readff`. A dataflow implementation using the "single-producer multiple-consumer" model is presented in Algorithm 4. The general structure and behavior of the dataflow-based algorithm is similar to the queue-based algorithm, except that threads do not have a shared work-queue to synchronize their operations. Instead, each thread will work on a block of vertices (Line 8 in Algorithm 4) by making a call to Procedure PROCESSVERTEXDF (presented in Algorithm 5) for each vertex that has not been processed before. For simplicity of description, we assume that the adjacency lists are maintained in a non-increasing order (vertex-ids are used to break ties consistently).

In Procedure PROCESSVERTEXDF, the thread processing vertex s sets the candidate mate of s to its current-heaviest neighbor and attempts to read if that neighbor wants to match with s. If a positive answer is observed from this neighbor, then the thread adds this edge to the current set of matched edges and proactively sets a negative response to its remaining neighbors (Lines 10 and 11). If a negative answer is observed from this neighbor, then the thread proceeds to the next heaviest neighbor. The number of vertices to be processed is generally much larger than the number of threads available on a system. Therefore, passively waiting for another thread to complete processing the desired vertex could lead to a dead-lock due to exhaustion of resources. While this problem can be addressed in multiple ways, we use *recursion* in combination with an additional data structure state that maintains a state (processed or not) for each vertex. The recursive call is shown in Line 5 of Algorithm 5.

In the absence of a shared work-queue, the algorithm unravels efficiently on the XMT. Extended memory semantic operations are executed efficiently on the XMT. When the tag-bits are in a desired state, memory operations complete in a single clock cycle. Otherwise, the request is returned to the ready queue for a retry. Similar to general memory operations, XMT tolerates the latencies arising from these operations via multithreading. Thus, in comparison to the queue-based implementation, we observe superior performance of the dataflow algorithm in our experiments. The results are presented in Figure 5.

Algorithm 4 Parallel Dataflow-based Implementation. *Input*: graph $G = (V, E)$. *Output*: A matching M represented in vector mate. *Data structures*: a vector candidate of size $2|E|$. For an edge $e_{u,v}$ the response from u to v, and vice-versa, is stored in candidate; a vector state of size $|V|$ represents if a vertex has been processed or not (zero implies an unprocessed state, a number > 0 implies that it has been processed).

```
1:  procedure PARALLEL-DATAFLOW(G(V, E), mate)
2:      for each u ∈ V in parallel do
3:          mate[u] ← ∅
4:          state[u] ← 0
5:      for each e_{u,v} ∈ E in parallel do
6:          writeef(candidate[u → v], 0)        ▷ Set full/empty bit to empty and value to zero
7:          writeef(candidate[v → u], 0)
8:      for each u ∈ V in parallel do
9:          isProcessed ← int_fetch_add(state[u], 1)
10:         if isProcessed = 0 then             ▷ Process only if u has not been processed before
11:             PROCESSVERTEXDF(u)
```

Algorithm 5 ProcessVertexDF

```
1:  procedure PROCESSVERTEXDF(s)
2:      for each t ∈ adj(s) in non-increasing order of weights do
3:          writeef(candidate[s → t], 1)        ▷ Set full/empty bit to full and value to 1
4:          if int_fetch_add(state[t], 1) = 0 then
5:              PROCESSVERTEXDF(t)              ▷ Proactively process the other end
6:          if readff(candidate[t → s])=1 then  ▷ Wait until full/empty bit becomes full
7:              mate[s] ← t                     ▷ Found a locally dominant edge
8:              mate[t] ← s
9:              break
10:     for each t_np ∈ adj(s) not already processed do   ▷ Set the value to zero for remaining neighbors of s
11:         candidate[s → t_np] ← 0
```

