This paper will apper in Parallel Architectures and Compilation Techniques(PACT) October 10-14, 2011

Efficient Parallel Graph Exploration on Multi-Core CPU and GPU

Sungpack Hong, Tayo Oguntebi, and Kunle Olukotun
Pervasive Parallelism Laboratory, Stanford University
{hongsup, tayo, kunle}@stanford.edu

***Abstract*—Graphs are a fundamental data representation that have been used extensively in various domains. In graph-based applications, a systematic exploration of the graph such as a breadth-first search (BFS) often serves as a key component in the processing of their massive data sets. In this paper, we present a new method for implementing the parallel BFS algorithm on multi-core CPUs which exploits a fundamental property of randomly shaped real-world graph instances. By utilizing memory bandwidth more efficiently, our method shows improved performance over the current state-of-the-art implementation and increases its advantage as the size of the graph increases. We then propose a hybrid method which, for each level of the BFS algorithm, dynamically chooses the best implementation from: a sequential execution, two different methods of multi-core execution, and a GPU execution. Such a hybrid approach provides the best performance for each graph size while avoiding poor worst-case performance on high-diameter graphs. Finally, we study the effects of the underlying architecture on BFS performance by comparing multiple CPU and GPU systems; a high-end GPU system performed as well as a quad-socket high-end CPU system.**

I. Introduction

The first decade of the new millennium has ended, producing many new interesting prospects in the computing landscape. Most prominently, multi-core CPUs have become commonplace, as they are widely used not only for high performance computations but also in ubiquitous consumer electronics such as hand-held devices. The idea of using the graphics processor for general purpose computation has also become popular, since this approach has yielded tremendous performance when applied to suitable problems [1]. Such proliferation of parallelism (multiple threads on a CPU or GPU) and heterogeneity (simultaneous use of a CPU and GPU) has succeeded in greatly improving the performance of many traditional computation-intensive workloads, provided that their parallel or heterogeneous implementations are well understood.

There still remain, however, problems that demand fast computation but for whom efficient parallel or heterogeneous implementations have yet to be identified. Graph exploration is one important example of such problems. Graphs are a fundamental data representation widely used in numerous fields such as intelligence analysis [2], robotics [3], social network analysis [4], and computational biology [5]. These applications have traditionally required long periods of processing time due to their massive data-set sizes. Parallelism has usually failed to alleviate matters, because parallel speedup of these applications is severely limited by the random nature of their memory access patterns, a fundamental property of graph processing algorithms [6].

Breadth-first search (BFS) is a fundamental graph algorithm that systematically explores the nodes in the graph. BFS is typically considered one of the most important graph algorithms, because it serves as a building block for many other algorithms including betweenness centrality calculation [4], connected component identification [7], community structure detection [8], and max-flow computation [9]. Benchmark suites targeting graph applications perennially include BFS as a primary element [10], [11].

Due to such importance, significant research has been conducted to efficiently implement a parallel BFS for a wide array of computing systems [12]–[19]. Two recent results particularly draw our attention. One is Agarwal et al's work [18] which presented a state-of-the-art BFS implementation for multi-core systems. Their implementation utilized sophisticated data structures to reduce cache coherence traffic between CPU sockets. When executed on a high-end multi-core system, their implementation outperformed previous proposals, even those including other architectures such as Cell [14], clusters [12], and shared memory supercomputers [13], [16].

The other proposal is a BFS implementation for GPUs by Hong et al [19], reported around the same time as the Agarwal et al paper. Hong et al solved the workload imbalance issue when processing irregularly shaped graphs, which had a devastating effect on previous GPU implementations. They demonstrated good performance improvement compared to multi-core CPU implementations. However, their comparison included neither Agarwal's work nor more recent architectures such as the Nehalem CPU family [20] or Fermi GPU [21].

