SAND2014-19275C

High-performance Graph Analytics on Manycore Processors

George M. Slota*, Sivasankaran Rajamanickam†, and Kamesh Madduri*
*Computer Science and Engineering, The Pennsylvania State University. Email: gslota@psu.edu, madduri@cse.psu.edu
†Scalable Algorithms Department, Sandia National Laboratories, Email: srajama@sandia.gov

***Abstract*—The divergence in the computer architecture landscape has resulted in different architectures being considered mainstream at the same time. For application and algorithm developers, a dilemma arises when one must focus on using underlying architectural features to extract the best performance on each of these architectures, while writing portable code at the same time. We focus on this problem with graph analytics as our target application domain. In this paper, we present an abstraction-based methodology for performance-portable graph algorithm design on manycore architectures. We demonstrate our approach by systematically optimizing algorithms for the problems of breadth-first search, color propagation, and strongly connected components. We use Kokkos, a manycore library and programming model, for prototyping our algorithms. Our portable implementation of the strongly connected components algorithm on the NVIDIA Tesla K40M is up to 3.25× faster than a state-of-the-art parallel CPU implementation on a dual-socket Sandy Bridge compute node.**

***Index Terms*—graph computations; BFS; Color propagation; GPUs; parallel performance; portability**

I. Introduction

Our work attempts to answer the following questions in the context of graph computations and GPUs:

(a) Can we identify frequently-used optimization strategies from the large and growing collection of tuned parallel graph computation implementations (e.g., [2], [5], [9], [23], [36]), and create a structured methodology for designing new parallel algorithms? If one were to build a new framework for high-performance domain-specific graph computations, what would be the key optimization strategies to consider, and best practices to follow?
(b) In addition to a *parallel-for*, data-parallel scans, reductions, and sorting methods, what are some common abstractions used to design parallel graph algorithms?
(c) Is it possible to develop performance-portable implementations of graph algorithms using advanced parallel programming libraries and frameworks with the optimizations and abstractions identified above?

We begin by observing that several recent graph algorithms and their efficient implementations follow the loop nest structure shown in Algorithm 1. This is particularly true for computations on large, sparse, and static graphs. The first point to note in Algorithm 1 is that the listing uses only simple, array-based data structures. Current state-of-the-art parallel implementations for several graph problems use array-based stacks, queues, and priority queues, as these structures are more amenable to applying data-parallel operations such as scans and reductions. Implementations of this general template differ in terms of graph representation, data structure access patterns, number of iterations of the outer loop, graph topology-based heuristics to reduce total work, synchronization overhead, etc. Intra- and inter-iteration dependencies hinder automatic compiler-based loop transformations such as unrolling, coalescing, collapse, and fusion.

Algorithm 1 A template followed by several serial and parallel graph algorithms operating on a sparse graph $G(V, E)$. $m = |E|$, $n = |V|$, and $m = O(n \log n)$.

```
Initialize temp/result arrays A_t[1..n], 1 ≤ t ≤ l.        ▷ l = O(1)
Initialize S_1[1..n].
for i = 1 to niter do                                      ▷ niter = O(log n)
    Initialize S_{i+1}[1..n].                              ▷ Σ_i |S_i| = O(m)
    for j = 1 to |S_i| do                                  ▷ |S_i| = O(n)
        u ← S_i[j]
        Read/update A_t[u], 1 ≤ t ≤ l.
        for k = 1 to |E[u]| do                             ▷ |E[u]| = O(n)
            v ← E[u][k]
            Read/update A_t[v].
            Read/update S_{i+1}.
        Read/update A_t[u].
```

Consider level-synchronous parallelizations of Breadth-First Search (BFS). $niter$, the number of outer-loop iterations, is bounded by the graph diameter. The arrays S_i correspond to the vertices in the current frontier, and the adjacencies of these vertices can be visited in parallel. Arrays of size n (A_t) are used to store parent information, whether a vertex has been previously visited or not, and distance from the source vertex. Finally, there is a barrier synchronization before every iteration of the outer loop. For low-diameter (diameter is $O(\log n)$) graphs, the overhead of barrier synchronization is insignificant in comparison to the work performed in the inner loops. The arrays S_i store vertices in an arbitrary order for BFS. For other algorithms, such as ones for single-source shortest paths (SSSP), the ordering of vertices in S_i may be important to reduce the outer-loop iteration count. The direction-optimizing heuristic in a recent BFS algorithm [3] and the push-pull optimization in a recent parallel SSSP algorithm [6] can be viewed as work-reducing heuristics to switch between alternate representations of S_i in the inner loop.

The label propagation community detection heuristic [29] and the PuLP graph partitioning strategy [34] also fit within this general template. They differ in the access patterns of

the temporary arrays and the result arrays in the inner loops. Algorithms that are similar to level-synchronous BFS, such as betweenness centrality [22], approximate diameter [8], strongly connected components [33], and biconnected components [32], also follow a similar structure. In fact, several PRAM graph algorithms can be viewed as instances of this template, and they would result in polylogarithmic parallel time algorithms (assuming low-diameter graphs and/or a $O(\log n)$ bound for the outer-loop iterations). This template is not restricted to shared-memory algorithms. Distributed-memory approaches for K-core decomposition also use a label propagation-like strategy [25], and implementations differ in the number of outer-loop iterations. Finally, open-source software packages providing state-of-the-art parallel implementations of graph algorithms, such as Parallel Boost graph library [15], MTGL [4], Galois [19], SNAP [21], PowerGraph [14], Ligra [31], NetworKit [35], etc., include several programs that fall under the Algorithm 1 template.

So we hypothesize that the first step towards creating an efficient parallel implementation of a graph algorithm would be to recast it such that it fits the general template shown in Algorithm 1. The focus of the current work is efficient graph analytics on manycore platforms such as NVIDIA and AMD GPUs, and the Intel Xeon Phi MIC coprocessor. We do not want to reinvent the wheel for data-parallel subroutine implementations and parallel-for support. Hence, we use an emerging node-level library and programming model called Kokkos [11], that lets us write code that is portable to GPUs, Intel Xeon Phi, as well as Intel and AMD x86 CPUs. In Section II, we discuss key Kokkos features that enable us to quickly develop and compare alternate implementations.

As the next step, we develop several manycore implementations for the graph problems of BFS, color propagation, and strongly-connected components (SCC) expressing them in the template shown in Algorithm 1. The inner loop nests of BFS and color propagation have several differences, and so we explore both of these problems. The general manycore SCC algorithm is based on our prior multicore SCC approach called Multistep [33]. Use of Kokkos lets us develop several alternatives for each problem and conduct a methodical evaluation of optimizations.

In Section III, we present the third step, the key optimizations that are critical to manycore performance and portability. We primarily evaluate several tuned loop transformation strategies for the inner loop nest, and we customize these strategies for our use case of small-world graph analytics. These loop transformations, in essence, improve load balance and reduce irregular memory accesses. Our proposed strategies are similar to a compiler-based loop collapse [1], [30]. However, compilers cannot automatically do this because of loop-carried dependencies. To the best of our knowledge, this is the first work to explore the loop transformations and GPU optimizations for SCC and color propagation problems. We arrive at this portable and performant manycore implementation of SCC using our algorithmic template, a Kokkos based implementation, and architecture-aware optimizations.

