Parallel Algorithms for Graph Problems

Nathaniel Crowell
University of Maryland
ncrowell@cs.umd.edu

Abstract

This work demonstrates good speedups for the Scalable Synthetic Compact Application 2 (SSCA2) and algebraic connectivity for small irregular graphs on the Explicit Multi-Threading (XMT) many-core architecture. Previous studies of these algorithms have been focused on using high performance computing architectures to solve large instances of the problems but little work has been done that establishes performance of these parallel algorithms on problem sizes solvable on a smaller scale system. Additionally, analysis of the algorithms is presented that shows how the ease of programming approach for XMT creates a clear path for developers to utilize multi-core resources without having to completely recreate the algorithm for the specific architecture by using the well-studied Parallel Random Access Model (PRAM) of algorithmic thinking. The compiler provided by the UMD XMT research team further makes it possible to trivially parallelize many segments of code and achieve acceptable speedup for the developer time invested. Future innovations from the UMD XMT research team will result in this process becoming even simpler with larger performance gains provided.

1. Introduction

Processors with multiple cores have become ubiquitous in desktop machines. Generally, software has failed to keep up and make use of the additional computing power made available by these newer processors. As manufactures such as Intel push to many-core architectures with twenty or more cores the problem will only get worse. The insufficient solution is that users will simply make use of this additional processing power by running more processes; however, such a solution concedes the ability to continue to provide faster and more powerful applications. In the past such improvements to software performance were free from the perspective of the software developer who could just wait around for a processor with a faster clock speed to be released. Examining the current course catalog and graudation requirements of many universities provides evidence that many trained computer scientists leave college without the necessary skills to efficiently program for a multiple core environment.

Computations on sparse graphs represent a class of algorithms for which the problem is made even more challenging. Even with proper training to develop parallel applications, a developer has to overcome the fact the majority of current computing systems can only manage a fraction of the computational peak performance on such problems [20]. Sparse graphs consume large amounts of memory and their memory access patterns exhibit low degrees of spatial and temporal locality. Additionally, the computation to memory access ratio is usually low enough that memory becomes the bounding factor to performance on most systems [20]. Such problems are defined as being irregular and their poor memory access patterns cause additional problems when trying to develop efficient parallel algorithms. Typically, the strength of additional cores is a direct improvement in computational capabilities but the additional memory overhead for synchronization reduces their benefit for these irregular problems. Most work in the area of parallel graph algorithms has been focused on computations over very large datasets [15, 20, 27] which result still in small fractions of peak performance but solve problems that would be otherwise intractable for serial computation.

A research team at UMD has designed and created a prototype of an architecture designed for improving single-task performance and ease of programming. The Explicit Multi-Threading (XMT) general-purpose architecture is supported by a programming model implementation based upon a high level language extension to standard C. The language extension adds a small number of primitives to support explicit parallelism which are very effective in achieving the goal of providing a good way to implement programs derived from Parallel Random Access Machine/Model (PRAM) algorithms [32].

This paper contributes to the understanding of parallel graph algorithm performance on a scale that more closely resembles a desktop environment. A large body of work exists which already addresses large scale high performance computing (HPC) performance of parallel graph algorithms; however, the majority of programmers are unlikely to be developers for such systems, so an incentive exists to provide performance metrics which encourage the use of parallel algorithms in the design of applications intended to run on systems whose memory capacity and number of cores/processors resembles current and not-too-distant-future desktop environments.

Achieving good speedup on problem which exhibit low levels of parallelism (compared to that typical for HPC) is an important goal. Complex applications such as user applications are likely to provide such levels of parallelism which should not just simply be ignored. Additionally, the parallelism is likely to vary over time much more than standard HPC benchmarks. Two very important goals in the current era of computation are the development of an architecture which is capable of exploiting these levels of parallelism effeciently and a programming model which simplies the process of parallel program design and implementation.

I examined and implemented algorithms for the Scalable Synthetic Compact Application 2 (SSCA2) benchmark suite and for the calculation of algebraic connectivity of a graph. Algebraic connectivity plays an important role in the ideal bisection and clustering of vertices in a graph [5, 18].

The XMT architecture is defined in Section 2. In section 3, I review the SSCA2 benchmarks. Section 4 provides an overview of algebraic connectivity. The implementation details of the parallel algorithms are described in Section 5. Benchmark results are provided in Section 6 and related work is reviewed in Section 7.

2. XMT

The Explicit Multi-Threading (XMT) general-purpose architecture is designed for improving single-task performance and ease of programming. The XMT programming model is implemented by XMTC, a high level language extension to standard C. The language only adds a small number of primitives to support explicit parallelism [30]. The XMT architecture has been demonstrated to show remarkable speedup [8–10] in comparison to commodity parallel processors. Additionally, the programming model has previously been shown to be much easier to learn and utilize effectively than competing expressions of parallelism [28].

2.1 Architecture

Design of the Explicit Multi-Threading (XMT) on-chip general-purpose computer architecture has been directed toward improvements in single-task performance. While alternative parallel architectures rely on embarrassingly parallel algorithms in order to achieve expected speedups, XMT was designed to leverage the vast existing body of knowledge relating to Parallel Random Access Model (PRAM) algorithms. [19]

A significant component of the XMT architecture (Figure 1) is an array of thread control units (TCUs), or cores. A Master TCU exists which has its own cache and is responsible for execution of serial code. The remaining TCUs are grouped into clusters that are connected by a high-throughput interconnect. A methodology called "independence of order semantics (IOS)" is implemented which ensures that during execution of parallel code, each thread advances at its own rate without having to busy-wait for

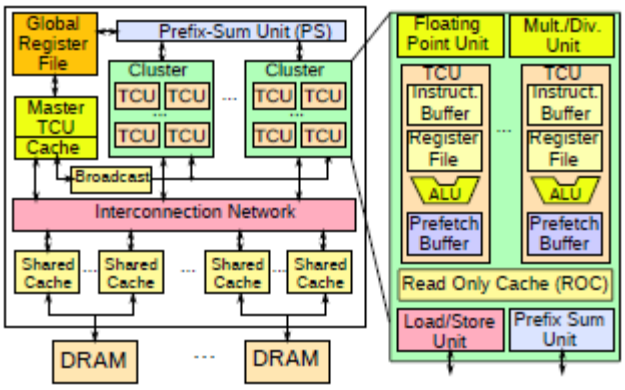

Figure 1. XMT Architecture.

other threads. Thus the programming model is PRAM-like with arbitrary concurrent read and concurrent write (PRAM-CRCW). An atomic prefix-sum primitive is provided and is implemented in hardware. This low overhead primitive is essential to the inter-thread coordination required to implement some algorithms that leverage irregular parallelism. [19]