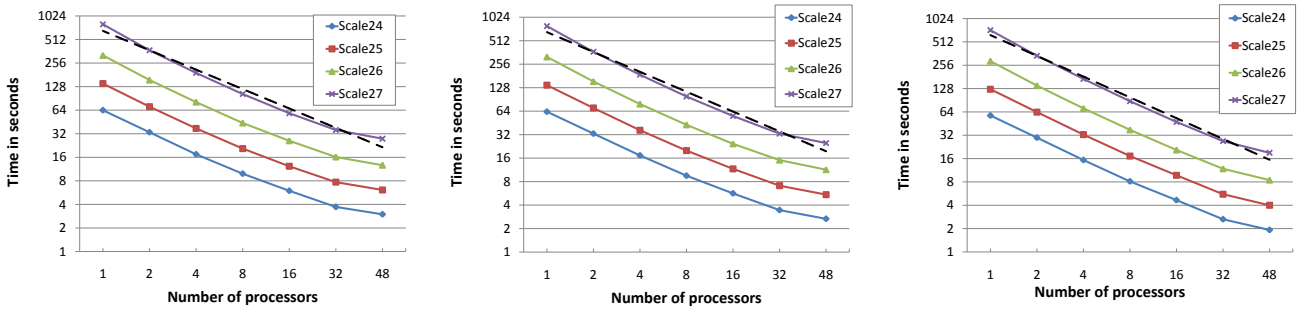
V. EXPERIMENTAL RESULTS

In this section, we provide experimental results on the performance of parallel $^1/_2$-approx algorithm on five platforms presented in Section III-B. Three classes of synthetic graphs of various sizes are used as input for experiments across the

platforms, and three instances of real-world data are used as input on the GPU platforms. Details on dataset are provided in Section III-C. We provide strong and weak scaling results on the Magny-Cours and XMT platforms for R-MAT graphs up to a billion edges (SCALE $= 24$ to 27). In addition, on the XMT we provide results for three variants of implementation: queue-based, queue-based with sorted adjacency lists, and dataflow-based. We provide strong scaling results on the Nehalem (SCALE $= 24$). Results from GPU platforms are normalized with the performance of one core of the host platform. In general, we avoid excessive fine-tuning of software for any given architecture. Instead, we strive for uniformity of the algorithm and code where possible. An exception to this goal is the dataflow algorithm specially targeted for the XMT. We now present the results and brief discussion on different aspects of performance.

General observations: Basic insight on the performance differences for the three classes of graphs (RMAT-ER, RMAT-G, RMAT-B) can be obtained by considering the cardinality as the algorithm progresses. The cardinality of matching (the number of edges in matching) represents the amount of work that can be parallelized, and therefore, determines the utilization of system resources. The cardinality, determined by the size of Q_C, at each iteration in the execution of the algorithm, for the three classes of graphs (with SCALE $= 24$) is provided in Figure 1(c). We provide the size of Q_C on the Y-axis in log-scale and the iterations are provided on the X-axis. It can be observed that an upper bound on the work roughly decreases by half after each iteration. The final cardinality of the three matchings as a percentage of the number of vertices is as follows: 94.12% for RMAT-ER, 81.70% (83.46% by accounting for isolated vertices) for RMAT-G, and 44.24% (63.94% by accounting for isolated vertices) for RMAT-B. The cardinality of matching after Phase-1 as a percentage of the final cardinality is as follows: 53.14% (ER), 46.33% (G), and 36.06% (B). We observe that the cardinality of matching for RMAT-B is much smaller than that for others. In addition, RMAT-B has more iterations with smaller amounts of work towards the end (refer Figure 1(c)). A smaller amount of work for RMAT-B in comparison to other classes, suggest that the total runtime for RMAT-B should be smaller than the other two classes. However, a longer tail with less work towards the end leads to inefficient use of resources on platforms such XMT and GPUs that have a large number of threads.

Magny-Cours: Strong and weak scaling results on Magny-Cours are presented in Figure 2. In our experiments, we pin the threads to cores using the environment variable `GOMP_CPU_AFFINITY` by using a round-robin scheme to place each consecutive thread on a different socket and a different chip within a socket. For example, thread-0 would be pinned to chip-0 on socket-0, followed by thread-1 to chip-0 on socket-1, ..., thread-4 to chip-1 on socket-0, and so on. Note that each socket has two chips, and each chip has six cores. In addition, we use the command `numactl --interleave` to enable NUMA-aware memory allocation on the system. We observe excellent scalability for all the three classes of graphs at different problem sizes. We observe minor degradation of performance at 48 cores, particularly for smaller graph sizes. While the Magny-Cours platform provides excellent scalability, the absolute runtime as compared to Nehalem and XMT is slower by a factor of 2.

