Better Speedups Using Simpler Parallel Programming for Graph Connectivity and Biconnectivity

James A. Edwards
University of Maryland
College Park, Maryland
jedward5@umd.edu

Uzi Vishkin
University of Maryland
College Park, Maryland
vishkin@umd.edu

ABSTRACT

Speedups demonstrated for finding the biconnected components of a graph: 9x to 33x on the Explicit Multi-Threading (XMT) many-core computing platform relative to the best serial algorithm using a relatively modest silicon budget. Further evidence suggests that speedups of 21x to 48x are possible. For graph connectivity, we demonstrate that XMT outperforms two recent NVIDIA GPUs of similar or greater silicon area. Previous studies of parallel biconnectivity algorithms achieved at most a 4x speedup, but we could not find biconnectivity code for GPUs to compare biconnectivity against them.

Ease-of-programming: The paper suggests that parallel programming for the XMT platform is considerably simpler than for the SMP and GPU ones. Unlike the quantitative speedup results, the ease-of-programming comparison is more qualitative. Productivity of parallel programming is a central interest of PMAM/PPoPP strongly favoring ease-of-programming. We believe that the discussion is on par with the state of the art on this relatively underexplored interest.

The results provide new insights into the synergy between algorithms, the practice of parallel programming and architecture: (1) no single biconnectivity algorithm is dominant for all inputs; (2) XMT provides good performance for each algorithm and better speedups relative to other platforms; (3) the textbook (TV) PRAM algorithm was the only one that provided strong speedups on XMT across all inputs considered; and (4) the TV implementation was a direct implementation of a PRAM algorithm, though a nontrivial effort was needed to get a PRAM version with lower constant factors. Overall, it appears that previous low speedups on other platforms were not caused by inefficient algorithms or their programming. Instead, it is because of the *better match* between the algorithms and the XMT platform. Given the growing interest in adding architectural support for parallel programming to existing multi-cores, our results suggest the following open question: can such added architectural support catch up on speedups and ease-of-programming with a design originally inspired by parallel algorithms, such as XMT? Finally, this work addresses another related interest of PMAM/PPoPP: new parallel workloads that improve synergy with emerging architectures. One variant of biconnectivity algorithms demonstrated the potential advantage of enhancing XMT by supporting in hardware more thread contexts, perhaps through context switching between them–apparently, a first demonstration of this old Cray MTA concept benefiting XMT.

Categories and Subject Descriptors

D.1.3 [**Programming Techniques**]: Concurrent Programming—Parallel programming; C.1.4 [**Processor Architectures**]: Parallel Architectures

General Terms

Algorithms, Experimentation, Performance, Theory, Verification

Keywords

biconnectivity, connectivity, GPU, graph algorithms, many-core

1. INTRODUCTION

Given an undirected graph G, two vertices u and v in G are in the same *connected component* of G if there is a path connecting them, and the *graph connectivity problem* is finding all connected components of an input graph G. The *diameter* of a connected graph is the length of the longest path in the set of all shortest paths between every pair of vertices in the graph.

Biconnectivity is a property of undirected graphs; an undirected graph G is called *biconnected* if and only if it is connected and remains so after removing any vertex and all edges incident on that vertex. A graph S is an *induced subgraph* of G if it comprises a subset of the vertices of G and all the edges of G connecting two vertices in S. A *biconnected component* of G is an induced subgraph of G that is biconnected whose vertex set cannot be expanded while maintaining the biconnectivity of its induced subgraph. A vertex whose removal increases the number of connected components in the graph is called an *articulation point*, and an edge whose removal increases the number of connected components is called a *bridge*. In this paper, the *biconnectivity problem* is understood as the problem of determining the biconnected components, articulation points, and bridges of an undirected graph, and a *biconnectivity algorithm* is an algorithm that solves the biconnectivity problem.

Connectivity is one of the most elementary graph problems. However, for brevity we pay more attention to biconnectivity, the more advanced problem considered in this work. Biconnectivity is an interesting problem to study for two reasons. First, the biconnected components of a graph can reveal useful information about the graph. For instance, if the graph represents a computer network, then a biconnected component of the graph is a subset of the network that will remain connected even if one computer fails, and articulation points (or bridges) are computers (or connections between computers) whose failure will disconnect the network. Second, biconnectivity algorithms are relatively complex: they are among the most advanced algorithms given in parallel algorithms textbooks and nearly the most advanced in serial algorithms textbooks, and biconnectivity or simpler problems were the basis for papers on other parallel computing platforms. Complex algorithms for natural problems may be better predictors of system behavior than the often used small kernels.

In serial computing, depth-first search is regarded as the best biconnectivity algorithm. However, power constraints impose a limit on the maximum performance of serial processors, and parallel processors are becoming the only way to improve performance. Therefore, it is desirable to find an efficient parallel biconnectivity algorithm. When it comes to programming parallel algorithms it is often the case, more so than with serial algorithms, that there is no single algorithm that performs best in all cases (for example, see [14]). Instead, the best algorithm to use could be sensitive to the computing platform and the properties of the input data. In the PRAM theory of parallel algorithms, the two main performance parameters of an algorithm (assuming synchronous execution and availability of as many processors as needed at each step of the algorithms) are: (i) work – the total number of operations performed by an algorithm, and (ii) depth – its number of steps. In the case of graph algorithms, the performance of a given algorithm may depend not only on the size of the input graph, but other properties of the input as well, such as the ratio of edges to vertices or the diameter of the graph.

Given a platform, this suggests viewing all non-dominated biconnectivity algorithms as a "collage" composed of "patches", where each patch represents a particular biconnectivity algorithm and the whole collage is a complete solution to the biconnectivity problem.

To demonstrate this approach, we evaluate three biconnectivity algorithms on the Explicit Multi-Threaded (XMT)[1] architecture developed at the University of Maryland. Because XMT is an experimental platform, we validate it by comparing it to a better established platform that uses similar silicon area, the NVIDIA GPU. We compare XMT to the GTX 280 (based on the older Tesla architecture) and the GTX 480 (based on the newer Fermi architecture) on significant portions of the biconnectivity algorithms for which optimized CUDA code has already been written by other programmers.

A 1024-core version of XMT, which would use a silicon area between that of one and two quad-core Intel Core i7 920 processors, demonstrated cycle count speedups of 9x to 33x on biconnectivity relative to a serial biconnectivity algorithm running on the Core i7 920, and further evidence suggests that speedups of 21x to 48x are possible when the investment in the design of the parallel processors matches that of the serial processor. The quantitative contributions of this paper include

- stronger speedups than in prior parallel biconnectivity studies (9x to 33x vs. $\leq$4x) across a varied family of graphs and
- stronger speedups on parallel connectivity than GPUs of similar or greater area (between 2x and 4.9x faster than the GTX 480).

Since Cong and Bader [12] appears to provide the most relevant prior work, we discuss the significance of the contributions by relating it to their discussion of the challenges they faced with adopting the Tarjan-Vishkin parallel biconnectivity algorithm to a 12-processor SMP. Cong and Bader noted that: (i) the TV algorithm is representative of many parallel algorithms that take drastically different approaches than the sequential algorithm to solve certain problems, and it employs basic parallel primitives such as prefix sum, pointer jumping, list ranking, sorting, connected components, spanning tree, Euler-tour construction and tree computations, as building blocks; (ii) while prior studies demonstrated reasonable parallel speedups for these parallel primitives on SMPs, they left unclear whether an implementation using these techniques achieves good speedup compared with the best sequential implementation because of the cost of parallel overheads encountered (i.e., of resorting to using all these primitives in the first place instead of doing DFS with a stack, per Hopcroft and Tarjan's original serial algorithm); (iii) looking at the whole algorithm rather than at individual primitives allows focusing on algorithmic overhead instead of communication and synchronization overhead; considering one primitive at a time tends to focus on input representations that do not necessarily fit together when used by a single algorithm; converting representations is not trivial, and incurs a real cost in implementations; and (iv) direct implementation of TV on SMPs fell behind the sequential implementation even at 12 processors. Their conclusion was to follow the major steps of TV, but use different approaches for several of the steps, guided by the challenge of reducing the overheads of TV in order to get ahead of the sequential implementation on the 12-processor SMP.