The XMT architecture has been prototyped using FPGA technology. The prototype Paraleap runs a 64-core configuration running at 75MHz and shows the feasibility of PRAM-On-Chip as an architecture. The total silicon area is equivalent to that of approximately one or two current commercial processor cores. Additionally, a simulator XMTSim can provide cycle accurate simulation of software running on a variety of architecture configurations. [19]

2.2 Programming Model

The XMT programming model provides fine-grained control over parallelism in a single program, multiple data execution model. The programming model adds three basic primitives to facilitate expressing parallelism: spawn, join, and prefix-sum. Spawn starts a parallel section of a program. Join indicates the point of termination for a thread. Prefix-sum defines an atomic operation that enables threads to coordinate computation while respecting the arbitrary concurrent-write PRAM model. Figure 2 shows the generic switching between serial and parallel modes of execution that could occur during a program. [29]

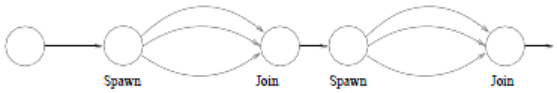

Figure 2. Program with serial and parallel execution modes.

XMTC is an implementation of the XMT programming model as an extension to the C programming language. XMTC adds a small number of instructions: spawn and prefix-sum. Additionally, standard libraries are being de-

veloped for commonly needed functions. The join command of the XMT programming model is implicit in XMTC. A spawn instruction indicates the beginning of a code block requiring parallel execution. The end of this code block implicitly functions as a join. Prefix-sum utilizes a hardware implemented mechanism for atomically adding a value to another already in register memory. The extension also defines a prefix-sum to memory operation which is more flexible but less efficient. By convention, threads in XMTC can refer to their own thread ID using the $ character. The spawn instruction takes arguments indicating the lowest thread ID and the highest thread ID to be spawned for the parallel code block. In following the ease of programming philosophy, variable declarations within a spawn code block are assumed to be thread local variables. [29]

Figure 3 demonstrates the simplicity of XMTC through an example that swaps values in arrays of length n. The XMT research team at University of Maryland has developed a compiler for programs written using XMTC. The compiled programs can be executed on their publicly available simulator (and can be implemented on their FPGA implementation of the XMT architecture). [32]

```
spawn(0, n-1) {
        int x;
        x = A[$];
        A[$] = B[$];
        B[$] = x;
}
```

Figure 3. Parallel array swap.

3. SSCA2 Benchmark

The first Scalable Synthetic Compact Application 2 (SSCA2) benchmark suite was a product of the DARPA High Productivity Computing Systems (HPCS) program. The motivation for a separate benchmark directed at sparse graph analysis is in large part due to these algorithms being highly memory intensive (i.e., large memory footprint, low degree of locality in memory access patterns, and a low computation to memory access ratio). Sparse graph analysis performance strongly correlates with memory subsystem performance rather than processor clock frequency or availability of arithmetic logic units. Parallel design of such algorithms is considered challenging as the partitioning over processors is often difficult without imposing large overhead latencies. [20]

The SSCA2 benchmark suite represents key components of graph analysis algorithms that occur in applications such as social network analysis, epidemiological studies, and network analysis in systems biology. It was developed with the intent to become a compact application that uses multiple analysis techniques all accessing a single data structure representing a weighted, directed graph. The intention is to provide a fair benchmark of graph analysis techniques in the general case, as modifying the data structure to optimize one kernel might negatively impact another analysis kernel. [20]

Version two of the benchmark specification was originally released in August 2006 [20]. SSCA2 consists of four kernels (analysis techniques) that operate on a synthetic graph (generated using the Recursive MATrix random graph generation algorithm). The kernels represent common irregular problems on graphs. The most complex kernel of the four is commonly considered the evaluation of betweenness centrality (kernel 4) and has been the focus of many recent papers [15, 20, 27]. [3]

3.1 Scalable Data Generator

The scalable data generator constructs power-law graphs using an algorithm based on the Recursive MATrix (R-MAT) scale-free graph generation algorithm. The algorithm provides a repeatable, verifiable mechanism for graph generation. Additionally, the graphs generated are understood to be representative of commonly analyzed real-world networks [20]. The input to the data generator is the graph scale. From the scale, the total number of vertices, the total number of edges, and the maximum (integer) value of an edge weight is determined in the following way. The number of vertices is $n = 2^{scale}$. The number of edges is $m = 8n$. The maximum edge weight is simply $maxWeight = n$. Internal parameters are used to ensure that the output graph is a power-law graph. The output of the data generator is a list of edge tuples containing start vertex, end vertex, and edge weight. [3]

3.2 Kernel 1 – Graph Construction

This kernel establishes the graph representation that will be used for the remaining three kernels. In the reference implementation, graph construction consists of processing the edge list from the data generator and storing the graph as an incidence list. A key detail of SSCA2 is that this data structure cannot be modified after the completion of kernel one (i.e., the same representation must be used for kernels two through four). [3]

3.3 Kernel 2 – Classify Large Sets

This kernel determines the set of edges which have the largest weight (i.e., if there are 8 edges of equal weight and no edges with more weight then the set would contain these 8 edges). The output from this kernel is an edge list containing the determined set of edges to be used by the following kernel. [3]

3.4 Kernel 3 – Graph Extraction

This kernel performs the following set of operations: for each edge e in the set produced by kernel 2 create a subgraph consisting of all paths of a defined length (i.e., it is a kernel parameter) starting with that edge e. This is with no regard to the actual weights of the edges. For this kernel, breadth first search is the typical solution. [3]

3.5 Kernel 4 – Graph Analysis Algorithm

This kernel computes betweenness centrality. Betweenness Centrality is a centrality metric based on an enumeration of all shortest paths in the graph. The betweenness centrality score of any given vertex in a graph is defined as

$BC(v) = \sum \frac{\sigma_{st}(v)}{\sigma_{st}}$

where σ_{st} denotes the number of shortest paths between vertices s and t, and $\sigma_{st}(v)$ is the number of those paths that pass through vertex v. The summation is taken over all values of $\frac{\sigma_{st}(v)}{\sigma_{st}}$ for which all three vertices are distinct. [3]

An efficient serial algorithm was proposed by Brandes [7] which computes exact betweenness centrality for all vertices in a unweighted graph in $O(nm)$ time and $O(n+m)$ space. Brandes notes a recursive relationship in betweenness centrality scores that can be exploited by a modification to Dijkstra's single-source shortest paths algorithm in which predecessor sets are formed as the graph is traversed. Betweenness centrality scores can be calculated using dependency relationships which correlate directly with the predecessor sets. [20]