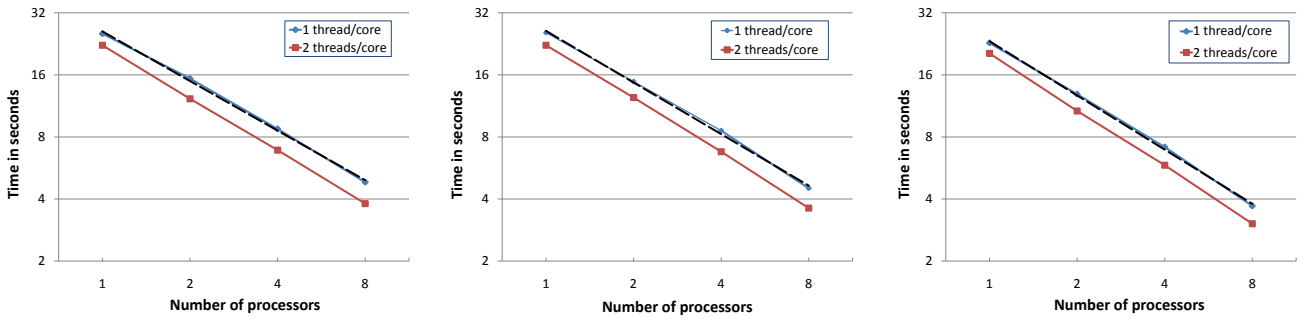


(a) RMAT-ER (b) RMAT-G (c) RMAT-B

Fig. 2. **Scaling on Magny-Cours:** Performance of queue-based implementation on AMD Magny-Cours with up to 48 cores. The threads are pinned to the cores such that the total memory bandwidth is maximized. The black dashed lines represent linear scaling.

Nehalem: Strong scaling results on Nehalem are presented in Figure 3. In our experiments, we used the environment variable `KMP_CPU_AFFINITY` to pin the threads to cores. Similar to Magny-Cours, we scatter the threads across the system to maximize available memory bandwidth. In order to evaluate the benefits of Hyper-Threads, we pin two threads to each core (shown with red lines in the figure). We observe a performance gain of 1.2 to 1.4 relative to the performance of one thread per core. The black dashed lines indicate linear speedup. We observe excellent scaling on Nehalem, which is enabled by advanced architectural features coupled with large memory bandwidth.

Tesla and Fermi: We provide performance results on the two GPU platforms, Tesla and Fermi, in Figure 4. The bars represent speedup relative to the performance of one core of the host platform. The host consists a 2-socket 6-core Intel Nehalem with 24 GBs of system memory. Note that the host processor is a new platform relative to the quad-core Intel Nehalem platform presented earlier. In our implementation, we transfer the graph data structures, we well as all the other supporting data structures to the GPU memory at the beginning of the execution. The data structures remain in the GPU memory during the entire execution. For each iteration of the **while** loop, the size of the queue Q_N is sent back to the host, which swaps the queue headers, determines the grid sizes and invokes a GPU kernel. In order to fit larger graphs on the

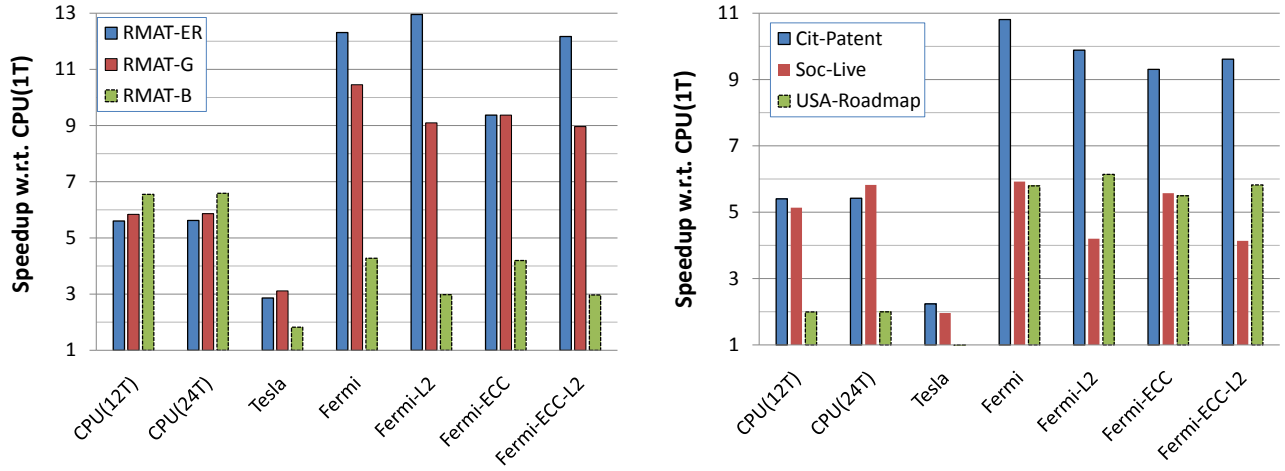


(a) RMAT-ER (b) RMAT-G (c) RMAT-B

Fig. 3. **Scaling on Nehalem:** Performance of queue-based implementation on Nehalem for the three variants of RMAT graphs (SCALE = 24). The two threads per core lines indicate the use of Hyper-threads. The threads are pinned to cores such that the total memory bandwidth is maximized. The black dashed lines represent linear scaling.