Our goal is different. While reducing overheads remains important, we try to stay much closer to the original PRAM description of TV taking advantage of the scalable XMT platform that was engineered to accommodate that. It is remarkable that XMT manages to get the strong speedups reported with such a relatively modest silicon budget. Also, our implementation demonstrates for the first time the potential advantage of enhancing XMT by supporting in hardware more thread contexts, perhaps through context switching between them. Namely, the significance of the contributions is

- new evidence supporting the practicality of algorithms derived from parallel random-access machine (PRAM) algorithmic theory for speedups and ease-of-programming,
- new evidence demonstrating the advantages of the XMT architecture for the same, and

[1]Not to be confused with the Cray XMT

- the demonstration of a synergistic approach to the design of algorithms and architectures.

The results presented herein are specific to graph connectivity and biconnectivity. Other papers [10, 9] show similar or better speedups for other graph and non-graph problems on XMT. Admittedly, these results do not (and cannot) establish the advantage of XMT for all possible tasks for which one might want to use a general-purpose computer. However, the importance of this work goes a bit beyond just providing one more point of reference. In a similar way that performance, efficiency and effectiveness of a car should not be tested only in first gear, productivity horizons of programming parallel algorithms on a given platform cannot only be studied using elementary algorithms. Graph connectivity problems provide a test case for a proverbial low gear with the more basic graph connectivity algorithms, and higher gear with more advanced graph algorithms for biconnectivity. This and other papers will enable more informed judgment on the overall relative productivity of various approaches. Such documented comparisons will reduce the risk to vendors, allowing them to make better decisions regarding platforms they may want to build.

1.1 Related Work

Although no studies of biconnectivity algorithms have previously been published for many-core processors, [12] examines such algorithms on a symmetric multiprocessor (SMP). Also, list ranking and connected components algorithms, two major components of the Tarjan-Vishkin biconnectivity algorithm, are examined in [5] on an SMP and on the Cray MTA.

Another parallel framework that bears limited resemblance to the many-core platforms evaluated here is MapReduce, which uses large clusters of computers to take advantage of massive parallelism in very large problems. This approach was used for estimating the diameter of large graphs in [18], and the potential to adapt PRAM algorithms into computationally feasible MapReduce algorithms was discussed in [19]. However, the applicability of MapReduce to high-end many-core platforms is not clear and the algorithms examined in this paper are not necessarily optimal for use in distributed systems such as MapReduce.

2. EVALUATED ALGORITHMS

Given a graph with n vertices and m edges, the biconnectivity problem can be efficiently solved on a serial computer in $O(n+m)$ time with an algorithm by Hopcroft and Tarjan [17] that performs a depth-first search (DFS) on the graph. This algorithm does not appear to have an efficient, poly-logarithmic-time implementation [26]. It is possible to extract some parallelism from this algorithm using the approach outlined in Exercise 36 in [32], and the resulting algorithm, which we will refer to as parallel DFS (pDFS), runs in $O(n)$ time using $\lceil m/n \rceil + 1$ processors. The main weakness of this algorithm is that the amount of parallelism available depends on the m/n, the "density" of the graph: vertices are processed in serial, and the parallelism available at a vertex is limited by its degree. This algorithm provides little to no parallelism for sparse graphs, where m/n is small.

A more scalable alternative is a biconnectivity algorithm given by Tarjan and Vishkin in [31] that runs in $O(\log n)$ time using $O(n+m)$ processors. The theoretical running time of this algorithm depends only on the size of the graph, not on its structure. This scalability comes at a cost, however: the Tarjan-Vishkin (TV) algorithm performs more operations per vertex and per edge than are required by the serial algorithm or pDFS. Thus, TV may be outperformed by other algorithms in certain situations despite being asymptotically more efficient, especially when running on computer hardware supporting a modest amount of parallelism (e.g. a 4- or 8-core processor).

In these situations, it may be worth modifying TV to be more work efficient. TV is a modular algorithm that calls upon parallel algorithms for simpler problems to do its work. The most significant of these in terms of running time is an algorithm to compute the connected components of an undirected graph (connectivity algorithm). To obtain the complexity bounds in [31], Tarjan and Vishkin used a variation of the Shiloach-Vishkin (SV) connectivity algorithm [27], which runs in $O(\log n)$ time using $O(n+m)$ processors. This algorithm is efficient in asymptotic terms, but its running time has a large constant factor due to the need to revisit vertices and edges multiple times throughout the algorithm. In some cases, it may be beneficial to use another connectivity algorithm, such as breadth-first search (BFS), in place of SV.

In this paper, we evaluate three biconnectivity algorithms, which we describe below: parallel depth-first search and two versions of the Tarjan-Vishkin algorithm, one using the SV connectivity algorithm and another using BFS in addition to SV.

2.1 Input and Output

The input to a biconnectivity algorithm is an undirected graph $G=(V,E)$ with $n=|V|$ vertices and $m=|E|$ edges. Without loss of generality, we assume that G is connected; if not, the biconnectivity problem can be solved for G by applying a biconnectivity algorithm to each connected component of G. To allow using directly the three biconnectivity algorithms, the input graph is given in the following format:

- Each undirected edge (u,v) in E is represented as a pair of antiparallel directed edges, $u \rightarrow v$ and $v \rightarrow u$. These $2m$ directed edges are stored in an array $edges[2m]$ sorted by the first endpoint.

- For each directed edge $edges_i = u \rightarrow v$ in $edges$, an array $antiparallel[2m]$ stores the index j of its antiparallel copy $edges_j = v \rightarrow u$ such that $antiparallel_i = j$ and $antiparallel_j = i$.

- An array $vertices[n]$ stores indices into the $edges$ array such that, if $vertices[u] = i$, then $edges_i$ is the first edge in $edges$ whose first endpoint is u.

- An array $degrees[n]$, where $degrees_v$ is the degree of vertex v.

Given the data listed above, the algorithm is expected to produce the following output:

- An array $bcc[2m]$ that identifies the biconnected component to which each edge belongs such that for any pair of edges $edges_i$ and $edges_j$, $bcc_i = bcc_j$ if and only if $edges_i$ and $edges_j$ are in the same biconnected component.

- An array $artic\_points[a], 0 \le a \le n$ of all the articulation points in G.
- An array $bridges[b], 0 \le b \le 2m$ of the indices in $edges$ of all the bridges in G.

2.2 Parallel Depth-First Search (pDFS)

The intuition behind pDFS is that, although vertices cannot be visited in parallel without potentially violating the order required by a depth-first traversal, edges can be. Initially, all edges are considered *active*. Whenever a vertex is visited in the DFS traversal, all edges leading to that vertex are *canceled*, or removed from the set of active edges. Only active edges are considered when checking for adjacent vertices. Given an input graph in the format described in section 2.1, a parallel version of the standard DFS algorithm proceeds as follows:

1. For each vertex v, create a doubly-linked list of its incident edges.
   - Using one thread per vertex, create an array $head[n]$ such that $head_v$ is the index of the first active edge in $edges$ originating from v, or -1 if no such edge exists. Initially, all edges are active, so $head_v \Leftarrow vertices_v$ if $degrees_v > 0$ and $head_v \Leftarrow -1$ otherwise.
   - Using one thread per edge, create the arrays $next[2m]$ and $prev[2m]$ such that $next_i$ and $prev_i$ are the indices in $edges$ of the next and previous active edges, respectively, that originate from the same vertex as $edges_i$, or -1 if no such edge exists. Initially, $next_i \Leftarrow i + 1$ and $prev_i \Leftarrow i - 1$ with the following exceptions: $prev_i \Leftarrow -1$ if $edges_i$ is the first edge in $edges$ that shares its origin and $next_i \Leftarrow -1$ if it is the last such edge.