Although the complexity is polynomial in time, computing the exact betweenness centrality of networks with millions of nodes and edges can be intractable. Thus, a commonly accepted approximation uses the "Brandes' Algorithm" to perform a constant number of breadth-first search expansions from a random sample of starting nodes instead of all of the nodes thus bounding the time complexity by the number of edges. [15]

Obviously, betweenness centrality provides a measure of the control a vertex has over communication in the network. Also it can be used to identify the most critical vertices in the network as well as the least important. High betweenness centrality indicates that relatively short paths exist between the vertex and other vertices in the graph. Betweenness centrality has been used in a variety of applications including routing in road networks [15], terrorist networks [20], and the study of sexual networks and AIDS. [20]

The design of this kernel makes the choice that all edges with a weight evenly divisible by eight are not considered in calculating betweenness centrality. Despite this restriction, the edges are otherwise considered without regard to weight during the computation. Evaluation of the kernel requires an all-pairs shortest paths analysis over the graph which finds all of the shortest-paths between each pair of vertices when many exist (as opposed to some algorithms which report only one such path when many exist). In fact, there is no known algorithm for computing exact betweenness centrality of a vertex without solving an all-pairs shortest paths problem on the graph [20]. Approximate implementations are permitted for this kernel. Performance computing betweenness centrality correlates strongly with performance computing related properties such as stress centrality and other metrics based on counting shortest paths [15]. [3]

4. Algebraic Connectivity and the Fiedler Vector

The Laplacian matrix of an undirected graph is defined in the following way. Let $G = (V, E)$ be an undirected weighted graph with positive weights $\{w_{ij}\}$. The weighted Laplacian $L(G)$is defined to be the $n \times n$ symmetric matrix

$$L(G) = D - W.$$

W has entries

$$(W)_{ij} = \begin{cases} w_{ij} & if(v_i, v_j)\epsilon E \\ 0 & otherwise \end{cases}$$

D is the diagonal matrix with entries $(D)_{ij} = \sum_i W_{ij}$. [17]

In 1975, Fiedler discovered that the eigenvector corresponding to second smallest eigenvalue λ_2 of the weighted Laplacian of a graph has special properties related to the connectivity of the corresponding graph [14]. His initial work on what is now referred to as *algebraic connectivity* provided the theoretical justification to use this eigenvector to partition graphs. As a result, the eigenvector of the second smallest eigenvalue of the Laplacian of a graph is called the Fiedler vector. The Fiedler vector is all real-valued and ordering its components monotonically (increasing or decreasing) can be used to induce a permutation of the vertices known as a spectral ordering. The important property of this ordering is that it relates to connectivity properties of the graph. Any component of the ordering x_j is connected to both its set of predecessors and set of successors if $(x_{j-1} < x_j < x_{j+1})$ (i.e., the value is distinct in the sorted Fiedler vector). [17]

Algebraic connectivity and the induced spectral ordering has a number of applications. Algebraic connectivity has been used to obtain an upper-bound on the max-cut problem (a well known NP-hard problem in combinatorial optimization) [1]. Fiedler vectors have found uses in analysis of RNA structures [4] and in decomposition of large scale finite element models found in the field of mechanical engineering [18]. Additionally, the Fiedler vector has been used in graph bisection algorithms [5].

5. Implementation Details

5.1 SSCA2

The first three kernels of the SSCA2 benchmark are relatively trivial to parallelize efficiently. Kernel 1 (Algorithm 1) is embarrassingly parallel after computation of a prefix sum over the degree of each vertex in the graph. Kernel 2 (Algorithm 2) is similarly trivial requiring only a max operation over the set of edge weights and then some coordination for each edge having this maximum weight being added to a queue. Kernel 3 (Algorithm 3) is a simple parallel breath-first search which has been studied previously

on the XMT [29]. The only addition here is that for Kernel 3, multiple breadth-first search expansions are performed in parallel as memory allows. Additionally, subroutines for prefix-sum, sum and max are all implemented using k-ary tree algorithms which have previously shown good results [29].

Algorithm 1 SSCA2 Kernel 1

Input: E the set of edges tuples for the graph of the form $< start, end, weight >$. n is the number of vertices in the graph. m is the number of edges in the graph.

Output: The graph $G(V, E)$ stored as an incidence list structure containing the arrays numedges, endVertex, and weight.

1: $d \leftarrow$ n element array initialized to all zeros.
2: **for** $i \leftarrow 1$ to m **in parallel do**
3: $u \leftarrow$E$_i$.start
4: $t \leftarrow$ fetch_and_add($\&d_u, 1$)
5: $position_i \leftarrow t$
6: **end for**
7: numedges $\leftarrow$ [0,prefix_sum(d)]
8: **for** $i \leftarrow 1$ to m **in parallel do**
9: $u \leftarrow$E$_i$.start
10: $v \leftarrow$E$_i$.end
11: $w \leftarrow$E$_i$.weight
12: $t \leftarrow$numedges$_u + position_i$
13: endVertex$_t \leftarrow v$
14: weight$_t \leftarrow w$
15: **end for**

Algorithm 2 SSCA2 Kernel 2

Input: The graph $G(V, E)$ stored as an incidence list structure containing the arrays numedges, endVertex, and weight.

Output: An array S of $< start, end, weight >$ tuples containing the determined start points.

1: $m \leftarrow$ max(weight)
2: $t \leftarrow 0$
3: **for** $i \leftarrow 1$ to m **in parallel do**
4: $w \leftarrow$ weight$_i$
5: **if** $w = m$ **then**
6: $u \leftarrow$ searchEdgeListL(i)
7: $v \leftarrow$ endVertex$_i$
8: $p \leftarrow$ fetch_and_add($\&t, 1$)
9: S$_p \leftarrow < u, v, w >$
10: **end if**
11: **end for**

In the algorithm listings, an assumption is made regarding the existence of atomic memory operations. For example, line 8 of Algorithm 2 performs fetch_and_add which atomically fetches the value from the specified memory address, adds 1 to the value and stores the result back in memory. Additionally, the original value is returned to the local thread.

Algorithm 3 SSCA2 Kernel 3

Input: $G(V, E)$, S output from Kernel 2, and an indication of the maximum depth to explore, $maxdepth$.

Output: L which is a list of $< vertex, depth >$-tuple lists. One edge list per breath-first search expansion.