GPU memory, the graph data structures are maintained in 32-bit primitive data types (integers for vertex ids and doubles for weight). The data structures on the host machine are also maintained in 32-bit primitives. Note that while the results on GPUs and the host machine are obtained using 32-bit primitives, the results on x86 (Opteron and Nehalem) and XMT are obtained using 64-bit primitives. On the host, we used Intel 11.1 compilers with `-fast` switch, and used the environment variable `KMP_CPU_AFFINITY` to pin the threads to cores such that the total memory bandwidth is maximized.

From the results presented in Figure 4 we make the following observations. (i) A significant speedup on Fermi relative to Tesla is enabled by efficient atomic memory operations that are executed in L2 on Fermi compared to the main memory in Tesla, and a larger number of cores although at a slightly lower clock speed; (ii) A significant performance gain on Fermi compared to 12-core and 24-core runs on the host machine. While we observe improvements in the absolute times for single core runs, the excellent scaling observed on Nehalem (Figure 3) is not repeated on the host machine. The relative increase in the number of cores without a corresponding increase in memory bandwidth and small problem sizes are the reasons for this lack of scaling on the host platform; (iii) A large variation in vertex-degree (RMAT-B) has an adverse impact on performance for both Tesla and Fermi; and (iv) Disabling L1 caches improves performance for well balanced work loads (RMAT-ER). This behavior can be attributed to the irregularity of memory accesses that thrash on a relatively small L1 cache.



(a) GPU-Synthetic (b) GPU-Real-World

Fig. 4. **Scaling on GPUs: (a)** Relative performance of GPUs on synthetic graphs(SCALE = 23). **(b)** performance on real-world instances. The runtimes are scaled with the performance of one core (thread) of the host machine (Intel Nehalem). The bars represent performance on 12-cores (CPU(12T)), 24-cores(CPU(24T)), Tesla, and Fermi respectively. The first two bars for Fermi represent results with ECC turned off, followed by two bars with ECC turned on. The two bars with a suffix L2 represent runtimes where caching on L1 is disabled.

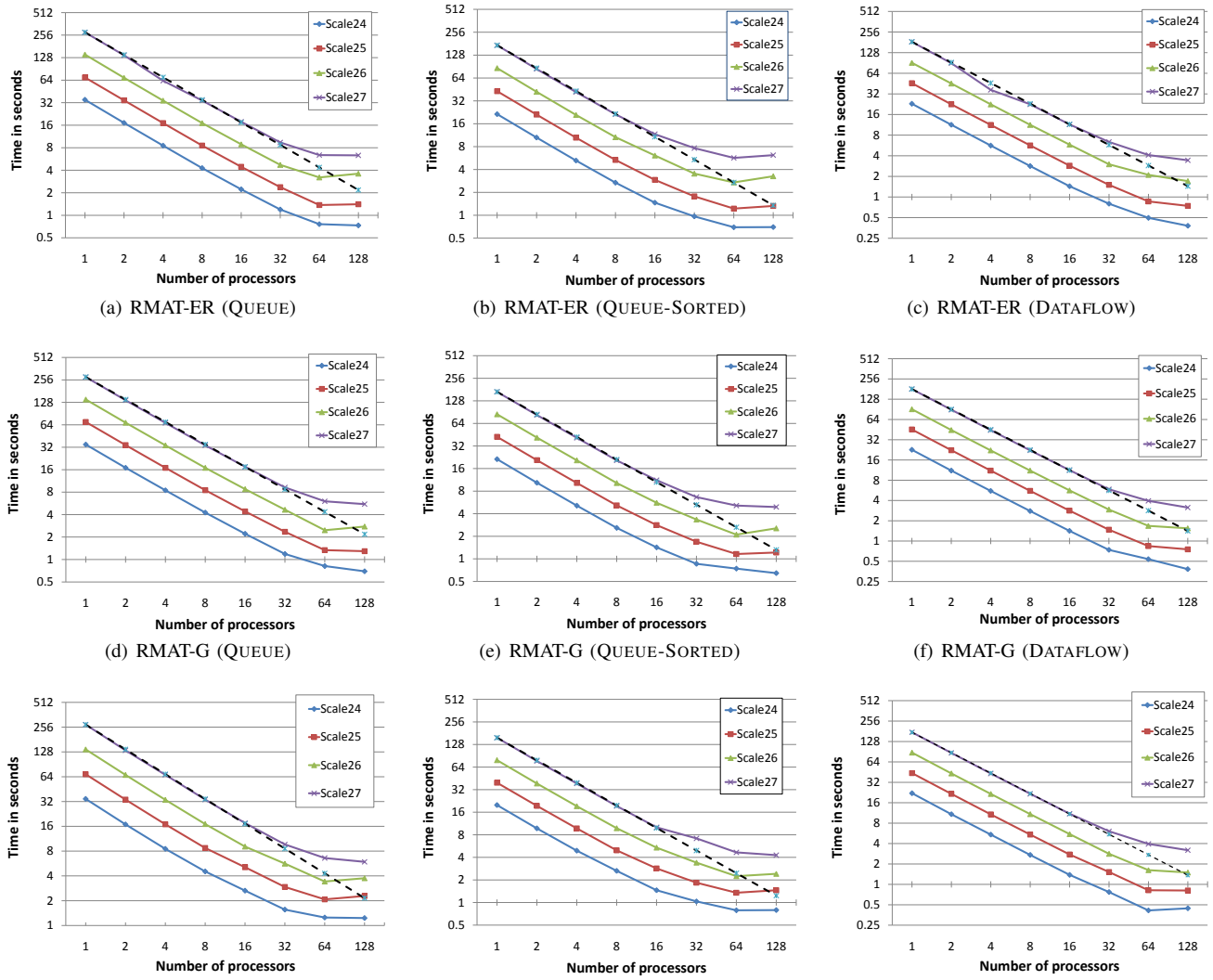
XMT: Strong and weak scaling results on the XMT using the three classes of graphs and three variants of implementation are presented in Figureñeff:xmt-scale. Results for the queue-based implementation are provided on the left column. We observe good scaling for up to 32 processors, and see a degradation of scaling for 64 and 128 processors. The middle column shows results for queue-based implementation where the adjacency lists are maintained in a sorted (non-increasing) order. We observe that sorting speeds the runtime by a factor of two with respect to the first variant (unsorted). However, sorting itself is much more expensive than matching. We present the results here for pedagogical reasons only. Results for the dataflow implementation