In this study, we build upon ideas from both previous works and incorporate them into a universal solution that utilizes both the CPU and GPU on a heterogeneous system. Specifically, we first propose a new BFS implementation for multi-core CPUs which performs better than state-of-the-art implementations [18] for large graph instances, while being simpler to implement (Section III-A). Second, we propose a hybrid method that, for each BFS iteration, dynamically selects the best execution method among sequential, parallel or GPU implementations. This approach benefits both large and small graphs and also prevents poor worst case performance (Section III-B). Finally, using the best implementations for both systems, we measure BFS performance on multi-core

CPU systems and GPU systems and study their architectural effects.

The specific contributions of this paper are:

- We present a BFS implementation method for multi-core CPUs which performs better than the current state-of-the-art implementation for large graph instances. The performance gap widens as the graph size grows. Additionally, our method is much simpler to apply.
- We present a hybrid method which dynamically chooses the best execution method for each BFS-level iteration from among: sequential execution, multi-core CPU executions, and GPU execution. We show that such a hybrid method is essential to prevent poor worst case performance.
- We provide a fair comparison of BFS performance on multi-core CPU and GPU, which reveals that single-socket high-end GPU performance can be matched by a quad-socket high-end multi-core CPU.

The remainder of this paper is organized as follows. In Section II, we observe the nature of the parallel BFS algorithm when applied to real-world graph instances. Based on these observations, we propose the new implementation methods that we outline in Section III. Section IV details the methodology for our experiments of which the results can be found in Section V. We also review the previous BFS implementations in Section VI, before concluding in Section VII.

II. Nature of Parallel BFS Algorithm on Irregular Graphs

Breadth-first search (BFS) is a fundamental graph search algorithm that systematically traverses the connected nodes of a graph starting from a given root node. The algorithm ensures that all nodes are visited in the order of hop distance from the root– a node with smaller hop distance is always visited before one with larger hop distance.

An important kernel, many applications use BFS as their basic building block; such applications insert extra computation during each BFS iteration [2]–[4], [8] and/or post-process the result [4], [7], [9]. Although the term *search* misleadingly implies that the algorithm should terminate once a designated node has been reached, many of these applications simply traverse all the connected nodes in BFS order. Also, in order to accommodate such a wide range of applications, many BFS implementations simply store the *BFS level* (i.e. hop distance from the root) of each node as their final output [12], [15], [19].

Two different strategies have been proposed for parallel (and distributed) execution of BFS. The first method, known as the *fixed-point* algorithm, continuously update the BFS level of every node, based on BFS levels of all neighboring nodes until no more updates are made. This method is preferred in distributed environments since it can naturally be implemented as message passing between neighboring nodes [12]. On the other hand, this method potentially wastes computation, since it processes the same edge multiple times whenever a corresponding node is updated. For this reason, the fixed-point algorithm is less favored in shared-memory environments than a second strategy: the *level synchronous* BFS algorithm.

Algorithm 1 Level Synchronous Parallel BFS

```
1:  procedure BFS(r:Node)
2:      V = C = ∅; N = {r}          ▷ Visited, Current, and Next set
3:      r.lev = level = 0
4:      repeat
5:          C = N
6:          for Node c ∈ C do                          ▷ in parallel
7:              for Node n ∈ Nbr(c) do                 ▷ in parallel
8:                  if n ∉ V then
9:                      N = N ∪ {n}; V = V ∪ {n}
10:                     n.lev = level + 1
11:         level++
12:     until N = ∅
```

Level	Num. Nodes	Fraction (%)$^{+}$
0	1	$3.1*10^{-6}$
1	4	$1.3*10^{-5}$
2	749	$2.0*10^{-3}$
3	109,239	0.34
4	7,103,690	22.20
5	9,088,766	28.40
6	130,298	0.41
7	172	$5.3*10^{-4}$
total visited nodes	16,432,919	51.35
total visited edges	255,962,977	99.99

$^{(+)}$ Fraction of the total number of nodes (edges) in the graph

TABLE I
Number of nodes in each BFS level: A result from typical execution in an RMAT graph

The level synchronous BFS algorithm is described in Algorithm 1. The algorithm manages three sets of nodes: the visited set V, current-level set C and next-level set N. Iteratively, the algorithm visits all the nodes in the current level (C) in parallel and collects the next-level set N (line 6 – 11). At the beginning of the next level iteration, C is populated with the values from N, and N is cleared for the new iteration. The iteration continues until there is no node in the next level. (line 12). In short, this method visits all the nodes in each BFS level in parallel, with the parallel executions being synchronized at the end of each level iteration.