1: $Q \leftarrow$empty stack
2: $depth \leftarrow 1$
3: $count \leftarrow 0$
4: **for all** $< p, q, w > \in S$ **in parallel do**
5: $L_{pq} \leftarrow$ empty stack
6: push $< p, -1 > \rightarrow L_{pq}$
7: push $< q, 1 > \rightarrow L_{pq}$
8: push $q \rightarrow Q$
9: $count \leftarrow count + 1$
10: **while** $count > 0$ and $depth <$ $maxdepth$ **do**
11: **for all** $v \in Q$ **in parallel do**
12: remove v from Q
13: $count \leftarrow 0$
14: **for** each neighbor w of v **in parallel do**
15: **if** w is not in L_{pq} **then**
16: push $< w, depth > \rightarrow L_{pq}$
17: push $w \rightarrow Q$
18: $count \leftarrow count + 1$
19: **end if**
20: **end for**
21: **end for**
22: one thread performs: $depth \leftarrow depth + 1$
23: **end while**
24: **end for**

Instances such as line 18 of Algorithm 3 which increment a value by one do so atomically. For the algorithms which perform a breadth-first search (Algorithm 3 and Algorithm 4), the algorithms need to be able to check if a node has been visited then perform some operations if it has not (such as lines 15-18 of Algorithm 3). In these cases, only one thread should execute the portion of the algorithm protected by the conditional so it is appropriate to think of the entire conditional and code block as atomic. The XMTC implementation utilizes locks and atomic memory primitives in order to ensure that no two threads will enter the code block protected by the conditional.

The betweenness centrality computation of Kernel 4 presents the most challenging algorithm to efficiently parallelize and there have been many attempts to do so [2, 15, 20, 27]. My XMTC implementation (Algorithm 4) closely resembles that presented in [20]. The algorithm is split in to three phases which are performed once for each vertex in the graph (or some subset of the vertices when performing an approximation instead of exact analysis). The first phase (lines 5 to 16) is the initialization which establishes that no vertex is currently on any shortest paths for the current iteration.

The second phase (lines 17 to 35) is the graph traversal. The algorithm proceeds in a similar manner to breadth-first search; however, for betweenness centrality a node can be visited from multiple predecessors instead of just one as is typical for breath-first search. The first time a node is visited (lines 22-25) establishes the depth of that node for the current iteration. Multiple visitations may occur if the node is found to exist on additional shortest paths. During each visitation, the node and its predecessor (the "predecessor edge") are added to the set of predecessors for the current depth ("phase") of the traversal. Additionally, the σ value is incremented. σ indicates how many shortest paths the particular node was found to be on.

The third phase (lines 36 to 46) exploits the recursive relationship of betweenness centrality by examining the set of predecessor edges collected during phase three in reverse order. As there are no dependency relationships among edges collected during the same phase, this computation can be performed in parallel over all edges found in one phase at a time.

The set of predecessor edges is where my implementation varies the most from that presented in other work. The algorithm presented in[20] builds predecessor sets for each node in the graph as it is traversed and requires a separate list for every node in the graph in order to do. Synchronized access to all of the lists was identified as a potential challenge to performance. Due to the availability of the prefix-sum primitive in XMTC, the algorithm can efficiently be implemented using one list which all threads can easily synchronize their accesses to.

5.2 Parallel Davidson Eigensolver

To compute the Fiedler vector of graphs, I implemented a parallel Davidson eigensolver as described in [6]. The Davidson for Several Eigenvalues algorithm iteratively finds each eigenvalue and eigenvector of the graph Laplacian focusing on one eigenpair at a time beginning with the smallest. This approach to finding the Fiedler vector is attractive as it only requires finding the first two eigenpairs of the Laplacian without concern for the remaining eigenpairs. In fact the loop can be designed such that the first eigenpair is ignored and computation just focuses on finding the Fiedler vector. The pseudocode for the serial algorithm is shown in Algorithm 5. The loop is partially unrolled so that the termination condition is on the loop condition instead of buried in the middle of the loop as it has previously appeared in literature [6].

The algorithm utilizes the matrix slice operator in order to simplify instances where only portions of the matrix are used in a calculation. The slice operator $[a, b]$ when applied to a matrix extracts the range of rows indicated by a and their respective columns indicated by b. Ranges are of the form $start : end$. Omitting the $start$ indicates including all rows (or columns) to the left (or above) of the end (and similarly omitting the end indicates the same but in the opposite direction). Thus, omitting both indicates using all of the rows or columns (depending upon which range is being specified).

The Davidson method is generally easy to understand. The strategy is to build up an approximation of the orthonormal basis of the graph Laplacian. At each iteration, an approximation to the desired eigenvalue is calculated along with its corresponding approximate eigenvector. The residual of the approximation is calculated based on the definition of an eigenpair. The residual provides an indication of how well the eigenpair approximates an actual eigenpair of the Laplacian. If the residual is small enough then the result is considered to have converged and the resulting eigenpair is returned. If it is has not converged then the orthonormal basis is expanded by adding another column called the correction vector and proceeding to the next iteration. The challenges are to perform each step as efficiently as possible and to do so in a stable way. Stability is maintained by using a stable eigensolver to calculate the estimate in each iteration and obtaining the correction vector by applying a preconditioner to the residual. [12]

The algorithm requires an initial guess x which is a normalized randomly generated (from a normal distribution) column vector. Algorithm 5 is partially unrolled such that the first trivial iteration is mostly outside of the loop (lines 3 to 10). Iterations of the general algorithm begin on line 18, progress to the end of the loop, and possibly terminate after calculation of the residual.

The basis for this implementation of the Davidson method relies upon solving an easier eigenvalue problem. The Laplacian of the graph is approximated by an arrowhead matrix for which a solution can more easily be found. An arrowhead matrix is one which contains zeros everywhere except for the main diagonal, the last row, and the last column . Additionally, the transpose of the last row is equivalent to the last column and the main diagonal contains values that are monotonically increasing [23]. After finding the eigenpairs of the arrowhead matrix (line 26), the results are used to calculate an approximate eigenvector of the graph Laplacian (line 32) for which a residual is calculated and convergence is tested. If convergence fails then a preconditioner matrix is calculated (line 14), the preconditioner is applied to the residual to find the correction vector, and then the correction vector is orthonormalized using the Modified Gram-Schmidt method (MGS) (line16). MGS is a very simple but effective method to orthonormalize a vector relative to a set of basis vectors (Algorithm 7). After adding the orthonormalized correction vector to the basis approximation, the next iteration begins.

For the arrowhead matrix eigenproblem, the stable algorithm identified in [23] is utilized (Algorithm 6). The algorithm utilizes a property of arrowhead matrices which states that each eigenvalue of the matrix has a defined upper and lower bound based upon the contents of the main diagonal.

Algorithm 5 Davidson for Several Eigenvalues

Input: A is the nxn Laplacian of the graph $G(V, E)$, x is the initial guess of the eigenvector, and ϵ is the termination threshold.