are presented in the column on right.

In our experiments, we request about 100 threads (thread-streams) on each processor and use block-dynamic scheduling of threads. The flat cacheless memory structure, extended memory semantics, and compiler-driven optimizations make programming on the XMT the easiest among the five platforms.

We observe that the dataflow implementation scales better than the queue-based implementation. Due to large amounts of concurrency, we observe memory hot-spotting on the queue, Q_N, that results in a loss of performance. The autonomy induced from the absence of shared work-queues enable the dataflow algorithm to scale better.

In order to determine the influence of the range of edge-weights on performance, we experimented with different ranges of weights. While the queue-based implementation was sensitive to the range, the dataflow-based implementation was stable and provided about a factor of two speedup for smaller ranges of weights over larger ranges. In summary, we observe excellent speedups on the XMT for all three variants of the implementation on all three classes of inputs.



(a) RMAT-ER (QUEUE) (b) RMAT-ER (QUEUE-SORTED) (c) RMAT-ER (DATAFLOW)

(d) RMAT-G (QUEUE) (e) RMAT-G (QUEUE-SORTED) (f) RMAT-G (DATAFLOW)

(g) RMAT-B (QUEUE) (h) RMAT-B (QUEUE-SORTED) (i) RMAT-B (DATAFLOW)

Fig. 5. **Scaling on the XMT:** Performance of three implementations of the matching algorithm – queue, queue-sorted and dataflow (left to right resp.) on three variants of RMAT graphs at different scales detailed in Table II. Both strong and weak scaling are captured in the plots; both axes are in logarithmic scale. We used block-dynamic scheduling and requested a maximum of 100 streams (threads) per processor. The black dashed lines represent linear scaling.

VI. CONCLUSIONS AND FUTURE WORK

We presented multithreaded implementations for the $^1/_2$-approx weighted matching problem on state-of-the-art multicore and manycore platforms, and a massively multithreaded architecture. Using a carefully chosen set of synthetic and real-world graphs we demonstrated scalable performance across the platforms. Matching is an important graph problem with numerous applications in scientific computing. Using irregular nature of this problem, we explored several architectural features that

enable efficient execution of irregular problems. We also presented a novel dataflow algorithms for the approximate matching problem that is suitable for architectures like the Cray XMT.