The main observations from our empirical performance evaluation (see Section IV) are as follows:

- Our new loop collapse strategy, termed *Local Manhattan collapse*, is very effective on GPUs and consistently results in the highest-performing variant for several problems.
- A GPU SCC implementation using the Local Manhattan collapse strategy demonstrates up to a 3.25× speedup relative to a state-of-the-art parallel CPU implementation running on a dual-socket compute node.
- We find our GPU BFS implementation averages 1.74 GTEPS across a suite of 12 test graphs, comparable to the current state-of-the-art, without any explicit BFS-specific tuning.

II. Portable Graph Algorithms for Manycore

A. The Kokkos Programming Model

The Kokkos library [11] was originally developed as a back-end for providing portable performance for scientific computing frameworks, but has since been extended to a more general-purpose library for parallel execution. The two primary capabilities of Kokkos include polymorphic multidimensional arrays optimized for varying data access patterns/layouts in different architectures and thread parallel execution that allows for fine-grained data parallelism on manycore devices.

The parallel execution model follows a dispatch model, where a single master CPU thread divides some N units of work to be processed on GPU. Each unit of work is executed by a single *thread* or *thread team*. On GPU, a thread team is comprised of multiple warps each executing on the same multiprocessor. This team of threads operates in a data parallel SIMT fashion, and is able to "communicate" via shared memory. In addition to optimizing the data layout in different devices, Kokkos also provides us the option to use the hierarchy of memory in manycore devices, such as thread block shared memory and texture cache. We use these features of Kokkos for the appropriate data structures. We also use the fast atomic operations and "thread team level" scan and reduction operations to synchronize between different threads in a thread block. One of the key design decision we made, to use low-level simple array-based data structures, helps us when using Kokkos, as the layout of these simple arrays is then optimized by Kokkos in different devices (CPUs, GPUs and Phis), different types of memory (e.g., shared memory) and different access patterns (e.g., coalesced access). Use of custom data structures that are optimized for any single architecture would have prevented Kokkos level optimizations, in turn affecting portability.

B. BFS

BFS is one of the most widely used and basic graph subroutines, appearing in a vast number of more complex graph analytics. The goal of BFS is commonly to determine from a given root either reachability status, distance, or BFS tree parent-child relationships for all or some subset of vertices in a graph. On each iteration $i \in niter$ of the algorithm, the status for all vertices that are distance i away from the root

is updated. Described in terms of Algorithm 1, we would first initialize S_1 to contain only the root. To determine distances, our result array A would be initialized to -1 for all vertices except for the root, which would be initialized to 0. Each iteration will examine all edges of all vertices in S_i. When a vertex v is encountered such that $A[v] < 0$, we update $A[v]$ to 1 and place v into S_{i+1}.

There has been a lot of recent work focused on optimizing both CPU and manycore-based implementations of BFS [3], [7], [12], [16], [17], [23]. We don't explicitly consider fully optimizing BFS itself through our framework, but rather show how close we can get to state-of-the-art traversal rates by only considering simple techniques that affect per-unit-work assignments and which are applicable to a much broader class of algorithms.

C. Color Propagation

Color propagation is an iterative procedure that is useful for many different graph connectivity problems [27], [31]–[33]. An overview of the general algorithm is given by Algorithm 2. Note how it also follows the Algorithm 1 template, where S_i is our current queue and the result array A can be considered as the current coloring of the graph.

Algorithm 2 Color Propagation pseudocode.

```
A[1..n] ← 1..n                          ▷ Set A[i] = i, 1 ≤ i ≤ n
S_1[1..n] ← 1..n                        ▷ Set S_1[i] = i, 1 ≤ i ≤ n
i ← 1
while S_i ≠ ∅ do
    S_{i+1} ← ∅
    for j = 1 to |S_i| do
        u ← S_i[j]
        for k = 1 to |E[u]| do
            v ← E[u][k]
            if A[u] > A[v] then
                A[v] ← A[u]
                Add u to S_{i+1}
    i ← i + 1
```

We initialize A to be unique vertex identifiers and S_1 as all vertices in the graph. We then examine all edges, and when there exists a source vertex that has a higher color than one of its neighbors, that vertex propagates its color to the neighbor. The next work set S_{i+1} is comprised of vertices that have had their color altered. This process continues iteratively until no further propagations occur. As with BFS, we implement color propagation in a straightforward manner within our general framework. Outside of initializations and the work performed on the innermost loop, there are few differences between the BFS and color propagation codes.

D. Strongly Connected Components

The problem of computing strongly connected components (SCCs) in large directed small-world graphs is a common analytic for social networks [24] and a preprocessing step in scientific computing (among other uses) [28]. Using either BFS or coloring, straightforward parallel strongly connected component decomposition algorithms can be implemented [18], [27]. Combining both subroutines into an efficient Multistep procedure can result in considerable speedup for small-world graphs [33].

We use the BFS and color propagation subroutines implemented in our framework to perform graph SCC decomposition via the Multistep procedure (we refer the reader to [33] for a more detailed description). Once again, outside of a few changes to initializations and the very innermost loops, few alterations need to occur to the original BFS and color propagation codes for the SCC problem.

III. Optimization Methodologies

In this section, we describe the optimization techniques used to achieve scalable performance on manycore architectures. These techniques are general enough for any algorithm that fits the template described in Algorithm 1. Furthermore, the optimizations are general enough for architectures that share similar characteristics, such as a very high core count, hierarchical memories, and small amounts of memory per thread. These characteristics of present day GPUs is expected to hold or become increasingly important in future manycore architectures. As a result, the optimizations described here are critical for scalable algorithms on current GPUs and future manycore architectures.

A. Thread teams, local synchronization, shared and global memory

Current GPUs are organized as a number of streaming multiprocessors (for instance, 16 in NVIDIA Maxwell GM204), each with a number of smaller cores (e.g., 128 CUDA cores in GM204). The number of threads that can be scheduled in a single streaming multiprocessor of a GPU can be up to 2048. The number of warps per streaming multiprocessor is 48-64, and the number of thread blocks is 8-16, depending on the microarchitecture. A similar hierarchy is also seen in the multiple hyperthreads per core in a Xeon Phi coprocessors, along with NUMA effects due to placement of cores near different memory regions. This results in multiple levels of parallelism that algorithm developers need to design for. The programming model in Kokkos abstracts this to a *thread team*, where a thread team corresponds to a thread block on GPU. In order to effectively utilize a streaming multiprocessor (SM), it is crucial to be able to schedule multiple thread blocks in each SM. Within each thread block, there is enough concurrency for thousands of threads, so that multiple warps can be kept busy at the same time. All of our algorithms use the thread teams concept to synchronize locally and utilize shared memory to communicate within a team when necessary.

The number of thread blocks that can be scheduled concurrently in a single SM is determined by the amount of shared memory used by each thread block (or a thread team). As all the threads in a team use the shared memory to synchronize among themselves before synchronizing to the global memory, the amount of shared memory used is an important resource. Increasing its size would reduce the number of writes to global memory by doing more local synchronizations, but it

would also decrease the number of concurrent thread blocks that can be scheduled. Our approach balances shared memory usage with the parallelism available within each thread block. Finally, it is important that reads and writes to global memory are coalesced. Essentially, we want reads and writes for a single warp to be to be performed at neighboring global memory addresses to reduce total memory bandwidth, improve cache utilization, and ensure that threads in the warp are not idle waiting for the memory request of a single thread.