Output: $eigenpair(\lambda, u)$

1: $v \leftarrow x$
2: $k \leftarrow 1$
3: $w \leftarrow Av$
4: $W \leftarrow w$
5: $\lambda \leftarrow 1$
6: $\Lambda \leftarrow \lambda$
7: $z \leftarrow v^T w$
8: $Y \leftarrow z$
9: $u \leftarrow VY$
10: $r \leftarrow Wy - \lambda u$
11: // End of initialization. Risidual is exceedingly unlikely to converge at this point.
12: **while** $||r|| > \epsilon$ **do**
13: // Calculate correction vector and update orthnormal basis.
14: $M \leftarrow A - \lambda I_n$
15: $v \leftarrow Mr$
16: $v \leftarrow mgs(V, v)$
17: $V \leftarrow [V, v]$
18: // Start of iteration.
19: // Expand Arrowhead approximation of A based on updated orthonormal basis.
20: $k \leftarrow k + 1$
21: $w \leftarrow Av$
22: $W \leftarrow [W, w]$
23: $s_{kk} = v^T w$
24: $s_{bk} = Y^T V[:, 1 : (k - 1)]w^T$
25: // Calculate eigenpairs of the approximation of A.
26: $\Lambda, Z \leftarrow arrowheadEigsolver(\Lambda, s_{bk}, s_{kk})$
27: $Y \leftarrow YZ[1 : k - 1, :]$
28: $Y \leftarrow [Y; Z[k - 1, :]]$
29: $\lambda = \Lambda[1]$
30: $y = Y[:, 1]$
31: // Calculate the approximate eigenvector and its corresponding residual.
32: $u = Vy$
33: $r = Wy - \lambda u$
34: **end while**

This property allows for a zero finding algorithm to be used to determine each of the eigenvalues (line 5). From these eigenvalues, the set of corresponding eigenvectors can be calculated (lines 7 to 11).

Parallelizing this Davidson algorithm was simple due to the ease of programming model provided by XMTC. All matrix operations were parallelized in the obvious way. Additionally, the loop which finds an eigenvalue of the arrowhead

Algorithm 6 Arrowhead Matrix Eigensolver

Input: The components of an nxn arrowhead matrix. d is the first $n - 1$ elements of the main diagonal. e is the first $n-1$ elements of both the last row and last column. p is the element of the matrix in the lower right corner.

Output: the set of eigenvalues Λ and the corresponding eigenvectors as columns of the matrix Z.

1: $d_0 \leftarrow min(d_1 - |e_1|, ...d_{n-1} - |e_{n-1}|, p - \Sigma|e_i|)$
2: $d_k \leftarrow max(d_1 + |e_1|, ...d_{n-1} + |e_{n-1}|, p + \Sigma|e_i|)$
3: $D < -[d_0, d, d_k]$
4: **for** $i = 0$ to $n + 1$ **do**
5: $\lambda_i \leftarrow findeig(D_i, D_{i+1}, d, e, p)$
6: **end for**
7: **for** $i = 0$ to $n + 1$ **do**
8: **for** $j = 0$ to n **do**
9: $Z_{j,i} = e_j/(\lambda_i - d_j)$
10: **end for**
11: **end for**

Algorithm 7 Modified Gram-Schmidt

Input: v is the vector being orthonormalized relative to V. n is the number of columns in r.

Output: v after orthonormalization.

1: **for** $i \leftarrow 1$ to n **do**
2: $t \leftarrow (V[:, i]^T v)V[:, i]$
3: $v \leftarrow v - t$
4: **end for**
5: $v \leftarrow v/||v||$

matrix during each iteration (Algorithm 6 lines 4 to 6) was parallelized trivially.

6. Results

My implementations of the Scalable Synthetic Compact Application 2 (SSCA2) benchmark suite for graph analysis were benchmarked on the Paraleap 64-core XMT prototype. The serial implementation was benchmarked using the Master TCU. The parallel implementation used the full prototype.

The inputs for SSCA2 were generated independently using Python and included in the binaries at compile time. In the future, the XMT will have support for file operations allowing for the possibility to load data at run time. Five random seeds were used at each graph scale between four and sixteen. In comparison to other work benchmarking using SSCA2, these are small graphs; however, the current XMT prototype is limited to 1GB of memory. Additionally, the programming model for prototype only supports an address space of 4GB. Therefore I am measuring the amount of parallelism the architecture can achieve on small graphs. Small graphs are often overlooked in the benchmarks of parallel algorithms as for other systems the overhead of parallelism is

often too large to utilize for relatively short tasks; however, progressing to a paradigm where desktop machines will have eight or more cores provides an environment where the need for speedup on such problems becomes more apparent.

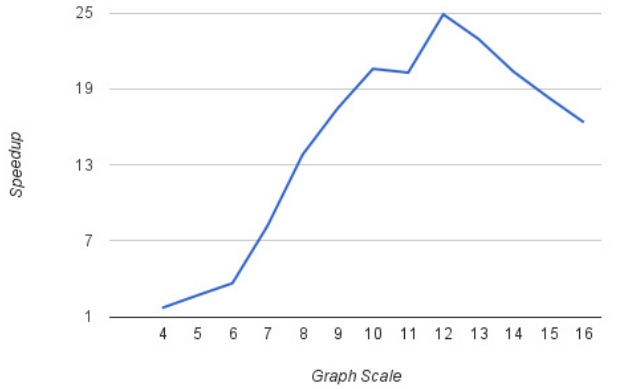

Figure 4. SSCA2 Kernel 1 Speedup

SSCA2 Kernel 1 shows the expected speedup as the only limiting factor to parallelism is a prefix sum over the degree of each node in the graph. Figure 4 shows how the speedup increases with graph size as the XMT is able to gradually utilize more cores effectively in solving the problem; however, the speedup peaks at 24.88 with graph scale 13 and begins to decline. A reason that I expect for this behavior is the synchronization overhead (from the k-ary tree prefix-sum subroutine) continues to grow with the size of the graph while the number of TCUs remains constant. A similar effect is observed in [26].

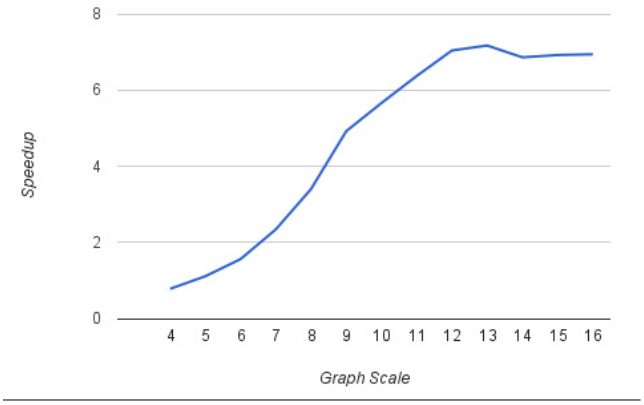

Figure 5. SSCA2 Kernel 2 Speedup

SSCA2 Kernel 2 shows performance similar to Kernel 1. In Kernel 2 the majority of the work is the calculation of the maximum weight. As in Kernel 1, the overhead induced by the k-ary tree computation of the primary subroutine (in this case max) continues to grow and reduces speedup after scale 13 (Figure 5). The speedup for this kernel peaks with a value of 7.17. One might consider that the k-ary computation is unnecessary since the maximum possible weight is known a priori and will be the actual maximum with high probability; however, the specification of the benchmark requires that the maximum among the actual edge weight be calculated as though the maximum weight could be unbounded (i.e., not known a prior).