Based on our current work and experimental results, we make the following broad conclusions:

- *Structure* of a graph, such as degree distributions and local clustering, as well as its numerical properties, such as the range of edge-weights, play a dominating role in determining the performance of serial and parallel algorithms. This is evident from our results using inputs with different characteristics. These effects are pronounced on Fermi with about four times difference in performance between RMAT-ER and RMAT-B for a similar size of input.
- Compared to cache hierarchies, *multithreading* provides an efficient means to tolerate latencies resulting from irregular memory accesses and synchronization between threads. Relatively superior performance of XMT relative to Opteron (about five times faster clock speed) supports this conclusion.
- *Architectural* support for light-weight synchronization will enable efficient implementation of fine-grained parallelism in applications that lack coarse-grained parallelism. Graph algorithms in particular will benefit from these features. As observed in our experiments, a large improvement in the performance of Fermi over Tesla is enabled in part by efficient atomic memory operations. Tag-bits and extended memory semantics on the XMT enable a different kind of thinking for algorithm design as well as better performance (dataflow algorithm).
- When supported by hardware features, for example efficient atomic memory operations, CUDA provides a relatively simple programming model with considerable performance benefits for some types of inputs (for example RMAT-ER). However, achieving good performance for difficult inputs on these systems remains challenging.

By contrasting and comparing the performance of approximate matching on different platforms, we expect the insights from this work to benefit the design and use of future generations of manycore architectures.

In the near future, we plan to extend this work along the following directions. We plan to implement optimal algorithms for weighted and unweighted matching problems on shared-memory platforms. Optimal algorithms have numerous applications but are challenging to implement. We plan to extend the work on GPUs along two directions: capability to handle load imbalances within a warp due to variation in vertex-degrees and scalable implementations on multi-GPU systems. We plan to explore NUMA-aware (Non-Uniform Memory Access) implementations that are relevant to current and emerging architectures. Finally, we plan to combine the work presented in this paper with our earlier work on distributed-memory systems using a hybrid programming approach.

ACKNOWLEDGMENTS

Funding for this work was provided by the Center for Adaptive Super Computing Software - MultiThreaded Architectures (CASS-MT) at the U.S. Department of Energy's Pacific Northwest National Laboratory. PNNL is operated by Battelle Memorial Institute under Contract DE-ACO6-76RL01830. Additional funding for this project was provided by the U.S. Department of Energy through the CSCAPES Institute (grants DE-FC02-08ER25864 and DE-FC02-06ER2775). We acknowledge many contributions from Florin Dobrian, Assefaw Gebremedhin, Fredrik Manne and Umit Catalyurek. We also thank the anonymous referees for their valuable comments that helped us improve the manuscript.

REFERENCES

[1] F. Manne and R. H. Bisseling, "A parallel approximation algorithm for the weighted maximum matching problem," in *The Seventh International Conference on Parallel Processing and Applied Mathematics*, 2007, pp. 708–717.

[2] I. S. Duff and J. Koster, "The design and use of algorithms for permuting large entries to the diagonal of sparse matrices," *SIAM J. Matrix Anal. Appl.*, vol. 20, no. 4, pp. 889–901, 1999.

[3] X. S. Li and J. W. Demmel, "Making sparse Gaussian elimination scalable by static pivoting," in *Supercomputing '98: Proceedings of the 1998 ACM/IEEE conference on Supercomputing (CDROM)*. Washington, DC, USA: IEEE Computer Society, 1998, pp. 1–17.

[4] O. Schenk, A. Wächter, and M. Hagemann, "Matching-based preprocessing algorithms to the solution of saddle-point problems in large-scale nonconvex interior-point optimization," *Comput. Optim. Appl.*, vol. 36, pp. 321–341, April 2007.

[5] A. Pinar, E. Chow, and A. Pothen, "Combinatorial algorithms for computing column space bases that have sparse inverses," *Electronic Transactions on Numerical Analysis*, vol. 22, pp. 122–145, 2006.

[6] A. Pothen and C.-J. Fan, "Computing the block triangular form of a sparse matrix," *ACM Trans. Math. Softw.*, vol. 16, no. 4, pp. 303–324, 1990.

[7] G. Karypis and V. Kumar, "Analysis of multilevel graph partitioning," in *Supercomputing '95: Proceedings of the 1995 ACM/IEEE Conference on Supercomputing (CDROM)*. New York, NY, USA: ACM, 1995, p. 29.

[8] S. Belongie, J. Malik, and J. Puzicha, "Shape matching and object recognition using shape contexts," *IEEE Trans. Pattern Anal. Mach. Intell.*, vol. 24, pp. 509–522, April 2002.

[9] K. Asanovic, R. Bodik, B. C. Catanzaro, J. J. Gebis, P. Husbands, K. Keutzer, D. A. Patterson, W. L. Plishker, J. Shalf, S. W. Williams, and K. A. Yelick, "The landscape of parallel computing research: a view from Berkeley," Electrical Engineering and Computer Sciences, University of California at Berkeley, Tech. Rep., Dec. 2006.