B. Hierarchical Exploration to Improve SM Utilization

A common optimization technique for algorithms dealing with irregular graph structure is special handling of the fringe cases, i.e., vertices with degree much larger than the average [16], [23]. This can be done at the granularity of a single level or through considering multiple classes of vertices in a hierarchy. In the GPU context, this might translate to a thread block working together to explore the edges of a vertex with an out-degree greater than the number of threads in the thread block, while a warp would explore vertices with an out-degree greater than the number of threads in a warp, but smaller than the number of threads in a block. Smaller vertices would be handled by individual threads. This general hierarchical technique has been previously used for irregular graph problems, referred to as the *deferring outliers* [16] and the *CTA+Warp+Scan* [9], [23] approaches.

Algorithm 3 Hierarchical Expansion.

```
Initialize A and S_1
for i = 1 to niter do
    Initialize S_{i+1}[1..n].
    for all Thread Teams do                          ▷ Team-level parallelism
        Retrieve subset V_T from S_i
        for j = 1 to |V_T| do                        ▷ Thread-level parallelism
            v ← V_T[j]
            if |E[v]| > |T| then
                Add v to Q_T                         ▷ Team-shared Queue
            else if |E[v]| > |W| then
                Add v to Q_W                         ▷ Warp-shared Queue
            else
                Add v to Q_t                         ▷ Thread-owned Queue
        Team-level synchronization
        for j = 1 to |Q_T| do
            v ← Q_T[j]
            for k = 1 to |E[v]| do                   ▷ Thread-level parallelism
                u ← E[v][k]
                Read/update A[u]
        Warp-level synchronization
        for j = 1 to |Q_W| do                        ▷ Warp-level parallelism
            v ← Q_W[j]
            for k = 1 to |E[v]| do                   ▷ Thread-level parallelism
                u ← E[v][k]
                Read/update A[u]
        for j = 1 to |Q_t| do                        ▷ Serial expansion by thread
            v ← Q_t[j]
            for k = 1 to |E[v]| do
                u ← E[v][k]
                Read/update A[u]
        Team-level synchronization
        Update S_{i+1}
```

Our implementation of this technique, which we term as *Hierarchical Expansion*, is given by Algorithm 3. In the Kokkos model, we consider parallelism at three hierarchies: team-level, warp-level, and thread-level. For each iteration of our algorithm, we remove a chunk of vertices V_S from the input work set S_i and pass it to a Kokkos thread team T. For good team-level work balance and multiprocessor utilization, the size of V_S is usually within a small factor of the size of T. The threads in each team work to process their input set, placing the high-degree vertices they encounter into a team-shared queue (when the degree is greater than the size of the thread team $|T|$) or warp-shared queue (when the degree is smaller than the size of a thread team, but larger than the size of a warp $|W|$). Smaller vertices get placed into a small thread-owned buffer Q_t for later serial expansion.

The vertices in the team-level queue Q_T are collectively expanded by all threads, with potential updates to S_{i+1} kept in team-level shared memory. Once the Q_T queue is exhausted, the warp queue Q_W is examined. Each warp removes a vertex from the queue, and cooperatively expands its adjacencies. Finally, each individual thread serially expands the vertices in its buffer. Once all work is exhausted, the team collectively pushes their updates to the next iteration's work set S_{i+1}. We use team-level scans and reductions whenever possible to minimize global synchronizations.

The primary benefit to this type of approach is that it allows fine-grained warp utilization by limiting the serial expansion of high degree vertices by a single thread or warp. This leads to better load balance at the thread and warp level. This approach also allows the use of shared memory to create new queues for the next iteration of an irregular graph problem, reducing the number of global synchronizations required. However, as vertices are assigned to a single team statically, there can still be some imbalance at the highest level. All teams might finish their work long before the team owning a highly-skewed vertex completes, delaying the start of the next iterations and vastly under-utilizing available processing resources.

C. Loop collapse For Better Load Balance

As shown by the template in Algorithm 1, many graph algorithms follow the pattern of two nested loops, where the outer loop is over the vertices and the nested inner loop is over the edges. In typical graph analytic algorithms, these two loops are not perfectly nested, as the vertex contents of the outer loop determine the start and end indices for the edges examined of the inner loop. There might be additional operations within the outer loop, such as changing the properties of vertices, adding vertices to the next queue, etc. While perfectly nested loops are great candidates for compiler-based optimizations, loops containing these other operations cannot be automatically optimized by compilers.

The importance of collapsing these loops increases when both loops are parallelizable and when the work in different outer loop iterations is heavily unbalanced. In graph analytic algorithms on graphs with skewed degree distributions, when the work in the outer loop varies based on the degree of the

vertices, collapsing the inner loop is critical. When there are few threads, like in the CPU, a simple dynamic scheduling runtime can alleviate the problem [33]. However, it is hard to scale this approach to the thousands of threads in manycore devices. In our framework, we do the optimization a compiler might do, and collapse the two loops over vertices and edges into a single loop over all possible edges.

We employ the *Manhattan collapse* [30], where a prefix-sum operation, easily parallelizable on GPUs with a scan-based procedure, is used to compute the bounds of each outer loop iteration. With the results of the prefix sums, a binary search is then used to compute the indices of the original inner-loop and outer-loop within the collapsed loop (given by Algorithm 4). The overhead associated with reverse-engineering the vertex information is offset by the good load balance achieved by each thread. This general approach has been explored before by Merrill et al. for GPUs in the context of BFS [23] and by Davidson et al. for SSSP [9]. As with the work of Davidson et al., we consider two forms of the Manhattan collapse, implementing it at both the global and local level.

1) Local Manhattan collapse: For our local implementation, we do not require any additional global storage, apart from the queues and work arrays updated in the algorithm. An overview of this approach is given by Algorithm 5. We statically partition our work set S_i on a per-vertex basis and pass each partition V_T to our thread teams. The thread team computes prefix sums P over V_T based on out-degree. P is stored in shared memory. The final prefix sum in P is the sum of edges for V_T, and therefore proportional to the total work that the team needs to do. We can then equally distribute this work among all the threads in the team.

Algorithm 4 Psueudocode for the HIGHESTLESSTHAN subroutine, used in both local and global Manhattan loop collapse.

```
procedure HIGHESTLESSTHAN(P, val)
    found ← 0
    bound_low ← 0
    bound_high ← |P| − 1
    while found = 0 do
        index = (bound_high + bound_low)/2
        if P(index) ≤ val and P(index + 1) > val then
            found = 1
        else if P(index) < val then
            bound_low = index
        else
            bound_high = index
    return index
```

To get a specific edge based on a per-thread work assignment j, the source vertex is determined by examining the prefix sums array, and finding the index k that corresponds to a value in P greater than or equal to j, and less than the value at the next highest index (Algorithm 4). The specific out-edge u from the source vertex can be found by using the difference between the work assignment and the value at the found index in the prefix sums. With the (u, v) pair, the thread can now perform its assigned work.