This strategy bears its own shortcomings: (1) Synchronization overhead needs to be paid at every BFS level, and (2) the amount of available parallelism is limited by the number of nodes in a given BFS level. Nevertheless, the strategy works quite well in practice for real-world graph instances that are irregularly shaped by nature. This is because it has been observed that the diameters of real-world graphs are small even for large graph instances, i.e. the small world phenomenon [22]. Consequently, the overhead of level-wise synchronization is tolerable since the number of synchronization events– the diameter of the graph– is small.

Similarly, because of the small world phenomenon the number of nodes in each BFS level cannot help but grow very rapidly. As an illustration, Table I shows the number of nodes in each BFS level, obtained from a typical BFS execution on a synthetic graph with 32 million nodes and 256 million edges

generated by RMAT model [23]. (See Section IV for more discussion about our graph generation models.) From the table, one can observe that the maximum BFS level is small (7) for such a large graph and that most of the nodes belong to only two levels (levels 4 and 5). Therefore the total execution time is bounded by the traversal of these levels, but the degree of parallelism (i.e. number of nodes) is large in these levels.

Despite these nice properties, it has been shown that the performance of level synchronous BFS is greatly affected by details of the specific implementation as well as the conformity to the targeted machine architecture [12], [13], [16], [18], [19]. In fact, we noticed that many previous implementations do not fully utilize the properties of the BFS algorithm on real-world graphs which we have discussed in this section, e.g. exponential growth of nodes in each level. In the next section, we propose a new implementation method for the level synchronous parallel BFS algorithm which considers such algorithmic properties as well as the underlying machine architecture.

III. Implementation Methods of Parallel BFS

A. A New Method for Multi-Core CPU

In implementing the level synchronous parallel BFS algorithm, there exists a rather direct implementation based on Algorithm 1 which uses lock-protected shared queues to implement the current-level set (C) and next-level set (N). Unfortunately such a naive implementation would surely suffer from significant locking overhead.

Recently, Agarwal et al [18] presented a state-of-the-art BFS implementation for multi-core CPUs in which they applied the following series of optimization techniques:

1) Use of a bitmap to compactly represent the visited set
2) Application of the 'test and test-and-set' operation when atomically updating the bitmap
3) Use of local next-level queues; process node insertions into the global queue in batch.
4) Maintaining per-socket next-level queues which are carefully implemented with ticket-locks and a fast-forwarding algorithm.

Among these optimizations, the first three are relatively easy to apply. Fig 1 shows the pseudo-code of such an implementation. In the code, `Bitmap V` (line 3) encodes the set of visited nodes. Since this set is the most frequently accessed data in the algorithm, using a bitmap data structure minimizes its size, allowing the cache to hold the largest possible portion of the set. Parallel access overhead to the Bitmap is minimized by using intrinsic atomic operations (e.g. `__fetch_and_or`) as well as the *test and test-and-set* method (line 11 - 12). The next-level nodes are first stored in the per-thread local queue (`LQ`) until they are bulk inserted into the global queue (line 16, 20). These bulk insertion can also be efficiently implemented using a single atomic operation by first increasing the queue index (`__fetch_and_add`) then performing a normal memory copy to the previous index. This is possible because the queue is only pushed but never popped by any thread in this stage. We refer to the method in Fig 1 as the *Queue-based method* in the rest of this paper.