[10] A. Lumsdaine, D. Gregor, B. Hendrickson, and J. W. Berry, "Challenges in parallel graph processing," *Parallel Processing Letters*, vol. 17, no. 1, pp. 5–20, 2007.

[11] B. Hendrickson and J. W. Berry, "Graph analysis with high-performance computing," *Computing in Science and Engineering*, vol. 10, no. 2, pp. 14–19, 2008.

[12] A. Abou-Rjeili and G. Karypis, "Multilevel algorithms for partitioning power-law graphs," in *Parallel and Distributed Processing Symposium, 2006. IPDPS 2006. 20th International*, april 2006, p. 10 pp.

[13] G. Cong and D. Bader, "Techniques for designing efficient parallel graph algorithms for SMPs and multicore processors," in *Lecture Notes in Computer Science*, vol. 4742. Springer, 2007, pp. 137–147.

[14] J. Nieplocha, A. Márquez, J. Feo, D. Chavarría-Miranda, G. Chin, C. Scherrer, and N. Beagley, "Evaluating the potential of multithreaded platforms for irregular scientific computations," in *CF '07: Proceedings of the 4th International Conference on Computing Frontiers*. New York, NY, USA: ACM, 2007, pp. 47–58.
[15] M. Halappanavar, "Algorithms for vertex-weighted matching in graphs," Ph.D. dissertation, Old Dominion University, Norfolk, VA, 2009.
[16] L. Lovasz, *Matching Theory (North-Holland mathematics studies)*. Elsevier Science Ltd, 1986.
[17] B. Monien, R. Preis, and R. Diekmann, "Quality matching and local improvement for multilevel graph-partitioning," *Parallel Comput.*, vol. 26, no. 12, pp. 1609–1634, 2000.
[18] A. Schrijver, *Combinatorial Optimization - Polyhedra and Efficiency*. Springer, 2003.
[19] D. Avis, "A survey of heuristics for the weighted matching problem," *Network*, vol. 13, pp. 475–493, 1983.
[20] R. Preis, "Linear time $\frac{1}{2}$-approximation algorithm for maximum weighted matching in general graphs," in *16th Ann. Symp. on Theoretical Aspects of Computer Science (STACS)*, 1999, pp. 259–269.
[21] D. E. Drake and S. Hougardy, "A simple approximation algorithm for the weighted matching problem," *Inf. Process. Lett.*, vol. 85, no. 4, pp. 211–213, 2003.
[22] J.-H. Hoepman, "Simple distributed weighted matchings," *CoRR*, vol. cs.DC/0410047, 2004.
[23] U. Catalyurek, J. Feo, A. Gebremedhin, M. Halappanavar, and A. Pothen, "Multithreaded algorithms for graph coloring," 2011, submitted to a journal. Invited presentation at SIAM Conference on Computational Science and Engineering (CSE11).
[24] D. A. Bader and K. Madduri, "Designing multithreaded algorithms for breadth-first search and st-connectivity on the Cray MTA-2," in *Proceedings of the 2006 International Conference on Parallel Processing*, ser. ICPP '06. Washington, DC, USA: IEEE Computer Society, 2006, pp. 523–530.
[25] V. Agarwal, F. Petrini, D. Pasetto, and D. A. Bader, "Scalable graph exploration on multicore processors," in *Proceedings of the 2010 ACM/IEEE International Conference for High Performance Computing, Networking, Storage and Analysis*, ser. SC'10. Washington, DC, USA: IEEE Computer Society, 2010, pp. 1–11.
[26] A. Yoo, E. Chow, K. Henderson, W. McLendon, B. Hendrickson, and U. Catalyurek, "A scalable distributed parallel breadth-first search algorithm on bluegene/l," in *Proceedings of the 2005 ACM/IEEE Conference on Supercomputing*, ser. SC '05. Washington, DC, USA: IEEE Computer Society, 2005, pp. 25–.
[27] P. Harish and P. J. Narayanan, "Accelerating large graph algorithms on the GPU using CUDA," in *Proceedings of the 14th International Conference on High Performance Computing*, ser. HiPC'07. Berlin, Heidelberg: Springer-Verlag, 2007, pp. 197–208.
[28] G. J. Katz and J. T. Kider, Jr, "All-pairs shortest-paths for large graphs on the GPU," in *Proceedings of the 23rd ACM SIGGRAPH/EUROGRAPHICS Symposium on Graphics hardware*, ser. GH '08. Aire-la-Ville, Switzerland, Switzerland: Eurographics Association, 2008, pp. 47–55.
[29] L. Luo, M. Wong, and W.-m. Hwu, "An effective GPU implementation of breadth-first search," in *Proceedings of the 47th Design Automation Conference*, ser. DAC '10. New York, NY, USA: ACM, 2010, pp. 52–55.
[30] S. Hong, S. K. Kim, T. Oguntebi, and K. Olukotun, "Accelerating CUDA graph algorithms at maximum warp," in *PPOPP*, 2011, pp. 267–276.
[31] "AMD Opteron 6100 series processor," available at http://www.amd.com/us/products/embedded/processors/opteron/Pages/opteron-6100-series.aspx.
[32] 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.
[33] "Tesla C1060 computing processor board," available at http://www.nvidia.com/docs/IO/43395/BD-04111-001_v06.pdf.
[34] "Tesla C2050 and Tesla c2070 computing processor board," available at http://www.nvidia.com/docs/IO/43395/Tesla_C2050_Board_Specification.pdf.
[35] 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.
[36] D. Chakrabarti and C. Faloutsos, "Graph mining: Laws, generators, and algorithms," *ACM Comput. Surv.*, vol. 38, no. 1, p. 2, 2006.
[37] D. A. Bader and K. Madduri, "SNAP, small-world network analysis and partitioning: an open-source parallel graph framework for the exploration of large-scale networks," in *Proceedings of the 2008 International Parallel and Distributed Processing Symposium (IPDPS 2008)*, Miami, FL, USA, 2008.