Algorithm 5 Local Manhattan loop collapse.

```
Initialize A_t and S_1
for i = 1 to niter do
    Initialize S_{i+1}[1..n]
    for all Thread Teams do                    ▷ Team-level parallelism
        Retrieve subset V_T from S_i
        P ← PrefixSums(V_T)
        Max ← max(P)
        for j = 1 to Max do                    ▷ Thread-level parallelism
            k ← HIGHESTLESSTHAN(P, j)
            u ← V_T[k]
            v ← E[u][j − P[k]]
            Read/update A_t[u] and A_t[v]
        Team-level synchronization
        Update S_{i+1}
```

Algorithm 6 Global Manhattan loop collapse.

```
Initialize A_t and S_1
Initialize P_1
for i = 1 to niter do
    Initialize S_{i+1}[1..n]
    for all Thread Teams do
        Retrieve subset j_T to j_{T+1} of max(P_i)
        for j = j_T to j_{T+1} do              ▷ Thread-level parallelism
            k ← HIGHESTLESSTHAN(P_i, j)
            u ← V_T[k]
            v ← E[u][j − P_i[k]]
            Read/update A_t[u] and A_t[v]
        Team-level synchronization
        Update S_{i+1} and P_{i+1}
```

The primary benefit of the Local Manhattan collapse is that it leads to full warp and thread utilization of processor resources. When the cost of looking up a work assignment is low compared to the work that needs to be done, this approach is highly beneficial. As with Hierarchical Exploration, a major drawback to doing the Local Manhattan collapse is that a vertex is still assigned to a single team, which might lead to work imbalances for highly skewed graphs.

2) Global Manhattan collapse: To alleviate any potential work imbalance issues, we implemented a fully-partitioned approach, where the prefix sums for the current iteration are computed on the previous iteration as updates were pushed to the next-iteration work set S_{i+1}. By doing this, we can statically distribute an equal number of *edges* to each team instead of vertices. As can be seen in Algorithm 6, the approach closely follows our local method.

The primary difference lies in the prefix sum array P_i, which must be globally stored and synchronously updated. To minimize data transfer requirements, each thread team can determine its start and end offsets in P_i and do a single transfer of the needed portion to shared memory. Additionally, pushes to P_{i+1} and S_{i+1} can also be coalesced, with only a single atomic update required per team. Because each team needs to determine the offset to start writing to P_{i+1}, as well as the current running sum, we package both these values into a single atomically-updated 64-bit `long int` and perform an atomic fetch-and-add on the current global value.

Ideally, the Global Manhattan collapse should offer the best

work partitioning among thread teams and fastest execution times for a given algorithm. However, as we will show in our results, there are other key factors that hurt the performance of the Global Manhattan collapse relative to the Local method. For simple graph algorithms with minimal work per edge, the cost of reading and writing to an additional global array is relatively high. Amortizing this startup and end cost by increasing work per team is not necessarily a good solution, as we would ideally like to have a as-large-as-is-practical number of teams to hide the memory access latencies inherent to the rest of the implemented algorithm. Further, on graphs with a relatively-consistent degree, or a modest number of outliers, this method offers no additional benefit in terms of equal per-team work distribution, relative to the Local collapse. Finally, the maximal degree of many real-world graphs is bounded by $O(\sqrt{n})$. As long as the maximal degree is less than $O(n)$ and there are relatively few outliers, the level of fine-grained global work distribution offered by the global collapse is likely not necessary.

IV. Performance Analysis and Discussion

A. Experimental Setup

We evaluate our algorithms on single nodes of three clusters, the *Shannon* and *Compton* systems at Sandia, and the NSF *Blue Waters* system at NCSA. A Shannon node has two Intel Xeon E5-2670 Sandy Bridge-EP processors with 128 GB main memory and an NVIDIA Tesla K40M GPU. The K40M GPU has 12 GB DDR5 memory, 2880 cores, and a peak memory bandwidth of 288 GB/s. Each GPU-enabled compute node of Blue Waters has one AMD 6276 Interlagos processor with 32 GB main memory and an NVIDIA Tesla K20X GPU. The K20X GPU has 6 GB DDR5 memory, 2688 cores, and a peak memory bandwidth of 250 GB/s. We use Compton nodes for running our Kokkos and OpenMP implementations on Intel Xeon Phi MIC coprocessors. Compton nodes are identical to Shannon nodes but only have 64 GB memory and house MICs containing 57 cores at 1.1 GHz with 6 GB memory. In all cases, the version of Kokkos used in our evaluation came from release 11.10.1 of Trilinos, we used `icc` and `nvcc` for compilation along with the `-O3` optimization option.

We used several real small-world directed graphs that range in size from 5.1 million to 936 million edges for testing. These are listed in Table I. The graphs are from the SNAP database [21], the Koblenz Network Collection [20], and the University of Florida Sparse Matrix Collection [10]. We selected these graphs to represent a wide mix of graph sizes and topologies. Graph topology also has a strong influence on the performance of BFS and color propagation, while the number of total and nontrivial SCCs, as well as the size of the largest SCC, play an important role in determining performance of the SCC algorithm.

We report BFS and color propagation performance in terms of the Giga Traversed Edges per Second (GTEPS) metric, which normalizes running time to the total number of edges accessed (in billions). Note that our input graphs are directed and most of them have a large SCC. For each BFS execution, we track the total number of edges visited. Similarly, we count the number of vertex color and edge updates to determine overall performance for color propagation. We also run multiple iterations of both algorithms on all the target systems to reduce any variation in running time. In order to be consistent with BFS and color propagation results, we normalize SCC performance also by the number of edges and report an overall GEPS (Giga Edges per second) rate for each graph.

TABLE I
Information about test networks. Columns are # vertices, # edges, average and max. degree, # of SCCs, # number of nontrivial SCCs, and size of the largest SCC.

Network	n	m	Degree		(S)CCs		
			avg	max	Count	nontriv.	max
Google	875 K	5.1 M	5.8	5 K	370 K	12 K	410 K
Flickr	820 K	9.8 M	12	10 K	277 K	7.3 K	530 K
XyceTest	1.9 M	8.2 M	4.2	250	400 K	2.0 K	1.5 M
LiveJournal	4.8 M	69 M	14	20 K	970 K	23 K	3.8 M
RMAT2M	2.0 M	128 M	64	8.7 K	1 M	1	1.0 M
GNP2M	2.0 M	128 M	64	95	1	1	2.0 M
Indochina	7.4 M	194 M	26	180 K	1.6 M	40 K	3.8 M
DBpedia	67 M	258 M	3.9	650 K	55 M	2.9 M	8.9 M
HV15R	2.0 M	283 M	140	170 K	24 K	15	120 K
uk-2002	18 M	398 M	16	4 K	3.7 M	70 K	12 M
WikiLinks	26 M	600 M	23	400 K	6.6 M	60 K	19 M
uk-2005	39 M	936 M	24	130 K	5.8 M	223 K	26 M

For the Kokkos GPU approach, we fix thread queue sizes at 16, work chunks at 256 vertices per thread team for Hierarchical and Local Manhattan, and work chunks at 2048 per thread team for Baseline (vertex chunks) and Global Manhattan (edge chunks). For Xeon Phi and CPU, we used larger queues of size 1024 and work chunks of 2048. These values were selected for exhibiting the fastest performance across a range of values on our test suite. We will fully explore the performance impact of these algorithmic parameters in future work.