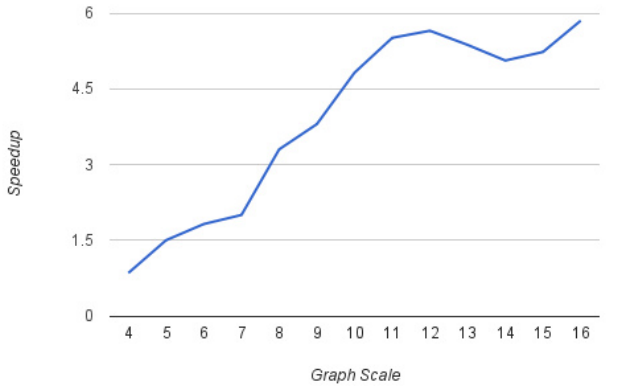

Figure 6. SSCA2 Kernel 3 Speedup

SSCA2 Kernel 3 shows excellent performance for breadth-first search on small graphs with limited depth (Figure 6). The speedup reaches 5.51 at scale 11 and continues to increase. A slight decrease occurs at scales 14 and 15, but the trend of the speedup is such that with more memory (to store graphs of a larger size) the speedup could continue to grow. Previous work with the XMT prototype [29] has shown greater speedup on the standard parallel breadth-first search algorithm which explores the entire graph; however, that work considered graphs with edge factors (ratio of edges of vertices) that were much larger than the graphs generated for SSCA2. The reference benchmark sets the maximum depth at 3 which means that when this kernel executes it will only perform two iterations of parallel breadth-first search starting at each start edge supplied as the inputs. With an edge factor of eight, the amount of parallelism even achievable is severely limited.

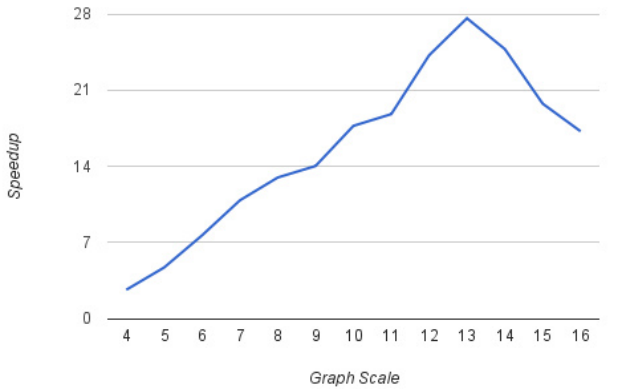

Figure 7. SSCA2 Kernel 4 Speedup

My implementation of betweenness centrality was tuned to permit the excess memory available when processing

smaller graphs to be used in performing multiple breadth-first search expansions simultaneously. The decreasing in speedup going beyond graph scale 13 is directly related to the graph being large enough to no longer allow more than one start point during the breadth-first search (Figure 7). At the peak, the speedup is nearly 28. To my knowledge, speedup of betweenness centrality for graphs of these scales has previously never been presented in literature.

I benchmarked the performance of my implementation of using the Davidson method (Algorithm 5) to calculate algebraic connectivity of graphs using the XMT simulator configured to simulate the 64-core FPGA prototype. For serial executions, I used an algorithm that only utilized the master TCU. For parallel executions, the algorithm used as many cores as necessary since the parallelism was expressed in terms of units of work rather than number of processors. I verified that the serial algorithm cycle count on the XMT simulator was similar (within 10 percent) of the cycle count when executed on an x86 processor.

I focused the benchmarks on examining acceleration of computing the Fiedler vector of small graphs in order to show that speedup is possible to achieve even on a small amount of data unlike related work which requires huge data sets in order to achieve their stated speedup values. I utilized the same RMAT scalable data generator used to produce graphs for the SSCA2 benchmarks to produce power law graphs containing just 16, 32, and 64 vertices. For graphs with 16 vertices, the simulator results showed a speedup on average of 7.3. For graphs with 32 vertices, the speedup increased to 14.6. And for 64 vertices the speedup was 27.2. Any individual parallel section of the algorithm has at most an amount of parallelism equal to the number of vertices in the graph processed. Thus for the first two graph sizes, the speedup is being achieved even when the cores are underutilized. At 64 vertices, the cores are being fully utilized for most of the matrix operations in the algorithm.

7. Related Work

In [2], the first parallel algorithm for betweenness centrality was introduced. Its implementation based on fine-grained parallelism is the basis for the reference implementation included in SSCA2 [20]. The authors later updated their algorithm [20] in a manner that reduced the amount of global synchronization of stacks required per iteration of breadth-first search. The proposed algorithm required maintaining a separate successor list per vertex to which access could be synchronized independently of the other successor lists. The overall memory requirement of the algorithm was increased but in exchange the algorithm showed a 2.31 speedup over their previous algorithm benchmarked on a 16-processor 500 MHz Cray XMT with 128 GB memory. Such a machine is one of a kind and very expensive.

In [15], the authors note that the common Brandes' algorithm approximation has a tendency to grossly overestimate the betweenness centrality of unimportant nodes. The effect is particularly profound on unimportant nodes which happen to be near one of the randomly selected starting points. For their applications in routing on road networks, minimizing the overestimation of unimportant nodes has significant value. Their algorithm requires having an a priori "start" and "end" vertex then during each iteration the algorithm performs one of $2n$ shortest path calculations in either the forward direction or the backward direction to another randomly selected node in the graph (hence the $2n$ possible shortest path calculations). The presented algorithm is embarrassingly parallel at a coarse-grain level, but shows difficulty for fine-grain parallelism. Their serial algorithm in fact is slower than Brandes algorithm on most data sets; however, they are primarily concerned with accuracy so they just restrict their run-time to that of being equal to Brandes algorithm on the same data and present accuracy results rather than speedup. [15]

In [27], the authors examine an algorithm most closely resembling the algorithm implemented in my work. The algorithm uses a global stack to store predecessor edges but does not attempt any coarse-grain parallelism choosing to only perform one breath-first search expansion at a time. The algorithm is benchmarked on two 2-ways, 4-cores SMPs (Intel Clovertown and AMD Barcelona). The authors do not provide the total memory available to these systems. Additionally they benchmark the original SSCA2 parallel implementation of betweenness centrality. Their results show speedup over the SSCA2 default algorithm similar to that achieved by the algorithm presented in [2] . Additionally, the authors provided analysis that their algorithm is work optimal CREW PRAM. [27]

Exact serial solutions to finding Fiedler vectors are often intractable for the Laplacian of large graphs as they rely on finding the eigenvalues and eigenvectors of the Laplacian. Thus approximation algorithms are often used. The fastest of these algorithms are often defined as multilevel approximations such as HSL_MC73 [17].