Argarwal et al's final optimization was to use a delicate queue implementation that minimized unnecessary coherence traffic as much as possible during queue operations. Even though this optimization provided some impressive performance benefit, we observe that the importance of reducing coherence misses is greatly diminished when the size of the input data set becomes very large and the performance is primarily governed by capacity misses.

We therefore take a different approach, focusing on efficient use of memory bandwidth. Our approach is inspired by previous BFS implementations for GPUs [15], [19] in which use of shared queues is intentionally avoided due to architectural traits of the GPU. Instead of a shared queue, the GPU implementations manage a single O(N) array that tells if a node belongs to the current-level set, next-level set, or the visited set. The array is repeatedly accessed at each level iteration, exploiting the vast memory bandwidth of the GPU.

Fundamentally, our approach merges and builds upon the key ideas from the previous approaches for CPU [18] and GPU [19]. The pseudo-code for our new method is presented in Fig 2. As in the Queue-based method, we keep the visited set as a bitmap (line 27) and access it through the same atomic updates (line 38, 39). In contrast to the Queue-based method, the next-level set and the current-level set are implemented together as a single O(N) array as in the GPU implementation.[1] Specifically, if the level of a node equals the current level, it means the node belongs to the current-level set (Line 36). Likewise, setting a node's level value to be current level plus one is, in fact, putting the node into the next-level set (Line 40). We refer this method as the *Read-based method*.

The Read-based method provides two major advantages. First, it is completely free from queue overhead. Not only do we remove atomic instructions previously used for the queue operations, we also save on cache and memory bandwidth. Second, the Read-based method's memory access pattern is more sequential. This idea is illustrated in Fig 3. Fig 3.(a) depicts the data access pattern for a Queue-based implementation in which there are six nodes ({0, 4, 2, 9, 7, 1}) in the current-level queue and a thread processes the first three nodes in order. Consequently, the algorithm accesses the adjacency list in random order. On the other hand, Fig 3.(b) shows that, for the same input, the Read-based method maximizes opportunity for sequential reads during level and adjacency list exploration (Line 36,37 in Fig 2).

As further justification of our approach, we measured the sequential and random read bandwidths on various machines (See Table III in Section IV for detailed specifications of these machines). Table II displays the results. Most notably, the data shows about a 9x difference in bandwidth between sequential and random reads on the multi-core CPUs, further highlighting the importance of our Read-based method's more sequential

[1]The per-node *lev* field in Fig 2 is actually implemented as a separate byte array. That is, `c.lev` in fact is `lev[c.id]`.

```
1  BFS_Queue(G: Graph, r: Node) {
2    Queue N, C, LQ[threads];
3    Bitmap V;
4    N.push(r); V.set(r.id);
5    int level = 0; r.lev = level;
6    while (N.size() > 0) {
7      swap(N,C); N.clear(); // swap Curr and Next
8      fork;
9        foreach(c: C.partition(tid)) {
10         foreach(n: c.nbrs) {
11           if (!V.isSet(n.id)) { // test and test-and-set
12             if (V.atomicSet(n.id)) {
13               n.lev = level+1;
14               LQ[tid].push(n);  // local queue
15               if (LQ[tid].size()==THRESHOLD){
16                 N.safeBulkPush(LQ[tid]); // global queue
17                 LQ[tid].clear();
18       } } } } }
19       if (LQ[tid].size() > 0) {
20         N.safeBulkPush(LQ[tid]);
21         LQ.clear();
22       }
23     join;
24     level++;
25 } }
```

Fig. 1. Queue-Based BFS Implementation Method

```
26 BFS_Read(G: Graph, r: Node) {
27   Bitmap V;
28   Bool fin[threads];
29   V.set(r.id);
30   int level = 0; r.lev = level;
31   bool finished = false;
32   while (!finished) {
33     fork;
34       fin[tid] = true;
35       foreach(c: G.Nodes.partition(tid)) {
36         if (c.lev != level) continue;
37         foreach(n: c.nbrs) {
38           if (!V.isSet(n.id)) { // test and test-and-set
39             if (V.atomicSet(n.id)) {
40               n.lev = level+1;
41               fin[tid] = false;
42       } } } }
43     join;
44     finished = logicalAnd(fin, threads);
45     level++;
46 } }
```