B. BFS performance

Figure 1 gives the performance rates of the Kokkos-based BFS implementations on the Tesla K40M GPU. The baseline rate in the figure corresponds to performance with a vertex-based partitioning of the frontier array among thread teams. It is not a trivial implementation and our speedup numbers are conservative in that sense. We see consistent and significant speedups with three loop collapse strategies (H: hierarchical, MG: Manhattan collapse using global memory, ML: Manhattan collapse using GPU shared memory). Using H, MG, and ML, the speedups (geometric mean) over baseline are 1.82×, 1.82×, 3.85×, respectively, for the twelve graphs considered. The graphs are ordered in the figure in increasing order of average vertex degree, from DBpedia (3.9) to HV15R (140). Apart from a couple of anomalies, there is a reasonable correlation between average vertex degree in the graph and the BFS performance of the best variant (ML). Prior GPU graph algorithms work [17] has also made similar observations. However, one striking aspect is that the tuned variant can be more than an order-of-magnitude faster than the baseline, as we note for the Flickr graph. This is a bit surprising, given

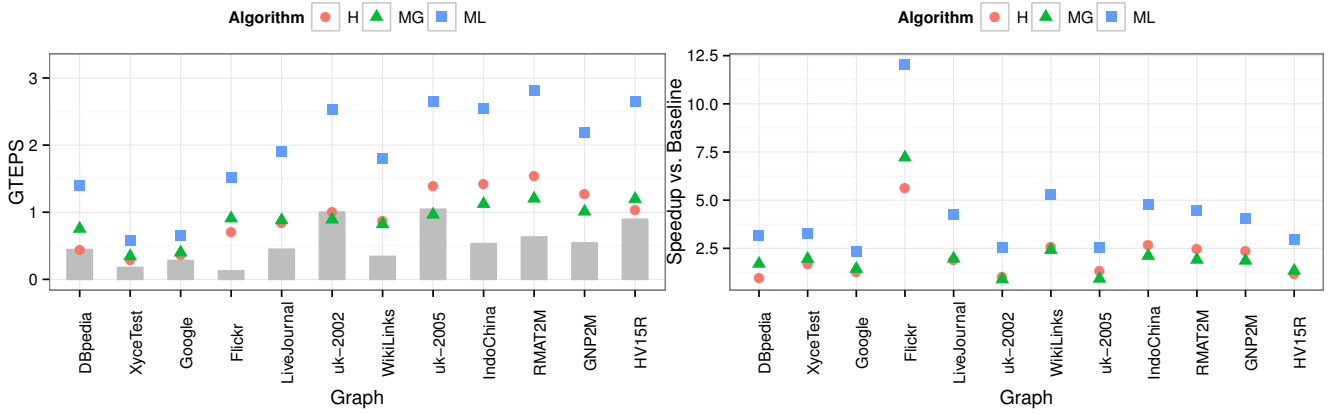

Fig. 1. BFS performance on Tesla K40M using Manhattan-Local (ML), Manhattan-Global (MG), and Hierarchical (H) loop collapse strategies.

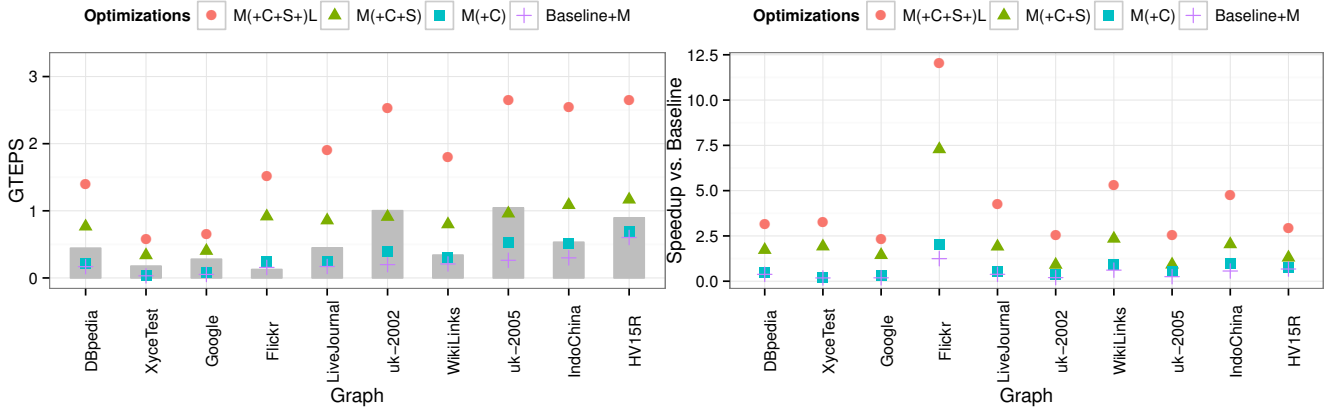

Fig. 2. Impact of various optim. strategies (Manhattan collapse (M), coalescing (C), team-scan (S), and local primitives (L)) on Tesla K40M BFS performance.

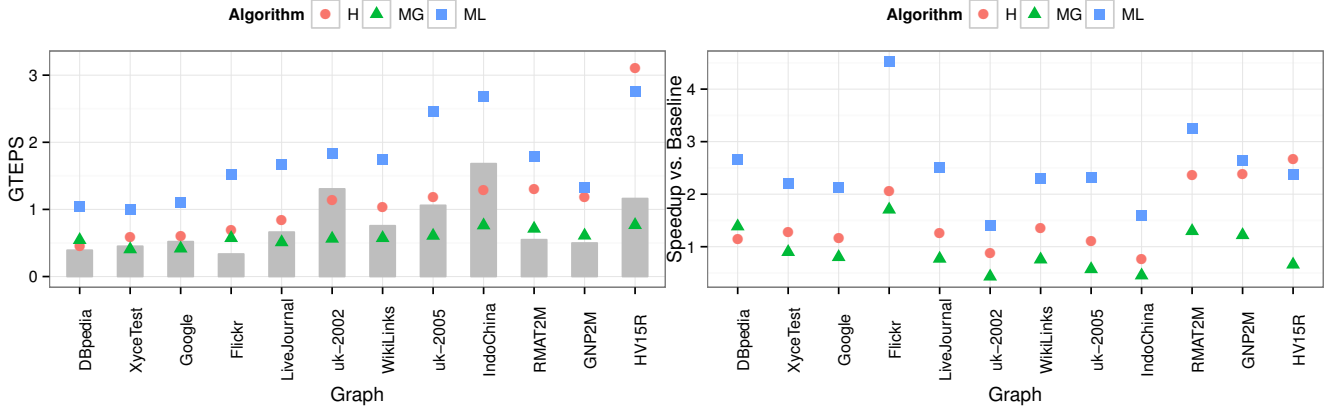

Fig. 3. Color propagation performance on Tesla K40M using Manhattan-Local (ML), Manhattan-Global (MG), and Hierarchical (H) loop collapse strategies.

that Flickr is not a very large graph, and neither does it have a high average degree.