The Multilevel Recursive Spectral Bisection (MRBS) algorithm in one such multilevel algorithm which has been parallelized [5]. MRSB bisects a graph through recursively finding the Fiedler vector of successively smaller sections of the graph (the outputs from the previous level). As a multilevel technique, MRBS uses graph contraction to construct a set of smaller graphs which are representative of the graph as a whole. Parallel Multilevel RSB (PMRSB) distributes these subgraphs over the processing nodes which calculate the Fiedler vectors of each such subgraph. Then the results are combined and used to bisect the graph. The algorithm is then repeated over each bisected component. PMRSB was benchmarked in 1995 using the Cray T3D which has 256 processors operating at 150 MHz and was found to produce a speedup of 140 over the serial algorithm performed on a SGI Indigo workstation (100 MHz processor speed) when

processing a graph that has 262,620 vertices and 764,268 edges. When processing a much smaller graph (16,386 vertices and 49,152 edges–which is still huge compared to the graphs for which I was able to get even better speedups), the measured speedup was considerably lower at only 20 over the serial algorithm performed on the workstation. [5]

More recently, [25] demonstrated a parallel implementation of the Davidson method for generalized eigenproblems. Their approach is very similar to the one presented in this paper and shows speedups between approximately 20 and 35 over prior serial implementation when operating on a matrix containing 66,127 rows and columns. They compared their parallel implementation against serial algorithms contained in the Scalable Library for Eigenvalue Problem Computations (SLEPc). Their parallel implementation is now part of this library. The parallel implementation was benchmarked on a cluster of 55 bi-processor nodes with 2.80 GHz Pentium Xeon processors. The serial implementations were benchmarked using one node of this cluster. [25]

The parallel Trace Minimization (TraceMin) algorithm was able to show speedups on the order of 200 over the HSL_MC73 algorithm for computing Fiedler vectors [21]. The TraceMin algorithm is iterative like many other algorithms with its most time consuming part of each iteration being a part where it must solve a saddle-point problem. The results are impressive but the smallest graph considered by the authors contained 2,000,000 nodes being processed on a cluster of Intel Xeon CPUs with an unspecified number of nodes. [21]

8. Conclusion

As the purpose of this work was to evaluate the ability to achieve parallelism easily, no attention was given to comparing the benchmark results on the XMT prototype with results from a state-of-the-art serial processor. Future work could be performed to examine such an experiment which would evaluate the XMT against another architecture.

One goal of this work was to examine the effectiveness of XMTC at expressing parallelism in a way that is actually achieved at run time. In this regard, the work was successful. In all algorithms the majority of the development time spent was on the initial serial implementation. Due to recent improvements in the XMTC compiler's ability to exploit nested parallelism, even algorithms like parallel breadth-first search achieved excellent speedups (considering the graph sizes examined) compared to their serial counterparts. Prior implementations of breadth-first search required flattening of the parallelism in order to effectively utilize a many-core architecture, a process which is not necessarily intuitive to a programmer trained only in serial algorithms. Additionally, good speedup was observed when processing even small graphs using algorithms similar to those benchmarks on other parallel systems (clusters, etc.) which require large data sets to achieve good speedup. The ease of programming model presented by XMTC provides a clear path for a traditional serial programmer to begin writing parallel code. Additionally, improvements to the XMT toolchain continue to automate some of the tuning that a developer might have previously been expected to perform themselves. The source code for this project has been made available in a Google code repository. Information can be found on http://code.google.com/p/xmtc-benchmarks/ .

Acknowledgments

I want to take a moment to say thank you to the UMD XMT research team who provided me with feedback during the research process and allowed me access to the XMT prototype in order to perform my experiments. In particular, I would like to thank James Edwards, Alex Tzannes, and Uzi Vishkin.

References

[1] N.M.M. de Abreu. Old and New Results on Algebriac Connectivity of Graphs. Linear Algebra and its Applications 423 (1), pages 53-73, 2007.

[2] D. Bader and K. Madduri. Parallel Algorithms for Evaluating Centrality Indices in Real-World Networks. Proceedings of the 35th International Conference on Parallel Processing (ICPP), 2006.

[3] D. Bader. HPCS Scalable Synthetic Compact Applications 2 Graph Analysis. www.highproductivity.org/SSCABmks.htm, 2007.

[4] D. Barash. Spectral Decomposition for the Search and Analysis of RNA Secondary Structure. Journal of Computational Biology, Vol. 11 (Number 6, 2004), pages 1169-1174, 2004.

[5] S. Barnard. PMRSB: Parallel Multilevel Recursive Specral Bisection. Proceedings of the 1995 ACM/IEEE Conference on Supercomputing, 1995.

[6] L. Borges and S. Oliveira. A Parallel Davidson Type Algorithm for Several Eigenvalues. Journal of Computational Physics, 144 (1998), pages 763-770, 1998.

[7] U. Brandes. A Faster Algorithm for Betweenness Centrality. Journal of Mathematical Sociology, 25, pages 163-177, 2001.

[8] G. Caragea, B. Saybasili, X. Wen, and U. Vishkin. Performance Potential of an Easy-to-Program PRAM-on-Chip Prototype Versus State-of-the-Art Processor. Proceedings of the 21st ACM SPAA Symposium on Parallelism in Algorithm and Architectures, pages 163-165, 2009.

[9] G. Caragea, F. Keceli, A. Tzannes, and U. Vishkin General-Purpose vs. GPU: Comparison of Many-Cores on Irregular Workloads. Proceedings of HotPar 2010, 2010.

[10] G. Caragea and U. Vishkin. Better Speedups for Parallel Max-Flow. SPAA 2011, to appear.

[11] M. Crouzeix, B. Philippe, and M. Sadkane. The Davidson Method. SIAM Journal on Scientific Computing, 15, pages 66-76, 1994.

[12] E. Davidson. The Iterative Calculation of a Few of the Lowest Eigenvalues and Corresponding Eigenvectors of Large Real-

Symmetric Matrices. Journal of Computational Physics 17, pages 87-94, 1975.

[13] J. Edwards and U. Vishkin. An Evaluation of Biconnectivity Algorithms on Many-Core Processors Under Review.