   This list contains all of the active edges originating from v.
2. Define the procedure dfs(v) as follows:
   (a) In parallel, for every edge $edges_i$ originating from v, remove $edges_j = w \to v$, where $j = antiparallel_i$, from the doubly-linked list in which it is contained:
      - if $prev_j \ne -1$ then $next[prev_j] \Leftarrow next_j$ else $head_w \Leftarrow next_j$
      - if $next_j \ne -1$ then $prev[next_j] \Leftarrow prev_j$

   (b) While $head_v \ne -1$, invoke dfs(w), where $v \to w = edges[head_v]$.
3. Invoke dfs(r) for some arbitrary vertex r

In order to use DFS to solve the biconnectivity problem, we need two pieces of information about each visited vertex v: its preorder number, pre_v; and the smallest preorder number seen while performing DFS on v and its descendants, low_v. In serial DFS, pre_v can be computed by keeping track of the number of vertices visited so far in a global variable $count$. Every time a new vertex v is visited, pre_v is set to $count$, and then $count$ is incremented by 1. The value of low_v is determined by initializing low_v to pre_v upon entering v and updating low_v after (re)visiting a child w as follows: $low_v \Leftarrow min(low_v, low_w)$.

In pDFS, pre_v can be computed the same way because the vertices are still visited serially. However, low_v cannot be because, unlike in serial DFS, visited vertices are never revisited since all edges leading to a visited vertex are always canceled. The key observation that allows us to compute low_v in parallel is the following: the final value of low_v is not needed until returning from the visit to v. Therefore, low_v can be computed just before returning from v as follows: $low_v \Leftarrow min(pre_v, min_{w \in \mathrm{children}(v)}(low_w))$. The remainder of the pDFS algorithm is identical to its serial counterpart.

2.3 Tarjan-Vishkin (TV)

The Tarjan-Vishkin biconnectivity algorithm [31] is a PRAM algorithm that was designed as a scalable alternative to DFS. It uses the same principle as the DFS biconnectivity algorithm: two edges in a graph are in the same biconnected component if and only if they are on a common simple cycle. However, TV can use any spanning tree, and it performs an Euler tour of the spanning tree to compute information equivalent to that computed in the DFS biconnectivity algorithm. (An Euler tour of a graph is a cycle that visits every vertex in the graph and visits every edge exactly once.) Given an input graph G, TV proceeds as follows:

1. Use a parallel connectivity algorithm to find a spanning tree T of G.
2. Compute an Euler tour of T', where T' is formed by replacing every undirected edge in T with a pair of antiparallel directed edges. This results in a linked list L of edges in T'.
3. Perform list ranking [14] on L to determine the distance of each edge from the end of the Euler tour. Use these distances to determine for each vertex v in T (1) the preorder pre_v of v in T and (2) the size $size_v$ of the subtree of T rooted at v.
4. For each vertex v, compute low_v and $high_v$. These are the lowest and highest preorder numbers, respectively, of the vertices in the set consisting of v, the descendants of v, and all vertices that are adjacent to v or one of its descendants by an edge in $G - T$.
5. Construct an auxiliary graph G', where the vertex set of G' equals the edge set of T and the edge set of G' is constructed as follows, where $p(v)$ denotes the parent of v in T and $v \to w$ denotes an edge in T such that $v = p(w)$:
   - for each edge $\{v, w\}$ in $G - T$, add $\{\{p(v), v\}, \{p(w), w\}\}$ to G' if and only if v and w are unrelated in T and
   - for each edge $v \to w$ in T, add $\{\{p(v), v\}, \{v, w\}\}$ if and only if $low_w < v$ or $high_w \ge v + size_v$.
6. Compute the connected components of G'. This defines an equivalence relation on the edges of T such that a pair of edges in T are in the same connected component of G' if and only if they are in the same biconnected component of G.
7. Extend the equivalence relation on the edges of T to the edges of $G - T$ by defining $\{v, w\}$ equivalent to $\{p(w), w\}$ for each edge $\{v, w\}$ of $G - T$ such that $pre_v < pre_w$.

8. Identify the bridges in G, which are the edges $v \to w$ of T such that low_w and $high_w$ are both descendants of w.

9. Identify the articulation points in G, which are the vertices of G that exist in more than one biconnected component of G.

In steps (1) and (6), any connectivity algorithm may be used without affecting the correctness of the overall biconnectivity algorithm. The version of this algorithm originally described by Tarjan and Vishkin uses the SV connectivity algorithm; we refer to this version simply as the Tarjan-Vishkin (TV) biconnectivity algorithm.

Our implementation of TV on XMT merits some discussion since it is path-breaking effort towards dual validation of the XMT platform and PRAM algorithmics. Originally inspired by PRAM algorithmics and its complexity analysis, the long-term objective of the XMT platform was to revisit the more advanced PRAM algorithms and show that their merit transcends theory. Each PRAM algorithm whose implementation beats the competition for the respective problem it addresses would constitute partial accomplishment of this objective. We are not aware of any prior implementation of a biconnectivity algorithm on XMT or any similar platform. Only the concomitant work [9] represents implementation of an algorithm of similar complexity on XMT.

Implementation.

The high-level description given in the original paper [31] focuses on achieving complexity results, requiring us to find an implementation that provides good performance. In contrast to [12], we leave the core algorithm as is without reducing its available parallelism, but we choose an implementation that minimizes the amount of work done by the algorithm. In steps (1) and (6), we compact the adjacency list every few iterations as more vertices are discovered to be in the same connected component. In step (3), we accelerate the iterations by choosing faster but more work demanding list ranking algorithms for different iterations ("accelerating cascades", [11]). Also, to save work we transition as many computations as possible from the original input graph to the spanning tree.

The following insights were observed in programming the TV PRAM algorithm. They attest that the practical challenge of effectively programming this theoretical parallel algorithm has a similar flavor to the practice of programming serial algorithms and are much simpler than parallel programming approaches such as [13] with their requirements for decomposition, assignment, orchestration and mapping.

1. Although the same connectivity algorithm is used in steps (1) and (6), it is worthwhile to code two variants of it: one that saves the spanning tree computed by the connectivity algorithm and one that does not. These two versions take different approaches to handling the arbitrary concurrent writes that result when multiple vertices try to hook on the same vertex. The version that saves the spanning tree needs to know which of the writes succeeded in order to know which edge should be added to the spanning tree. On XMT, this is accomplished by performing a prefix sum to memory on a gatekeeper array. On the other hand, if the spanning tree is not needed, then it is not necessary to know which processor succeeded, and this extra work can be avoided, as the connectivity algorithm is in the common CRCW model.

2. The best data structure for storing the spanning tree is the same one as used for the input graph. This can be derived from the output of step (1) in the following way. Step (1) produces an array T with one entry per edge in the input graph where entry i is 1 if edge i is in the spanning tree and 0 if it is not. The edge list for the spanning tree should be produced by the standard order-preserving PRAM compaction algorithm. The remaining arrays (vertices, degrees, and antiparallel) can then be trivially derived from the corresponding arrays in the input graph. If we use instead a platform-specific optimization (such as prefix sum to registers on XMT) to create the edge list, then we will not be able to derive the necessary tree data structure from the input graph, and it will be difficult to implement the rest of the biconnectivity algorithm (especially the Euler tour) efficiently.