Next, we summarize the impact of other optimization strategies discussed in Section III. Figure 2 gives BFS performance of the baseline and the ML variant again. In addition, we add optimizations in a structured manner to the code, starting with Baseline and finally getting to tuned ML (indicated by M+C+S+L in the figure). The intermediate steps are untuned Manhattan collapse (indicated by Baseline+M), Manhattan collapse with memory coalescing (M+C), Manhattan collapse with memory coalescing and utilizing team-based scan procedures (M+C+S). The final step is the usage of temporary shared memory arrays for each thread team. It is interesting to note that Manhattan collapse by itself does not provide much performance improvement. It is only after a methodical restructuring of the code, including optimizations such as coalescing and use of optimized scan primitives, that we are able to get the full benefit of the loop collapse optimization.

C. Color Propagation performance

Figure 3 shows the performance of the loop collapse strategies on color propagation. Unlike BFS, the global Manhattan collapse strategy does not consistently improve performance

over the baseline. For a majority of the graphs, it is actually slower than baseline. Using H and ML, though, the speedups (geometric mean) over baseline are 3.10× and 1.72×, respectively, for the twelve graphs considered. Performance of the best variant (ML) seems to well-correlated with average graph degree, with the exception of the synthetic RMAT2M and GNP2M graphs. We see the highest overall speedup over baseline (nearly $4.5\times$) with the Flickr graph. MG performs poorly on several instances due to the nature of iterative color propagation, which tends to have a long tail containing lots of low degree vertices. This effect is especially pronounced on the web graphs (uk, IndoChina), which tend to have long strings of singly-connected vertices. This hurts the performance of MG relative to ML in two ways. Firstly, the low average degree increases the amount of total transfer per team to and from the global prefix sum array. Secondly, the *consistently* low degree offers no benefit with regards to work partitioning among teams relative to the other approaches.

D. SCC evaluation, performance portability

We finally evaluate performance of various SCC implementations. Recall that SCC algorithms use both BFS-like and color propagation-like loop nests, in addition to other graph topology-related work reduction heuristics [18], [33]. Our baseline Kokkos implementation for SCC is based on our prior Multistep [33] multicore parallel algorithm. Note that the BFS subroutine in Multistep is direction-optimizing, similar to [3], [7]. Thus we have also been able to express a fast heuristic work-reduction strategy in our Kokkos framework. We further improved the baseline approach using the ML and MG loop collapse strategies, and other GPU-specific optimizations. Figure 4 provides a cross-platform comparison of the various approaches on our test suite. Performance rates are indicated in terms of billions of edges per second. Our prior CPU Multistep implementation uses OpenMP, and can be compiled and run on x86 systems as well as Intel's Xeon Phi coprocessors. We thus report these results on the Sandy Bridge-EP host processor and the Xeon Phi coprocessor. The platforms are indicated as SNB and KNC in the figure, and the OpenMP Multistep implementation is labeled OMP. The Kokkos baseline approach runs on all four platforms, and it is labeled as previous (B). Because Kokkos uses a single thread per team for the Xeon Phis and CPU, we only report performance for the MG variant of loop collapse on these systems, as the ML variant would default into an inefficient (B).

Consider the SNB column of the figure first. We observe that the OMP multistep performance varies between 0.1 to 5 GEPS, nearly a $50\times$ variation. The anomalously-high performance on RMAT2M and GNP2M is due to the fact these synthetic graphs are relatively easy instances for the Multistep algorithm (there is only a single non-trivial SCC, so color propagation is never run). The graphs are ordered from top to bottom by average vertex degree. While OMP tends to do better than the Kokkos baseline on smaller graphs, for four of the twelve graphs, including the larger uk web crawls, the Kokkos baseline is in fact faster than the state-of-the-art OpenMP-based Multistep. MG, the algorithmic variant designed for GPUs, did not do as well as the baseline in SNB.

TABLE II
CROSS-ARCHITECTURAL PERFORMANCE COMPARISON OF BEST VARIANTS.

Network	SCC				BFS	Coloring
	SNB	KNC	K20X	K40M	K40M vs K20X	
	GEPS				GTEPS ratio	
Google	0.16	0.08	0.09	0.15	1.66	1.19
Flickr	1.29	0.14	0.38	0.56	1.37	1.00
XyceTest	0.25	0.13	0.14	0.23	1.08	1.16
LiveJournal	1.03	0.24	0.53	0.68	1.11	1.04
RMAT2M	4.99	0.47	1.28	1.35	1.13	0.95
GNP2M	3.66	0.56	1.06	1.05	1.04	1.04
Indochina	0.30	0.09	0.11	0.18	1.08	1.22
DBpedia	0.20		0.33	0.34	0.82	1.16
HV15R	2.09	0.43	1.18	1.23	2.20	0.99
uk-2002	0.55		0.20	0.33	1.26	1.15
WikiLinks	0.79			0.45	1.27	1.12
uk-2005	0.22			0.18		
Geom mean	0.69	0.21	0.35	0.43	1.23×	1.09×

The Xeon Phi performance results are quite interesting. The Kokkos baseline variant now consistently outperforms Multistep OMP. In comparison to parallel SNB performance, the absolute performance results on KNC are lower. However, note that these results were obtained with little or no parameter tuning for KNC. Besides three instances, MG again lags behind baseline. Thus we can conclude that the loop collapse strategies designed specifically for GPUs may not really lead to portable performance on KNC, without additional tuning.

The GPU SCC performance results are as expected. Notably, we could easily combine the Kokkos BFS and coloring implementations to create this SCC algorithm, and overall performance is quite favorable in comparison to the best parallel CPU implementation.

The original Multistep algorithm compiled with OpenMP and running on CPU shows the most consistent performance, followed by the GPU Kokkos ML algorithm running on GPU. Exploring cross-architectural and cross-implementation performance on each graph instance, we note different reasons for why a particular implementation is faster or slower. Multistep was designed to run on CPU with low diameter graphs and, as such, tends to dominate performance-wise on the smaller graphs, where there is lower available work parallelism, the graphs are less skewed, and the problems are generally easier to solve. This is apparent on the two simplest instances, the GNP and RMAT graphs. The GPU ML code arguably shows increasing relative performance with increasing problem difficulty, which is exemplified by DBpedia, the most skewed graph with the largest number of nontrivial SCCs. The additional parallelism for GPU ML across the adjacencies of the largest outliers in DBpedia makes a large relative impact. MG for both CPU and GPU does not show as good performance for DBpedia because, while there is improved parallelism across the largest adjacencies, DBpedia also has a very long tail of low degree vertices. This makes color propagation run very slow with MG due to all of the additional read and writes to