[14] M. Fiedler. A Property of Eigenvectors of Nonnegative Symmetric Matrices and Its Application to Graph Theory. Czechoslavok Mathematics Journal, 25, pages 619-633, 1975.

[15] R. Geisberger, P. Sanders, and D. Schultes. Better Approximation of Betweenness Centrality. Proceedings of the 10th Workshop on Algorithm Engineering and Experiments (ALENEX 08). SIAM, 2008.

[16] M. Holzrichter and S. Oliveira. A Graph Based Method for Generating the Fiedler Vector of Irregular Problems. Parallel and Distributed Processing, Lecture Notes in Computer Science, Volume 1586/1999, pages 978-985, 1999.

[17] Y. Hu and J. Scott. HSL_MC73: A Faster Multilevel Fiedler and Profile Reduction Code. RAL-TR-2003-36, Numerical Analysis Group, Computational Science and Engineering Department, Rutherford Appleton Laboratory, 2003.

[18] A. Kaveh, H. Rahimi Bondarabady. Bisection for Parallel Computing Using Ritz and Fiedler Vectors. Acta Mechanica 167, pages 131-144, 2004.

[19] F. Keceli, A. Tzannes, G. Caragea, R. Barua and U. Vishkin. Toolchain for Programming, Simulating and Studying the XMT Many-Core Architecture. Proc. 16th International Workshop on High-Level Parallel Programming Models and Supportive Environments (HIPS 2011), in conjunction with IPDPS, Anchorage, Alaska, May 20, 2011.

[20] K. Madduri, D. Ediger, K. Jiang, D. Bader, and D. Chavarria-Miranda. A Faster Parallel Algorithm and Efficient Multithreaded Implementations for Evaluating Betweenness Centrality on Massive Datasets. 2009 IEEE International Symposium on Parallel and Distributed Processing, pages 1-8, 2009.

[21] M. Manguoglu. A Highly Efficient Parallel Algorithm for Computing the Fiedler Vector. Submitted to ACM TOMS 2010, arXiv:1003.3689v1.

[22] B. Mohar. The Laplacian Spectrum of Graphs. Graph Theory, Combinatorics, and Applications, Vol. 2 (1991), pages 871-898, 1991.

[23] D. P. O'Leary and G. W. Stewart. Computing the Eigenvalues and Eigenvectors of Symmetric Arrowhead Matrices. Journal of Computational Physics, 90 (1990), pages 497-505, 1990.

[24] E. Romero and J. Roman. A Parallel Implementation of the Trace Minimization Eigensolver. High Performance Computing for Computational Science, VECPAR 2008, pages 255-268, 2008.

[25] E. Romero and J. Roman. A Parallel Implementation of the Davidson Method for Generalized Eigenproblems. Advances in Parallel Computing, Vol. 19, pages 133-140, 2010.

[26] A. B. Saybasili, A. Tzannes, B.R Brooks and U. Vishkin. Highly parallel multi-dimensional fast Fourier transform on fine- and course-grained many-core approaches. Proc. 21st Conference on Parallel and Distributed Computing Systems (PDCS), Cambridge, MA, November 2-4, 2009

[27] G. Tan, D. Tu, and N. Sun. A Parallel Algorithm for Computing Betweenness Centrality. International Conference on Parallel Processing (2009), pages 340-347, 2009.

[28] S. Torbert, U. Vishkin, R. Tzur and D. Ellison. Is teaching parallel algorithmic thinking to high-school student possible? One teacher's experience. Proceedings of 41st ACM Technical Symposium on Computer Science Education (SIGCSE), 2010.

[29] U. Vishkin, G. Caragea and B. Lee. Models for Advancing PRAM and Other Algorithms into Parallel Programs for a PRAM-On-Chip Platform. Chapter 5, Handbook on Parallel Computing: Models, Algorithms, and Applications, Editors: S. Rajasekaran and J. Reif, Chapman and Hall/CRC Press, 2008.

[30] U. Vishkin. Using Simple Abstraction to Reinvent Computing for Parallelism. Communications of the ACM 54, 1 (January 2011), pages 75-78, 2011.

[31] X. Wen and U. Vishkin. PRAM-On-Chip: First Commitment to Silicon. Brief announcement in Proc. 19th ACM Symposium on Parallel Algorithms and Architectures (SPAA), pages 301-302, June 9-11, 2007.

[32] X. Wen and U. Vishkin. FPGA-Based Prototype of a PRAM-On-Chip Processor. ACM Computing Frontiers, 2008.

Algorithm 4 SSCA2 Kernel 4

Input: $G(V, E)$

Output: $BC[1..n]$, where $BC[v]$ gives the centrality score for vertex v.

1: **for all** $v \in V$ **in parallel do**
2: $BC[v] \leftarrow 0$
3: **end for**
4: **for all** $s \in V$ **do**
5: // I. Initialization
6: $\sigma[s] \leftarrow 1$
7: $d[s] \leftarrow 0$
8: **for all** $t \in V$ **in parallel do**
9: $\sigma[t] \leftarrow 0$
10: $d[t] \leftarrow -1$
11: **end for**
12: $phase \leftarrow 0$
13: $P[phase] \leftarrow$empty list
14: $S[phase] \leftarrow$empty stack
15: push $s \rightarrow S[phase]$
16: $count \leftarrow 1$
17: // II. Graph traversal
18: **while** $count > 0$ **do**
19: $count \leftarrow 0$
20: **for all** $v \in S[phase]$ **in parallel do**
21: **for** each neighbor w of v **in parallel do**
22: **if** $d[w] < 0$ **then**
23: $d[w] \leftarrow d[v] + 1$
24: $count \leftarrow count + 1$
25: push $w \rightarrow S[phase + 1]$
26: **end if**
27: **if** $d[w] = d[v] + 1$ **then**
28: $\sigma[w] \leftarrow \sigma[w] + \sigma[v]$
29: append $< v, w > \rightarrow P[phase]$
30: **end if**
31: **end for**
32: **end for**
33: $phase \leftarrow phase + 1$
34: **end while**
35: $phase \leftarrow phase - 1$
36: // III. Dependency accumlation by back-progation
37: $\delta[t] \leftarrow 0 \; \forall \; t \in V$
38: **while** $phase > 0$ **do**
39: **for all** $< v, w > \in P[phase]$ **in parallel do**
40: $\delta[v] \leftarrow \delta[v] + \frac{\sigma[v]}{\sigma[w]}(1 + \delta[w])$
41: **end for**
42: $phase \leftarrow phase - 1$
43: **end while**
44: **for all** $v \in V$ **do**
45: $BC[v] \leftarrow BC[v] + \delta[v]$
46: **end for**
47: **end for**