Fig. 2. Sequential Array Read-Based BFS Implementation Method

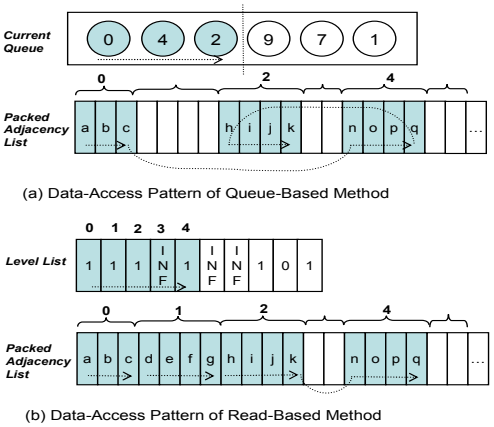


Fig. 3. Data Access Patterns

Machine	Seq. Read	Random Read
Nehalem CPU	8.6 GB/s	0.98 GB/s
Core CPU	3.0 GB/s	0.25 GB/s
Fermi GPU	76.8 GB/s	2.71 GB/s
Tesla GPU	72.5 GB/s	3.15 GB/s

TABLE II
MEMORY READ BANDWIDTH MEASURED ON MACHINES IN TABLE III.

memory access pattern. We remind the reader, however, that even the Read-based method does not completely eliminate random read accesses. The algorithm still requires additional indirection from the adjacency list to the destination node (Line 38 in Fig 2), an inherently random operation. Also, note that both the Queue-based method and Agarwal et al's proposal perform the same random accesses to the bitmap as well. All these methods including ours, however, rely on the last-level cache to process such random acceses fast.

The primary disadvantage of the Read-based method is that it reads out the entire level array at every level iteration, even if only a few nodes belong to that level. However, this seldom affects the overall performance because of the following characteristics of real-world graph instances: (1) the diameter of the graph is small so the maximum amount of re-read is bounded and (2) there are a few critical levels in which the number of nodes is O(N) (see Table I) and thus sequentially reading out the whole O(N) array is not wasteful for these levels. In addition, the total algorithm execution time is already governed by the processing time of these critical levels. We remind the reader that this *small world property* is not merely an observation made in certain graph instances, but rather a fundamental characteristic of randomly-shaped real-world graphs [22]. So while the Read-based approach may be, in theory, algorithmically inefficient, the small world property guarantees that realistic execution in real-world graphs is rarely near the worst case.

Nevertheless, this disadvantage can create an undesirable performance cliff when using the Read-based method for worst-case data inputs. Two specific examples are (1) small (sub-)graphs and (2) long diameter graphs such as meshes. In the next subsection, we propose a hybrid scheme which alleviates this problem.

B. Hybrid Methods

To address the inefficient processing of non-critical levels, we propose a hybrid scheme that dynamically determines which method to apply when processing each level. The basic idea is simple: If the current level contains only a few nodes, use the Queue-based method (Fig 1). Otherwise, use the Read-based method (Fig 2).

Our hybrid method can be represented as a state machine, shown in Fig 4. The hybrid method begins in Seq state so that level 0 (the root) is processed sequentially. Afterward, if the size of the next level is larger than T_1, the method switches to Queue state and the level is processed with the (parallel) Queue-based method. Similarly, Queue state is exited when the next level size is larger than T_2; such critical levels are

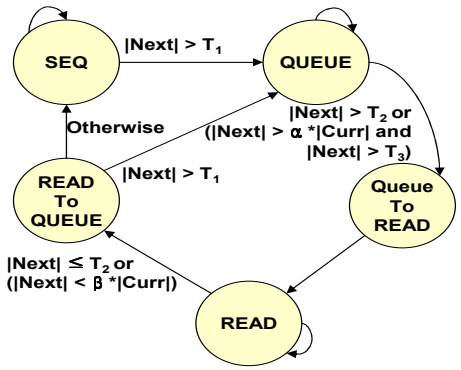


Fig. 4. State machine of Hybrid Read and Queue Method (for CPU execution).