3. Depending on the platform, it may be worthwhile to explicitly relabel the vertices in the graph after rooting the spanning tree by creating a new edge array where the entry corresponding to the edge (u, v) contains the entry $(preorder(u), preorder(v))$. This is an expensive operation up front, but it can save more time in later steps of the algorithm when compared to the alternative of accessing the preorder array each time a relabeled vertex number is needed.

4. When computing global low and high numbers for each vertex, it is necessary to find the minima/maxima of some subarrays of preorder numbers. The PRAM algorithms for doing this first find prefix minima/maxima and suffix minima/maxima relative to subarrays that occur naturally as a result of using a balanced binary tree over an array representing an Euler tour. It has been observed in [33] that a balanced k-ary tree will be more efficient in practice than a balanced binary tree with the exact k depending on the specific machine at hand. Replacing a binary tree by such a k-ary tree generates different subarrays. This implies finding prefix minima/maxima and suffix minima/maxima relative to these subarrays, and to retrieving low and high numbers from them.

2.4 Tarjan-Vishkin with a BFS Spanning Tree (TV-BFS)

For some inputs, better performance can be obtained using a connectivity algorithm with worse asymptotic time bounds but a lower constant factor on work, such as breadth-first search (BFS). BFS naturally lends itself to a parallel implementation, and such an implementation runs in $O(h \log n)$ time and $O(n + m)$ work, where h is the number of layers in the BFS traversal of the graph [15]. The value of h depends on the size and shape of the graph as well as the starting vertex for the traversal, and it can be as large as the diameter of the graph. Notably, for graphs with a diameter that is $O(\log n)$, BFS runs in poly-logarithmic time and thus is an asymptotically efficient parallel algorithm. Even on graphs with somewhat larger diameters, BFS can run more quickly than SV due to its lower constant factor, but for graphs with a large diameter relative to the number of vertices (long, thin graphs), there is too little parallelism available for BFS to be efficient.

In theory, BFS can be used in place of SV for computing both the spanning tree of the original graph and the connected components of the auxiliary graph. However, the most natural representation for the auxiliary graph generated by TV is a list of edges in arbitrary order. This representation is not suitable as input to BFS, which requires the graph to be represented as an adjacency list. Therefore, BFS cannot be used to find the connected components of the auxiliary graph as is. It is possible to convert the edge list produced by TV to an adjacency list, but doing so requires sorting the edge list, which reduces or eliminates the benefit of using BFS in place of SV, so we do not consider it further. If the input to the biconnectivity algorithm is in the proper format, BFS can be used in place of SV to find the spanning tree of the input graph, and we call this variation TV-BFS.

3. EVALUATED PLATFORMS

We briefly review relevant specifics of the computing platforms on which our experiments are performed. A more detailed overview can be found in [10]. Specifications of the specific configurations evaluated can be found in Table 1.

	GTX 280	GTX 480	XMT-1024	XMT-2048
Principal Computational Resources				
Cores	240 SP	480 SP	1024 TCU	2048 TCU
Integer Units	240 ALU +MDU	480 ALU +MDU	1024 ALU, 64 MDU	1024 ALU, 64 MDU
(Floating Point Units)[a]	240 FPU, 60 SFU	480 FPU, 60 SFU	64 FPU	64 FPU
On-chip Memory				
Registers	1920KB	1920KB	128KB	256KB
Prefetch Buffers	-	-	32KB	64KB
Regular caches	480KB	1728KB[b]	4104KB	4104KB
Constant cache	240KB	120KB	128KB	128KB
Texture cache	496KB	120KB	-	-

[a]None of the algorithms in this paper use the floating-point units.

[b]64KB configurable shared memory/L1 cache per SM and 768KB unified L2 cache

Table 1: Specifications of the platforms evaluated in the experiments (1 KB = 1024 bytes, SP = Streaming Processor, TCU = Thread Control Unit, ALU = Arithmetic/Logic Unit, MDU = Multiply/Divide Unit, SFU = Special Function Unit)

3.1 GPUs

Though not originally designed for general-purpose computing, modern graphics processing units (GPUs) are capable of being used as highly parallel computing platforms; this usage of GPUs is referred to as general-purpose GPU (GPGPU). Examples of prevalent GPGPU architectures include Tesla and Fermi, both by NVIDIA. GPUs based on the Tesla architecture are widely used, and there are many parallel applications available to run on them. GPUs based on the Fermi architecture are newer, and there are fewer applications optimized specifically for them, though they are backward compatible with applications written for the Tesla architecture.

The Tesla architecture consists of a number of Streaming Multiprocessors (SMs) connected to a number of DRAM controllers and off-chip memory through an interconnection network. An SM consists of a shared register file, shared memory, constant and instruction caches, special function units (SFUs), and a number of streaming processors (SPs) with integer and floating point ALU pipelines. SFUs are 4-wide vector units that can handle complex floating-point operations.

With respect to biconnectivity algorithms, which do not use floating-point operations, the main advantage of the Fermi architecture over Tesla is the addition of L1 and L2 caches. In Fermi, each SM has 64 KB of memory, which can be split into shared memory and L1 cache in one of two ways: 48 KB shared memory and 16 KB L1 cache or 16 KB shared memory and 48 KB L1 cache [2]. There is also a 768 KB L2 cache shared by all the SMs.

For more information about Tesla, see [22], and for Fermi, see [24].

3.2 XMT

The Explicit Multi-Threading (XMT) general-purpose computer architecture is designed to improve single-task completion time. It does so by supporting programs based on Parallel Random-Access Machine (PRAM) algorithms but relaxing the synchrony required by the PRAM model. The XMT programming model differs from the strict PRAM model in two ways:

1. The PRAM model requires specifying the instruction that will be executed by each processor at each point in time, but XMT uses the work-depth methodology [28], which allows the programmer to specify all of the operations that can be performed at each point in time while leaving to the runtime environment the assignment of those operations to processors.

2. The PRAM model requires instructions to be executed in lockstep by all processors at once, but XMT programs follow independence-of-order semantics: parallel sections of code are delimited by spawn-join instruction pairs, and threads only synchronize when they reach the join instruction at the end of the parallel section.

The XMT architecture consists of the following: a number of lightweight cores (TCUs) grouped into clusters, a single core (master TCU or MTCU) with its own local cache, a number of mutually-exclusive cache modules shared by the TCUs and MTCU, an interconnection network connecting the TCUs to the cache modules, and a number of DRAM controllers connecting the cache modules to off-chip memory. Each TCU has a register file, a program counter, an execution pipeline, and a lightweight ALU. Each TCU also contains prefetch buffers, which can be used by the compiler to prefetch data from memory before it is needed, reducing the length of the sequence of round trips to memory (LSRTM) and improving performance [33]. Each cluster has one or more multiply/divide units (MDUs), floating-point units (FPUs), and a compiler-managed read-only cache, all of which are shared by the TCUs within the cluster. When a parallel section of code is reached, the MTCU broadcasts the instructions in that section to all of the TCUs, and each TCU stores the instructions in a buffer. Virtual threads are assigned to TCUs using a dedicated prefix-sum network.

As noted, a more detailed overview of XMT and the GTX 280 can be found in [10].

3.3 Evaluated configurations

The Tesla and Fermi architectures are used in commercially-available products. Therefore, we do not need to establish the practicality of their implementation. We choose the GTX 280 GPU, based on the Tesla architecture, and the GTX 480, based on the Fermi architecture, to represent their respective architectures.