Mahantesh Halappanavar is a Research Scientist with the Computational Sciences and Mathematics Division at the Pacific Northwest National Laboratory. He received his MS and PhD degrees in Computer Science from the Old Dominion University in 2003 and 2009 respectively. His work focuses on parallel graph algorithms and spans several application areas including analysis of electric power grids, statistical textual analysis, numerical linear algebra, information security and machine learning. He explores the interplay of algorithm design, architectural features, and input characteristics targeting massively multithreaded architectures such as the Cray XMT and emerging multicore and manycore platforms.

John Feo is the Director of the Center for Adaptive Supercomputer Software at the Pacific Northwest Laboratory. Dr. Feo received his PhD in Computer Science from The University of Texas at Austin. He began his career at Lawrence Livermore National Laboratory where he managed the Computer Science Group and was the principal investigator of the Sisal Language Project. Dr. Feo then joined Tera Computer Company (now Cray Inc) where he was a principal engineer and product manager for the MTA-1 and MTA-2, the first two generations of the Cray's multithreaded architecture. After a short two year "sabbatical" at Microsoft where he led a software group developing a next-generation virtual reality platform, he joined PNNL. Dr. Feos research interests are parallel programming, graph algorithms, multithreaded architectures, functional languages, and performance studies. He has held academic positions at UC Davis and is an adjunct faculty at Washington State University.

Oreste Villa is a Research Scientist at the Pacific Northwest National Laboratory (PNNL) with a research focus on computer architectures and simulation, accelerators for scientific computing and irregular applications. He joined PNNL in May 2008 after receiving his PhD from Politecnico di Milano for his research on "Designing and Programming Advanced Multicore Architectures". While earning his PhD, he was an intern student at PNNL, conducting research in programming techniques and algorithms for advanced multicore architectures, cluster fault tolerance and virtualization techniques for HPC. Dr. Villa received a M.S. degree in Electronic Engineering in 2003 from the University of Cagliari in Italy and an M.E. degree in 2004 in Embedded Systems Design from the University of Lugano in Switzerland.

Antonino Tumeo received the M.S. degree in Informatic Engineering, in 2005, and the PhD degree in Computer Engineering, in 2009, from Politecnico di Milano in Italy. Since February 2011, he has been a Research Scientist at Pacific Northwest National Laboratory (PNNL). He joined PNNL in 2009 as a post doctoral research associate. Previously, he was a post doctoral

researcher at Politecnico di Milano. His research interests are modeling and simulation of high performance architectures, hardware-software codesign, FPGA prototyping and GPGPU computing.

Alex Pothen received his PhD from Cornell University and is a Professor of Computer Science at Purdue University. His research interests include combinatorial scientific computing, high performance computing and bioinformatics. He served as the Director of the CSCAPES Institute which developed combinatorial algorithms for enabling computational science on Petascale computers, with funding from the Office of Science of the U.S. Department of Energy from 2006-2012. He serves on the editorial boards of Journal of the ACM, and SIAM Review, the flagship journals of the Association for Computing Machinery and the Society for Industrial and Applied Mathematics.