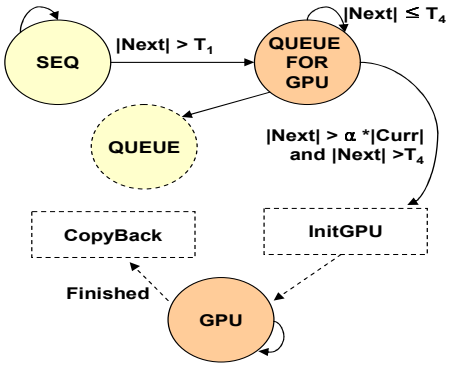


Fig. 5. State machine of the Hybrid CPU and GPU Method.

processed with the Read-based method. The size of the next-level set can be easily counted after a test-and-set operation (i.e. Line 12 or Line 39) using private counters which are merged upon synchronization.

There is also an additional QueueToRead state which represents the transitional state between Queue and Read states. In QueueToRead state, the current level set is read from the current-level queue but the next-level nodes are written back not in the queue, but to the level array.

If exponential growth in the number of nodes in the next level is detected ($|Next| > |Curr| * \alpha$), we exit Queue state early (T_3). This is based on the following observation: Since Read state is reached through one extra transitional state (QueueToRead) and the number of nodes in successive levels grows very rapidly, there will be enough nodes in the current level by the time Read state is finally reached.

The transition from Read state to Queue state is similar. Read state is exited when the size of next-level set becomes small or when the exponential growth in the number of nodes stops; the second condition is checked only if read state is reached via exponential growth.

We refer to this method, shown in Fig 4, as the *Hybrid Read and Queue method*. The Hybrid Read and Queue method avoids the worst-case inefficiency problem of the Read-based method by applying the Queue-based method for non-critical levels. We used $(T_1, T_2, T_3) = (64, max(2^{18}, N*0.01), 2048)$ and $(\alpha, \beta) = (2.0, 2.0)$.

We now observe that the idea of the hybrid method can enhance the previous GPU BFS implementation [19] using the same principles. Although not explicitly mentioned in its paper, this GPU implementation does suffer from the same non-critical level inefficiency issue as the Read-based method, since it also reads out the whole array at every level iteration.

Fig 5 shows the extension of our Hybrid Read and Queue method for GPU execution. Fundamentally, the first few levels are executed on the CPU-side before migrating to the GPU when the size of the current level becomes larger than T_4 (our default was 16384). GPU execution is initiated only if there is exponential growth in the number of nodes in each level. Otherwise, we fall back to the Queue state in Fig 4. Such a decision is based on the observation that in high-diameter graphs (e.g. 2-D meshes) there is not enough parallelism at each level to saturate the massive parallel hardware of the GPU.

Also, once GPU execution is initiated, execution never returns to the CPU until finished, since the benefit of moving back to the CPU is quickly negated by the overheads required: (a) the size of the next-level set has to be counted during GPU execution in order to determine when to come back and (b) the bitmap for the visited set has to be reconstructed on the CPU-side.

Note that in InitGPU stage in Fig 5, the whole level array is not simply copied to the GPU. Instead, we save all the contents of current-level queues during CPU-side processing and copy only those queues to the GPU. Using those queues, the GPU can reconstruct the entire level array, a much faster operation than copying the whole O(N) level array from the CPU. We refer to this scheme, shown in Fig 5, as the *Hybrid CPU and GPU method*.

To summarize, this section proposes a series of new methods for BFS implementation. We began with the Read-based method which is simple but utilizes memory-bandwidth efficiently for large graphs. Our hybrid methods prevent worst-case execution patterns when processing small or high-diameter graphs.