Because XMT is an experimental platform, we establish that XMT is competitive with single-chip multi-cores and many-cores currently available on the market by choosing a configuration of XMT that would use resources comparable to the GTX 280, the less resource-intensive of the two GPUs evaluated. The GTX 280 uses 576 mm^2 of silicon in 65 nm technology, and according to [10], a 1024-TCU configuration of XMT would use a comparable silicon area. The GTX 480 uses 529 mm^2 of silicon in 40 nm technology and contains more SPs and memory than the GTX 280. Therefore, it can be argued that a 1024-TCU configuration of XMT (XMT-1024) would use at most 529 mm^2 of silicon, and likely less, in 40 nm technology. The 45-nm Intel Core i7 920 quad-core processor, which uses 263 mm^2 of silicon, is half the area of the GTX 480. This places an upper bound on the area of XMT-1024; a lower bound of $576\text{mm}^2 \times \left(\frac{45\text{nm}}{65\text{nm}}\right)^2 = 276\text{mm}^2$ can be found by assuming ideal scaling from 65 nm to 45 nm. In summary, XMT-1024 would use

- about the same area as the GTX 280, while remaining in the same power envelope [20],
- less area than, or at worst the same area as, the GTX 480, and
- an area somewhere between that of one and two Core i7 920 quad-core processors.

To determine the sensitivity of the biconnectivity algorithms to the number of concurrent hardware threads, we also consider a configuration of XMT identical to XMT-1024 with the exception of having twice as many TCUs per cluster; we call this configuration XMT-2048. We do not attempt to argue here that the silicon area of XMT-2048 matches the aforementioned GPUs but merely use it as a reference point.

To collect cycle counts for programs executed on the XMT-1024 and XMT-2048 configurations, we used XMTSim, the cycle-accurate simulator of the XMT architecture. XMTSim and the XMTC compiler are described in [21] and have already been the basis for several publications including [10].

4. EXPERIMENTAL EVALUATION

4.1 Tested Graphs

Data set	Vertices	Edges	Average	Diameter	
			Degree	Min.	Max.
1kv-500ke-complete	1,000	499,500	999.00	1	1
20kv-5me-random	20,000	5,000,000	500.00	2	4
1mv-3me-planar	1,000,002	3,000,000	6.00	333,333	333,333
USA-road-d.LKS	2,758,119	3,397,404	2.46	3,240	6,480
web-Google-con	855,802	4,291,352	10.00	15	30

Table 2: Properties of the graphs used in the experiments. For graphs whose diameter is not known, lower and upper bounds are given based on the number of layers in a BFS traversal of the graph.

In our experiments, we use three synthetic graphs and two graphs derived from real-world data. Properties of these graphs are given in Table 2. The synthetic graphs are as follows:

- 1kv-500ke-complete: The complete graph of 1,000 vertices (and ∼500,000 edges)
- 20kv-5me-random: A graph with 20,000 vertices generated by adding 5 million unique edges between randomly selected pairs of vertices
- 1mv-3me-planar: A maximal planar graph with 1 million vertices generated layer by layer using the following rules:
  - The first layer is the complete graph of three vertices (and three edges). Call this graph G_1 and its three vertices the *external vertices* of G_1.
  - Given a graph G_i generated according to these rules with external vertices a, b, and c, generate a new graph G_{i+1} by adding vertices a', b', and c' and the following edges: (a', a), (a', b), (a', b'), (b', b), (b', c), (b', c'), (c', c), (c', a), (c', a'). Vertices a', b', and c' are the external vertices of G_{i+1}.

The real-world graphs are as follows:

- USA-road-d.LKS: A graph of the road network in the Great Lakes region, taken from [1].
- web-Google-con: The largest connected component of the Google web graph of web pages and hyperlinks between them, taken from [3]. This is actually a directed graph, but we convert it to an undirected graph by treating each edge in the original graph as an undirected edge.

Of the five graphs, the first two (the complete graph and the random graph) are of less interest in practical applications of biconnectivity because random graphs are very unlikely to have "interesting" articulation points or bridges (those that divide the graph into large blocks), and complete graphs have none at all. They are included only to show the behavior of the algorithms on dense graphs.

It is possible that larger graphs than the ones listed here may provide more parallelism. However, for the purposes of this paper, the evaluated graphs are sufficiently large; they provide enough parallelism for the SV connectivity algorithm and the TV biconnectivity algorithm, and the parallelism available to TV-BFS and pDFS depends on the shape of the input graph.

4.2 Results for comparing GPUs and XMT

To support a fair comparison of XMT with the GPUs, we compare against code optimized by others for GPUs. However, at the time of this writing, no such code exists to solve the biconnectivity problem on GPUs. Therefore, we could only test the most time-consuming algorithms used in the Tarjan-Vishkin biconnectivity algorithm, which are logarithmic-time connectivity and BFS.

The 1024-TCU configuration of XMT was already shown to perform better than the GTX 280 on BFS by a factor of

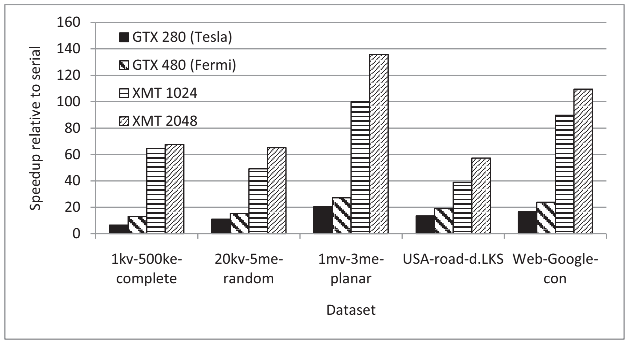


Figure 1: Speedups of the parallel SV connectivity algorithm on the evaluated platforms with respect to serial DFS running on the Core i7 920.

Dataset	GTX 280	GTX 480	XMT 1024	XMT 2048
1kv-500ke-complete	6.60	13.13	64.54	67.56
20kv-5me-random	10.98	15.41	49.09	65.06
1mv-3me-planar	20.45	27.11	99.85	135.79
USA-road-d.LKS	13.45	19.04	38.99	57.35
Web-Google-con	16.58	23.82	89.75	109.53

Table 3: Speedups of the parallel SV connectivity algorithm on the evaluated platforms with respect to serial DFS running on the Core i7 920.

5 in [10], so we will not consider it any further in this paper. Instead, we focus on logarithmic-time connectivity and compare our implementation of the Shiloach-Vishkin connectivity algorithm on XMT against code written by Soman et al. in [29], the only implementation of graph connectivity on GPUs we are aware of at the time of this writing. As shown in Figure 1 and Table 3, XMT with 1,024 TCUs outperforms the stronger among the GTX 280 and the GTX 480 by factors ranging between 2.2x and 4x on all input data sets considered.

Soman et al. [29, 30] report that irregular memory access algorithms such as the ones for finding connected components are not a good fit for the GPU computation model, which relies heavily on regularity of memory access; they review both the practical improvements they introduced to the SV algorithm in order to reduce its number of operations, as well as the non-trivial problems they had to overcome in order to fit the GPU model. While our work shares similar features with the former, the flexibility of the XMT architecture freed us from the latter concerns.

This is one of the main results of this paper. We also expect XMT to perform competitively in solving the biconnectivity problem. This result should not be generalized much further beyond this; in particular, we do not claim that XMT provides a similar performance advantage over GPUs on applications with regular memory access patterns, for which GPUs were designed.