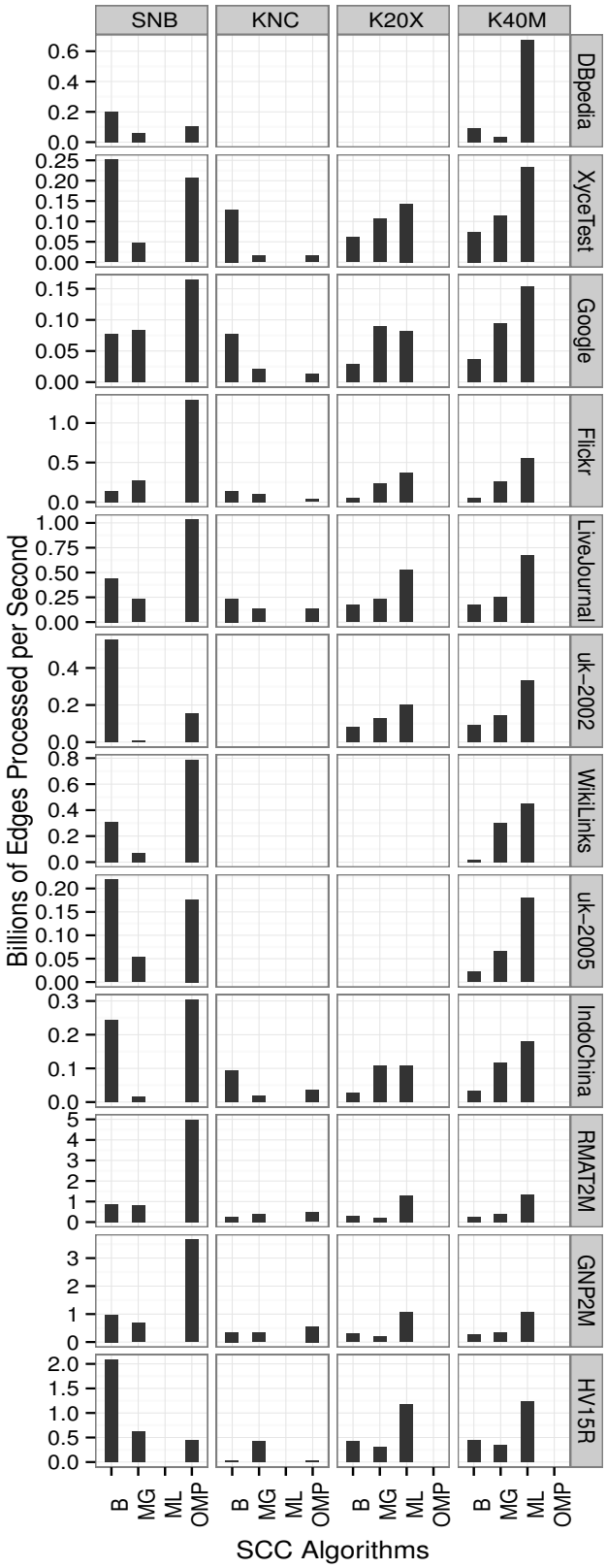

Fig. 4. Cross-platform performance comparison of SCC implementations.

the global prefix sums array.

In Table II, we list the SCC GEPS rate of the best-performing variant on each platform. The geometric mean of GEPS rates on each platform for SCC are also listed. Overall, K20X is $1.67\times$ faster than KNC for SCC. For BFS and Coloring, we compare performance of the best variant on K40M to the best-performing one on K20X. We observe that K40M is overall $1.23\times$ faster for BFS and $1.09\times$ faster for color propagation.

E. Comparisons to prior work

To the best of our knowledge, this is the first work on Kokkos-based graph computations targeting GPUs and Xeon Phi accelerators. For SCC, we performed direct comparisons with our prior OpenMP based Multistep method, as discussed in the previous subsection. In terms of mean performance rates, we believe that 0.43 GEPS for SCC, using a high-level framework such as Kokkos, is significant. Our mean BFS performance rate on the K40M is 1.74 GTEPS across all the test networks considered, and the best rate is 2.82 GTEPS for the RMAT2M network. Recently, Nguyen et al. [26] compare performance of several parallel graph analysis frameworks (Ligra, Galois, PowerGraph, GraphChi, and variants) for various graph analytics routines on a 40-core Intel Westmere-EX system. The best BFS performance reported, with Galois on the twitter40 graph, corresponds to a GTEPS rate of 2.1. Merrill et al. [23] report up to 3.3 GTEPS on the RMAT2M network for an optimized CUDA BFS implementation. For tuned CUDA-based SSSP approaches, Davidson et al. [9] report a peak performance rate of 0.35 GTEPS on an RMAT network and an NVIDIA GTX 680 (GK104). Thus, we believe that our approaches are competitive with the current state-of-the-art on multicore and manycore platforms.

V. Conclusions

We used an algorithmic template that is common to lot of graph algorithms to express algorithms for strongly connected components, breadth first search and color propagation. This algorithmic template was used for a portable manycore implementation using the Kokkos library and then optimized for architecture specific features like teams of threads and algorithmic features like loop-collapsing. We gave credence to the the efficacy of our approach by demonstrating the performance of a strongly connected components algorithm that is up to $3.25\times$ faster than parallel CPU implementation.

We conclude with some commentary on questions posed in Section I. Wherever possible, we advocate using simple array-based data structures and an iterative loop nest to perform graph computations, as shown in Algorithm 1. This simplifies transitioning from serial to multicore to manycore algorithms. The *Local Manhattan collapse* optimization proved to be the biggest contributor to performance improvement over a baseline version. Given that most current and emerging real-world networks have skewed degree distributions, this would be the primary optimization strategy for graph analytics. The algorithms we studied in this paper use the abstraction: "given a large unordered set of vertices, how do we efficiently read and update attributes of the vertices and their adjacencies?" Using Kokkos, we see promising results for performance

portability. The performance of our Baseline SCC algorithm on Xeon Phi is 1.97× faster than an OpenMP-based implementation. Further, the multicore CPU algorithm based on Kokkos is only 30% slower than a hand-tuned OpenMP code.

We intend to apply this methodology to other graph analytics in future work. There are several research efforts on using both the host and the accelerator for graph analytic workloads [12], [13], [17], and this is another avenue for futute work.

ACKNOWLEDGMENT

We thank Carter Edwards and Christian Trott for adding features to Kokkos that enabled us to do this work. This research is part of the Blue Waters sustained-petascale computing project, which is supported by the National Science Foundation (awards OCI-0725070, ACI-1238993, and ACI-1444747) and the state of Illinois. Blue Waters is a joint effort of the University of Illinois at Urbana-Champaign and its National Center for Supercomputing Applications. This work is also supported by NSF grants ACI-1253881, CCF-1439057, and the DOE Office of Science through the FASTMath SciDAC Institute. Sandia National Laboratories is a multi-program laboratory managed and operated by Sandia Corporation, a wholly owned subsidiary of Lockheed Martin Corporation, for the U.S. Department of Energy's National Nuclear Security Administration under contract DE-AC04-94AL85000.

REFERENCES

[1] D. F. Bacon, S. L. Graham, and O. J. Sharp, "Compiler transformations for high-performance computing," *ACM Computing Surveys*, vol. 26, no. 4, pp. 345–420, 1994.

[2] D. S. Banerjee, S. Sharma, and K. Kothapalli, "Work efficient parallel algorithms for large graph exploration," in *Proc. Int'l. Conf. on High Performance Computing (HiPC)*, 2013.

[3] S. Beamer, K. Asanović, and D. Patterson, "Direction-optimizing breadth-first search," in *Proc. Int'l. Conf. for High Performance Computing, Networking, Storage and Analysis (SC)*, 2013.