IV. Methodology

This section provides details of our experiments. In our experiments, we measure the performance of our BFS implementation methods in Section III with various machines and different graph instances. We primarily compare our results against those of Agarwal et al's[18], since their implementation was said to outperform virtually all other previous proposals [12], [14], [16].

As for input data, we use two different kinds of popular graph generators which produce differently shaped graph instances with a given number of nodes (N) and edges (M). The first one is the Erdős-Rényi, or uniformly random, model; this model uniformly and randomly picks M pairs of nodes out of N nodes and creates edges between them. The second one is the RMAT model [23] which produces a so-called scale-free graph, characterized by its skewed degree distribution and

	Nehalem CPU	Fermi GPU	Core CPU	Tesla GPU	SC10-EP	SC10-EX
Core Architecture	Intel Nehalem	Nvidia Fermi	Intel Core	Nvidia Tesla	Intel Nehalem	Intel Nehalem
Model No.	Xeon X5550	Tesla C2050	Xeon E5345	GeForce GTX275	Xeon X5570	Xeon X7500
Core Frequency	2.67 GHz	1.15 GHz	2.33 GHZ	1.40 GHz	2.93 GHz	2.26 GHz
Num Socket	2	1	2	1	2	4
Num Core/Socket	4	14*2[a]	4	30	4	8
HW-thread/Core	2	∼32	1	∼32	2	2
SIMD/SIMT width	- (not used)	32	-	32	-	-
Total Last Level Cache	16 MB	2MB	8 MB	-	16 MB	96 MB
Main Memory	24 GB	3GB	32 GB	896MB	48 GB	256 GB
Memory Bandwidth[b]	100 GB/s	128 GB/s	10.4 GB/s	127 GB/s	100 GB/s	266 GB/s
Total Num Transistors	1.4 Billion	3.0 Billion	1.1 Billion	1.4 Billion	1.4 Billion	9.2 Billion
Total Power (TDP)	190 W	238 W	160 W	210 W	190 W	520 W

[a] Each core processes two warps at a time.

[b] Theoretical maximum: For GPU and Nehalem CPU, this is (num channels) x (dram bandwidth). For Core CPU, this is FSB bandwidth. See Table II for bandwidths actually measured on the systems.

TABLE III
THE SPECIFICATION OF MACHINES USED IN OUR EXPERIMENTS AND THE PREVIOUS WORK [18]

fractal community structure. Both of our generators came from a graph library called SNAP [24]. For the parameters of the RMAT graph, we used default values in the SNAP library: (a,b,c)=(0.45,0.25,0.15).

As for graph representation, we used the CSR (Compressed Sparse Row) format which merges the adjacency lists of all nodes into a single O(M)-sized array, with the beginning location of each node's adjacency list stored in a separate O(N)-sized array. This data structure has been popularly used in many previous works [15], [17], [19] due to its low memory requirement and simplicity. The BFS-level of each node is stored in a separate O(N) byte array. Our code ware compiled with gcc 4.3.3 and nvcc 3.2 with the -O3 option.

Table III summarizes the specification of the machines used in our experiments and in the previous work [18]. Our main experiments are conducted on the first two machines in the table, which are referred to as Nehalem CPU and Fermi GPU in the rest of the paper.

Our experiments have two goals. First, we evaluate the effectiveness of our methods, described in Section III. Second, we study the architectural effects on BFS performance of multiple kinds of GPUs and CPUs. While evaluating our methods, we directly compare our performance against the values reported in the previous paper [18]. Note that the previous work used two different machines (SC10-EP and SC10-EX in Table III); we compare our results against the SC10-EP results since it is comparable to– although faster than– our machine (Nehalem CPU). When exploring the architectural effects, we measure the best BFS performance on all four of our machines detailed in Table III. We include both results in the previous work [18] in comparison as well.