4.3 Biconnectivity Algorithms: Overall Speedups and Comparison of Algorithms

Figure 2a and the left half of Table 4 show the speedups of the three parallel biconnectivity algorithms on XMT with respect to serial DFS on the Core i7 920. We used our implementation of Tarjan's serial DFS algorithm, similar to Cong and Bader, who used theirs. The 64-TCU results were obtained from the Paraleap FPGA [35], and the 1024-TCU and 2048-TCU results were obtained from the XMT simulator. The simulator produces inaccurate cycle counts for serial code because it does not simulate the local cache of the MTCU. The FPGA does have a local cache for the MTCU, so it provides more accurate cycle counts for serial code. The following steps were taken to compensate for this discrepancy:

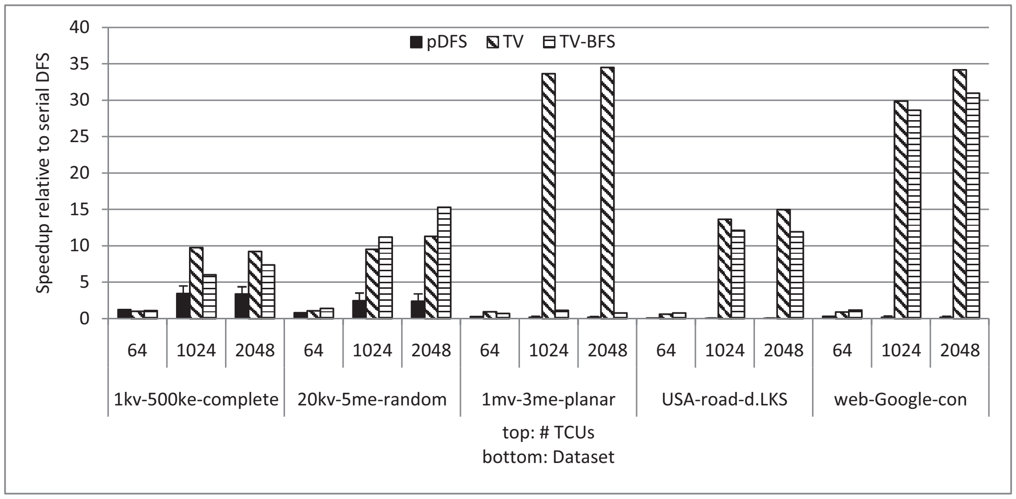
- Cycle counts for the serial versions of the algorithms, which are used as baseline values for the speedups of the parallel algorithms versus the XMT MTCU, were measured on the FPGA.
- For pDFS, which is the only parallel algorithm with a significant serial component evaluated in this paper, cycle counts for serial sections and parallel sections of execution were measured separately. For the 1024-TCU and 2048-TCU results, we added the serial cycle count from the FPGA to the parallel cycle count from the simulator to obtain a compensated cycle count. This compensated cycle count is lower than the true cycle count because it does not account for the additional delay of the larger interconnection network in the simulated configurations. Thus, it forms a lower bound on the true cycle count. The non-compensated cycle count is larger than the true cycle count and therefore forms an upper bound. We report speedups based on both sets of cycle counts.

We make the following observations about the results and their significance:

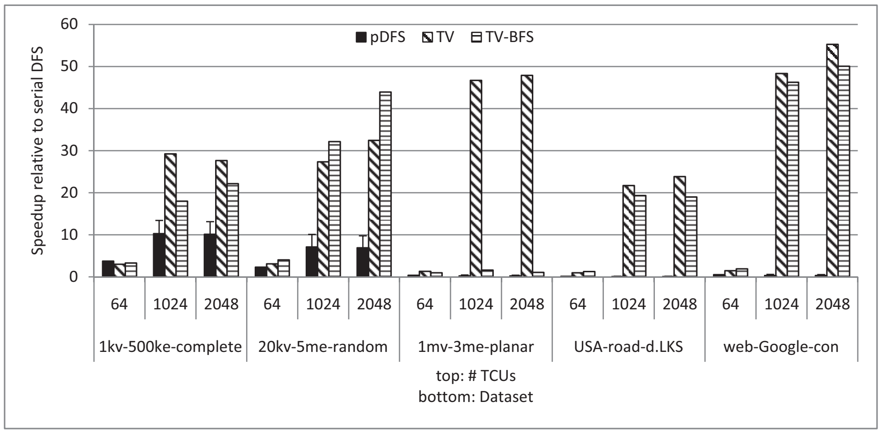
- The lack of significant speedups for the 64-TCU configuration is due in part to the parallel algorithms performing more work than the serial algorithm. What make achieving speedups relative to the serial Hopcroft-Tarjan biconnectivity algorithm particularly challenging is that it is very compact, requiring a single visit to each vertex and each edge, as opposed to several visits in the TV-based algorithms.
- The TV algorithm provides speedups of at least 9x relative to the Core i7 and 21x relative to the XMT MTCU on all inputs with 1,024 TCUs. This implies that TV is a good general-purpose parallel biconnectivity algorithm.
- For the 1mv-3me-planar graph, TV provides significantly higher speedups than the other algorithms considered. This is because this graph has a very large diameter and low degree per vertex, which means that there is too little parallelism for pDFS and TV-BFS to exploit. TV is the only algorithm that can provide adequate performance in this case. It is worth noting that this graph is a good representative of many real-world graphs for which one might want solve the biconnectivity problem, so the results for this graph are likely to show the performance of the algorithms in typical usage.
- On the 20kv-5me-random graph, TV-BFS provides the best performance because this graph has a very small diameter. This means that in situations where the

		Speedup vs. Core i7 920			Speedup vs. XMT MTCU		
Data set	TCUs	pDFS	TV	TV-BFS	pDFS	TV	TV-BFS
1kv-500ke-complete	64	1.25	1.01	1.10	3.73	3.02	3.30
	1024	3.45 (4.49)	9.77	6.02	10.31 (13.44)	29.23	18.02
	2048	3.38 (4.39)	9.25	7.40	10.13 (13.13)	27.67	22.15
20kv-5me-random	64	0.81	1.08	1.41	2.31	3.09	4.06
	1024	2.48 (3.53)	9.53	11.21	7.13 (10.13)	27.37	32.19
	2048	2.42 (3.40)	11.30	15.31	6.94 (9.76)	32.44	43.96
1mv-3me-planar	64	0.29	0.97	0.72	0.40	1.35	1.00
	1024	0.19 (0.32)	33.63	1.16	0.26 (0.45)	46.72	1.61
	2048	0.18 (0.30)	34.50	0.79	0.25 (0.42)	47.92	1.10
USA-road-d.LKS	64	0.09	0.63	0.79	0.14	1.00	1.26
	1024	0.05 (0.10)	13.66	12.14	0.09 (0.16)	21.74	19.32
	2048	0.05 (0.10)	14.98	11.95	0.08 (0.15)	23.85	19.01
web-Google-con	64	0.32	0.92	1.19	0.52	1.49	1.93
	1024	0.21 (0.38)	29.89	28.62	0.34 (0.61)	48.32	46.26
	2048	0.20 (0.35)	34.19	30.97	0.32 (0.57)	55.26	50.06

Table 4: Speedups of the evaluated biconnectivity algorithms on XMT relative to the serial DFS-based Hopcroft-Tarjan biconnectivity algorithm (values in parentheses for pDFS are based on compensated cycle counts). Key: pDFS = parallel DFS, TV = Tarjan-Vishkin, TV-BFS = Tarjan-Vishkin using BFS to find the spanning tree.



(a) Speedups vs. the Core i7 920



(b) Speedups vs. the XMT MTCU

Figure 2: Speedups of the evaluated biconnectivity algorithms on XMT relative to the serial DFS-based Hopcroft-Tarjan biconnectivity algorithm. For pDFS, the filled black bar marks a lower bound and the top of the "T" above the bar marks an upper bound. Key: pDFS = parallel DFS, TV = Tarjan-Vishkin, TV-BFS = Tarjan-Vishkin using BFS to find the spanning tree.

graphs being considered are known to be of low diameter, TV-BFS is preferable to TV. Also, for large, dense graphs, with many more edges than vertices, TV-BFS is likely to provide superior performance to TV.