[4] J. W. Berry, B. Hendrickson, S. Kahan, and P. Konecny, "Software and algorithms for graph queries on multithreaded architectures," in *Proc. Workshop on Multithreaded Architectures and Applications (MTAAP)*, 2007.

[5] M. Burtscher, R. Nasre, and K. Pingali, "A quantitative study of irregular programs on GPUs," in *Proc. IEEE Int'l. Symp. on Workload Characterization (IISWC)*, 2012.

[6] V. T. Chakaravarthy, F. Checconi, F. Petrini, and Y. Sabharwal, "Scalable single source shortest path algorithms for massively parallel systems," in *Proc. IEEE Int'l. Parallel and Distributed Proc. Symp. (IPDPS)*, 2014.

[7] F. Checconi and F. Petrini, "Traversing trillions of edges in real-time: Graph exploration on large-scale parallel machines," in *Proc. IEEE Int'l. Parallel and Distributed Proc. Symp. (IPDPS)*, 2014.

[8] G. H. Dal, W. A. Kosters, and F. W. Takes, "Fast diameter computation of large sparse graphs," in *Proc. IEEE Int'l. Conf. on Parallel, Distributed and Network-based Processing (PDP)*, 2014.

[9] A. Davidson, S. Baxter, M. Garland, and J. D. Owens, "Work-efficient parallel GPU methods for single-source shortest paths," in *Proc. IEEE Int'l. Parallel and Distributed Proc. Symp. (IPDPS)*, 2014.

[10] T. A. Davis and Y. Hu, "The University of Florida sparse matrix collection," *ACM Transactions on Mathematical Software*, vol. 38, no. 1, pp. 1–25, 2011.

[11] H. C. Edwards, C. R. Trott, and D. Sunderland, "Kokkos: Enabling manycore performance portability through polymorphic memory access patterns," *Journal of Parallel and Distributed Computing*, 2014, in press.

[12] T. Gao, Y. Lu, B. Zhang, and G. Suo, "Using the Intel Many Integrated Core to accelerate graph traversal," *Sage Int'l. Journal of High Performance Computing Applications*, vol. 28, no. 3, pp. 255–266, 2014.

[13] A. Gharaibeh, L. B. Costa, E. Santos-Neto, and M. Ripeanu, "On graphs, GPUs, and blind dating: A workload to processor matchmaking quest," in *Proc. IEEE Int'l. Parallel and Distributed Proc. Symp. (IPDPS)*, 2013.

[14] J. E. Gonzalez, Y. Low, H. Gu, D. Bickson, and C. Guestrin, "PowerGraph: Distributed graph-parallel computation on natural graphs," in *Proc. USENIX Conf. on Operating Systems Design and Implementation (OSDI)*, 2012.

[15] D. Gregor and A. Lumsdaine, "Lifting sequential graph algorithms for distributed-memory parallel computation," in *Proc. ACM SIGPLAN Conf. on Object Oriented Programming, Systems, Languages, and Applications (OOPSLA 2005)*, 2005.

[16] S. Hong, S. K. Kim, T. Oguntebi, and K. Olukotun, "Accelerating CUDA graph algorithms at maximum warp," in *Proc. ACM SIGPLAN Symp. on Principles and Practice of Parallel Programming (PPoPP)*, 2011.

[17] S. Hong, T. Oguntebi, and K. Olukotun, "Efficient parallel graph exploration on multi-core CPU and GPU," in *Proc. Int'l. Conf. on Parallel Architectures and Compilation Techniques (PACT)*, 2011.

[18] S. Hong, N. C. Rodia, and K. Olukotun, "On fast parallel detection of strongly connected components (scc) in small-world graphs," in *Proc. Int'l. Conf. for High Performance Computing, Networking, Storage and Analysis (SC)*, 2013.

[19] M. Kulkarni, K. Pingali, B. Walter, G. Ramanarayanan, K. Bala, and L. P. Chew, "Optimistic parallelism requires abstractions," in *Proc. ACM SIGPLAN Conf. on Programming language design and implementation (PLDI)*, 2007.

[20] J. Kunegis, "KONECT - the Koblenz network collection," http://konect.uni-koblenz.de/, last accessed Oct 17, 2014.

[21] J. Leskovec, "SNAP: Stanford network analysis project," http://snap.stanford.edu/index.html, last accessed Oct 17, 2014.

[22] A. McLaughlin and D. A. Bader, "Scalable and high performance betweenness centrality on the GPU," in *Proc. Int'l. Conf. for High Performance Computing, Networking, Storage and Analysis (SC)*, 2014.

[23] D. Merrill, M. Garland, and A. Grimshaw, "Scalable GPU graph traversal," in *Proc. ACM SIGPLAN Symp. on Principles and Practice of Parallel Programming (PPoPP)*, 2012.

[24] A. Mislove, M. Marcon, K. P. Gummadi, P. Druschel, and B. Bhattacharjee, "Measurement and analysis of online social networks," in *Proc. ACM SIGCOMM Conf. on Internet Measurement (IMC)*, 2007.

[25] A. Montresor, F. D. Pellegrini, and D. Miorandi, "Distributed k-Core decomposition," 2011, arXiv:1103.5320.

[26] D. Nguyen, A. Lenharth, and K. Pingali, "A lightweight infrastructure for graph analytics," in *Proc. ACM Symp. on Operating Systems Principles (SOSP)*, 2013.

[27] S. Orzan, "On distributed verication and veried distribution," Ph.D. dissertation, Free University of Amsterdam, 2004.

[28] A. Pothen and C.-J. Fan, "Computing the block triangular form of a sparse matrix," *ACM Transactions on Mathematical Software*, vol. 16, no. 4, pp. 303–324, 1990.

[29] U. N. Raghavan, R. Albert, and S. Kumara, "Near linear time algorithm to detect community structures in large-scale networks," *Physical Review E*, vol. 76, no. 3, p. 036106, 2007.

[30] M. Ringenburg and S.-E. Choi, "Optimizing loop-level parallelism in Cray XMT™ applications," in *Proc. Cray User Group meeting (CUG)*, 2009.

[31] J. Shun and G. E. Blelloch, "Ligra: a lightweight graph processing framework for shared memory," in *Proc. ACM SIGPLAN Symp. on Principles and Practice of Parallel Programming (PPoPP)*, 2013.

[32] G. M. Slota and K. Madduri, "Simple parallel biconnectivity approaches for multicore platforms," in *Proc. IEEE Int'l. Conf. on High Performance Computing (HiPC)*, 2014, to appear.

[33] G. M. Slota, S. Rajamanickam, and K. Madduri, "BFS and coloring-based parallel algorithms for strongly connected components and related problems," in *Proc. IEEE Int'l. Parallel and Distributed Proc. Symp. (IPDPS)*, 2014.

[34] ——, "PULP: Scalable multi-objective multi-constraint partitioning for small-world networks," in *Proc. IEEE Int'l. Conf. on Big Data (BigData)*, 2014.

[35] C. L. Staudt, A. Sazonovs, and H. Meyerhenke, "NetworKit: An interactive tool suite for high-performance network analysis," 2014, arXiv:1403.3005.

[36] J. Zhong and B. He, "Medusa: Simplified graph processing on GPUs," *IEEE Transactions on Parallel and Distributed Systems*, vol. 25, no. 6, pp. 1543–1552, 2014.