- The presented results assume that given an input it is known which algorithm of the collage to apply. If this is not the case, then a default option would be to use TV, or pDFS if the ratio $|E|/|V|$ is sufficiently large.

- For the data sets considered, all of the algorithms except for pDFS benefit from increasing the number of hardware threads from 1,024 to 2,048 when enough parallelism is available. This is especially noticeable for TV-BFS on the 20kv-5me-random data set and TV on the web-Google-con data set. Biconnectivity algorithms are not very arithmetic-intensive, so additional hardware threads serve primarily to hide memory latency. This technique works as long as there is enough parallelism to keep all of the threads busy and enough bandwidth to DRAM to fulfill the additional requests. This case, where additional hardware threads are needed for latency hiding but not computation, suggests that it would be worthwhile to augment the XMT architecture with support for thread context switching, where each TCU stores two or more sets of thread state and switches contexts whenever a memory request blocks. The silicon area required to support context switching would be less than that required to increase the number of TCUs as functional units would not need to be duplicated.

- The speedups relative to the XMT MTCU, as shown in Figure 2b and the right half of Table 4, are between 1.3x and 3x larger than the corresponding speedups relative to the Core i7 920. Although we base our primary speedup claims on the Core i7, the speedups relative to the XMT MTCU are in a sense more relevant, as the MTCU reflects the same technology and engineering effort as the rest of the XMT architecture and we expect them to scale up at the same rate as they are further developed. This suggests that speedups of 21x to 48x could be obtained if XMT were brought up to industry grade on par with the Core i7.

The reported speedups are made possible by support in the XMT architecture for the efficient execution of programs with fine-grained, irregular parallelism. The XMT implementation of TV consists of many short parallel sections of code due to the synchronous nature of the algorithm. The instruction broadcast and prefix-sum network provide a low overhead for entering parallel sections and starting threads within a section, which allows even short threads to be profitable. Also, there are many indirect accesses to memory that, depending on the structure of the graph, may exhibit poor locality of reference. The TV algorithm provides a large amount of parallelism (one thread per vertex or per edge), which allows many memory requests to be issued in parallel, reducing the impact of the latency of any one request.

5. DISCUSSION

The discussion below suggests that contrary to common practice (or belief) there appears to be no principled need to compromise ease-of-programming in order to get strong speedups.

- The new NSF/IEEE-TCPP curriculum [4] views the PRAM model as overly simplistic. In contrast, using the XMT architecture we were able to obtain stronger speed-ups than in prior parallel biconnectivity studies, 9x to 33x through direct implementation of PRAM algorithms versus the previously reported of up to 4x in [12]. Interestingly, [12] was also driven by PRAM algorithms, though they had to work around an SMP architecture.

- Another example is the BFS algorithm. [4] also suggests teaching BFS. The recent paper [25] reported that none of the 42 students who took a joint UIUC/UMD parallel algorithms/programming class in Fall 2010 was able to get any speedups using OpenMP on an 8-processor SMP machine, while the speedups on a 64-processor XMT hardware, which uses at most 1/4 of the silicon area of the 8-processor machine, ranged between 7x and 25x. BFS is an example where OpenMP programming was not substantially different than XMT programming, but the XMT architecture allowed the speedup difference. See also the comment on bandwidth later in this section.

- The TCPP curriculum does not include any of the poly-logarithmic PRAM graph algorithms. However, this paper shows that they provide robust speedups on XMT that are unmatched by any of the graph algorithms the curriculum lists.

- Speedup problems with OpenMP are not new (for example, see [16]). The reason for comparing them with XMT above is that ease of programming is a priority for both. A short comparison on ease of programming follows. Teaching of XMT programming was done in parallel algorithms courses without any introduction to architecture and only a 20-minute introduction to XMT programming [25]. In contrast, the TCPP curriculum ranks parallel algorithms as third in priority of teaching after architecture and programming. Introduction of OpenMP is typically tied to architecture concepts such as the memory hierarchy.

- Interestingly, [10] and the current paper show that XMT is also competitive on performance with GPUs, which are performance-driven but are much more challenging to program effectively, as demonstrated in the comparison with [29, 30] in Section 4.2. The starting point of this research was that the SV parallel connectivity was given as a programming assignment in parallel algorithm courses at our university (name omitted) and was even solved by a couple of 10th graders in a course offered at a nearby high school. While our work reduced the total of operations (without the changing the basic work complexity of SV), our biggest effort was the extension beyond connectivity to biconnectivity. For this reason, the fact that no GPU biconnectivity implementation has been reported in spite of the mushrooming of GPU research is perhaps another demonstration of the practicality of XMT programming relative to GPU programming. Personal communication with the authors of [29] regarding the wording of their reference to possible use of their GPU connectivity program in a biconnectivity one confirmed that it was not meant to pass judgment on the relative difficulty of the two programs.

- In contrast to the implementation of biconnectivity for SMPs by Cong and Bader [12], which consists of over 5,800 lines of C code, our implementation for XMT only requires about 1,600 lines of code. Also, the effort required to tune and debug our implementation was comparable to that required for a serial program of similar size. In fact, serial debugging tools (GDB and Valgrind) were sufficient to catch and fix nearly all bugs in our parallel XMTC code.

- Much of the effort in writing the parallel biconnectivity code was in writing and tuning functions to perform basic tasks in parallel such as prefix sum, range-minimum queries, finding a spanning tree of a graph, and computing the preorder numbering of the nodes in a tree. These basic tasks are more general than biconnectivity and can be separated into a standalone library for reuse in other software projects.

- Using the above library, we plan to give biconnectivity as an optional programming assignment to a graduate class in the Spring 2012 semester. Providing the library to the students will reduce the complexity of the task to that of understanding how the PRAM algorithm works and seeing how the building blocks provided by the library can be assembled to construct a working implementation.

- Since the PMAM/PPoPP community is represented on both sides of the unfolding debates presented here, we thought that this discussion would be of interest in spite of its length.

For placing this debate in historical context, recall that claims that the main reason that parallel machines are difficult to program is that the bandwidth between processors and memories is so limited are not new, as formally demonstrated in [23, 34]. [8] suggested that: 1. Machine manufacturers see the cost benefit of lowering performance of interconnects, but grossly underestimate the programming difficulties and the high software development costs implied. 2. Their exclusive focus on runtime benchmarks misses critical costs, including: (i) the time to write the code, and (ii) the time to port the code to different distributions of data or to different machines that require different distribution of data. The XMT platform [6, 7] was finally able to demonstrate an affordable prototype providing the bandwidth that the 1994 paper [8] sought, but using today's technology.

Competition among hardware vendors in the desktop computing space has greatly diminished in recent years. Yet, the adoption of the few industry many-core solutions falls far behind serial platforms, which is a cause of extra concern. As believers in the eventual power of ideas, we are doing our best with XMT to keep some intellectual competition alive in spite of the huge funding gap with industry.

6. CONCLUSION

Of the biconnectivity algorithms evaluated, the logarithmic-time Tarjan-Vishkin algorithm, derived using PRAM algorithmic theory, provided the best performance overall. Of the parallel computing platforms evaluated, the XMT platform, designed with PRAM algorithms in mind, provided the best performance. These two facts demonstrate that with the proper many-core architecture, the relative simplicity of the PRAM can, perhaps surprisingly, be combined with the best performance.

More generally, this work provides another example that should help void PRAM criticism and address asymptotic analysis criticism. Criticism of the PRAM model has sometime been confused with criticism of the constants hidden by asymptotic analysis. In our opinion the XMT platform, which was originally inspired by PRAM algorithmics, and the performance it facilitated have voided much of the criticism on the PRAM model. However, one has to be a bit more careful with understanding the issue of constant factors. In the same way that theoretical papers on serial algorithms and their asymptotic analysis were often followed by separate efforts minimizing constant factors, the current work complements the original theory PRAM papers by reducing them to practice with respect to XMT, accounting for constant factors and concrete speedups. This often amounts to first modifying a published PRAM algorithm to another PRAM algorithm or other supporting data structures whose constant factors are better, which is, in fact, where the intellectual merit of this work lies; only then the revised PRAM algorithm is programmed for the XMT platform, which turns out to be a rather simple task. For biconnectivity, even optimizing LSRTM for performance tuning of XMT (per [33] as noted earlier), was adequately picked up by the compiler.

7. ACKNOWLEDGMENTS

We would like to thank the authors of [29] for providing us with the CUDA code for graph connectivity tested in this paper.

8. REFERENCES

[1] The ninth DIMACS implementation challenge: The shortest path problem. `http://www.dis.uniroma1.it/~challenge9/`, 2005.
[2] NVIDIA's next generation CUDA compute architecture: Fermi. `http://www.nvidia.com/content/PDF/fermi_white_papers/NVIDIA_Fermi_Compute_Architecture_Whitepaper.pdf`, 2009.
[3] Stanford network analysis platform. `http://snap.stanford.edu/index.html`, 2009.
[4] NSF/IEEE-TCPP curriculum initiative on parallel and distributed computing - core topics for undergraduates. `http://www.cs.gsu.edu/~tcpp/curriculum/index.php`, December 2010.
[5] D. Bader, G. Cong, and J. Feo. On the architectural requirements for efficient execution of graph algorithms. In *Proc. Int'l Conf. on Parallel Processing (ICPP)*, pages 547–556, June 2005.
[6] A. Balkan, M. Horak, G. Qu, and U. Vishkin. Layout-accurate design and implementation of a high-throughput interconnection network for single-chip parallel processing. In *Hot Interconnects 15*, pages 21–28, August 2007.
[7] A. Balkan, G. Qu, and U. Vishkin. An area-efficient high-throughput hybrid interconnection network for

single-chip parallel processing. In *Proc. IEEE/ACM Design Automation Conf.*, pages 435–440, June 2008.

[8] G. E. Blelloch, B. M. Maggs, and G. L. Miller. The hidden cost of low bandwidth communication. In U. Vishkin, editor, *Developing a computer science agenda for high-performance computing*, pages 22–25. ACM Press, New York, NY, USA, 1994.

[9] G. Caragea and U. Vishkin. Better speedups for parallel max-flow, brief announcement. In *Proc. ACM-SPAA*, 2011.

[10] G. C. Caragea, F. Keceli, A. Tzannes, and U. Vishkin. General-purpose vs. GPU: Comparison of many-cores on irregular workloads. In *HotPar '10: Proceedings of the 2nd Workshop on Hot Topics in Parallelism*. USENIX, June 2010.

[11] R. Cole and U. Vishkin. Deterministic coin tossing and accelerating cascades: micro and macro techniques for designing parallel algorithms. In *Proceedings of the eighteenth annual ACM symposium on Theory of computing*, STOC '86, pages 206–219, New York, NY, USA, 1986. ACM.

[12] G. Cong and D. Bader. An experimental study of parallel biconnected components algorithms on symmetric multiprocessors (SMPs). In *Proc. 19th IEEE International Parallel and Distributed Processing Symposium.*, page 45b, April 2005.

[13] D. Culler and J. Singh. *Parallel Computer Architecture: A Hardware/Software Approach*. Morgan-Kaufmann, 1999.

[14] S. Dascal and U. Vishkin. Experiments with list ranking for explicit multi-threaded (XMT) instruction parallelism. *J. Exp. Algorithmics*, 5, December 2000.

[15] D. M. Eckstein. *Parallel graph processing using depth-first search and breadth-first search*. PhD thesis, University of Iowa, 1977. AAI7728449.

[16] K. Fürlinger and M. Gerndt. Analyzing overheads and scalability characteristics of OpenMP applications. In *High Performance Computing for Computational Science - VECPAR*. 2006.

[17] J. Hopcroft and R. Tarjan. Algorithm 447: efficient algorithms for graph manipulation. *Commun. ACM*, 16(6):372–378, June 1973.

[18] U. Kang, C. Tsourakakis, A. P. Appel, C. Faloutsos, and J. Leskovec. HADI: Fast diameter estimation and mining in massive graphs with Hadoop, 2008.

[19] H. J. Karloff, S. Suri, and S. Vassilvitskii. A model of computation for MapReduce. In *Proceedings of the Twenty-First Annual ACM-SIAM Symposium on Discrete Algorithms (SODA)*, pages 938–948, 2010.

[20] F. Keceli, T. Moreshet, and U. Vishkin. Power-performance comparison of single-task driven many-cores. Under review.

[21] F. Keceli, A. Tzannes, G. C. Caragea, R. Barua, and U. Vishkin. Toolchain for programming, simulating and studying the XMT many-core architecture, 2010. Under review.

[22] E. Lindholm, J. Nickolls, S. Oberman, and J. Montrym. NVIDIA Tesla: A unified graphics and computing architecture. *IEEE Micro*, 28:39–55, 2008.

[23] Y. Mansour, N. Nisan, and U. Vishkin. Trade-offs between communication throughput and parallel time. In *Proc. 26th Annual ACM Symp. on Theory of Computing*, pages 372–381, 1994.

[24] J. Nickolls and W. Dally. The GPU computing era. *IEEE Micro*, 30(2):56–69, March-April 2010.

[25] D. Padua, U. Vishkin, and J. Carver. Joint UIUC/UMD parallel algorithms/programming course. In *First NSF/TCPP Workshop on Parallel and Distributed Computing Education (EduPar-11) , in conjunction with IPDPS*, Anchorage, Alaska, May 16, 2011.

[26] J. H. Reif. Depth-first search is inherently sequential. *Information Processing Letters*, 20(5):229–234, 1985.

[27] Y. Shiloach and U. Vishkin. An $O(\log n)$ parallel connectivity algorithm. *J. Algorithms*, 3(1):57–67, 1982.

[28] Y. Shiloach and U. Vishkin. An $O(n^2 \log n)$ parallel max-flow algorithm. *J. Algorithms*, 3(2):128–146, February 1982.

[29] J. Soman, K. Kishore, and P. Narayanan. A fast GPU algorithm for graph connectivity. In *Parallel Distributed Processing, Workshops and PhD Forum (IPDPSW), 2010 IEEE International Symposium on*, pages 1–8, April 2010.

[30] J. Soman, K. Kishore, and P. Narayanan. Some GPU algorithms for graph connected components and spanning tree. *Parallel Processing Letters*, 20(4):325–339, December 2010.

[31] R. E. Tarjan and U. Vishkin. An efficient parallel biconnectivity algorithm. *SIAM J. Computing*, 14(4):862–874, 1985.

[32] U. Vishkin. Thinking in parallel: Some basic data-parallel algorithms and techniques. `http://www.umiacs.umd.edu/users/vishkin/PUBLICATIONS/classnotes.pdf`, February 2009.

[33] U. Vishkin, G. Caragea, and B. Lee. Models for advancing PRAM and other algorithms into parallel programs for a PRAM-on-chip platform. In S. Rajasekaran and J. Reif, editors, *Handbook on Parallel Computing: Models, Algorithms, and Applications*, chapter 5. Chapman and Hall/CRC Press, 2008.

[34] U. Vishkin and A. Wigderson. Trade-offs between depth and width in parallel computation. *SIAM J. Computing*, 14(2):303–314, 1985.

[35] X. Wen and U. Vishkin. FPGA-based prototype of a PRAM-on-chip processor. In *Proceedings of the 5th conference on Computing frontiers*, CF '08, pages 55–66, New York, NY, USA, 2008. ACM.