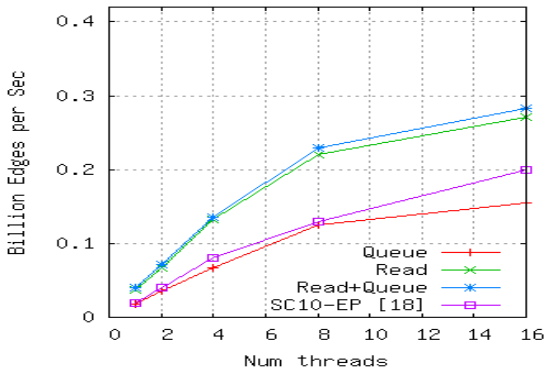
We measure our performance by executing BFS 10 times from 10 different root nodes which are (pseudo-)randomly chosen. However, all the root nodes belonged to the same connected component whose size is O(N). We take the average of multiple of such measurements. When measuring GPU performance, we do not include the time to set up the graph data structure (i.e. nodes and edges) in the GPU memory because the graph is not mutated during execution; this step is analogous to a CPU loading the graph into the main memory from the file system. However we *do* include the time for initializing the BFS-level array in GPU memory and copying back its final values to CPU memory, since these steps have to be repeated at every BFS execution.

V. EXPERIMENTAL RESULTS

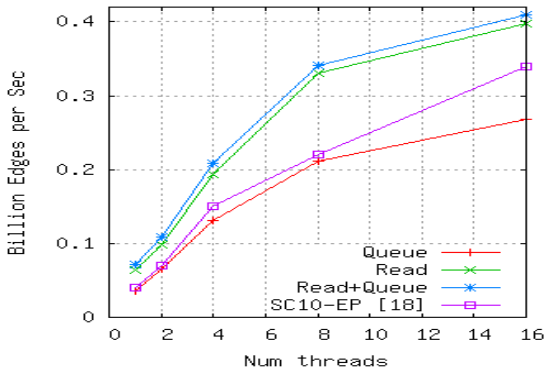
In the first part of our experiments, we validate the effectiveness of the implementation methods we presented in Section III. To begin, we measure the performance of those methods on our Nehalem CPU machine (Table III), using Uniform and RMAT graph instances with 32 million nodes and 256 million edges.

The result is shown in Fig 6 where the y-axis is the measured performance (number of processed edges per second; higher is better) and the x-axis is the number of threads used. We remind the reader that the machine has only 8 physical cores but supports two hardware threads per core. In the figures, three plots indicate our measured performances: Queue is the result of the Queue-based method (Fig 1), Read represents the Read-based method (Fig 2) and Read+Queue represents the Hybrid Read and Queue method (Fig 4). The figures also display the performance values reported in the state-of-the-art previous work [18]; their results were conducted on a machine comparable to ours (SC10-EP in Table III) with identically-sized graphs. Note that Queue-based method is not our main contribution but rather illustrates the case when a sophisticated queue data structure, such as the one Agarwal et. al.'s work relies on (Section III), is not utilized.

In the figure, one can first notice that our simple Read-based method outperforms both the plain Queue-based method and the previous work, although the previous work seems to better utilize hardware multi-threading. The performance difference comes from the Read-based method's saving of memory bandwidth and more sequential data access pattern, as discussed in Section III. Second, the Hybrid Read and Queue method provides additional performance improvement over the Read-based method but marginally. We will analyze this marginal improvement later in this section.

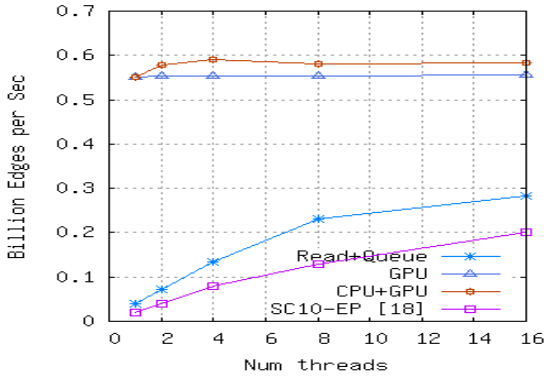


(a) Uniform

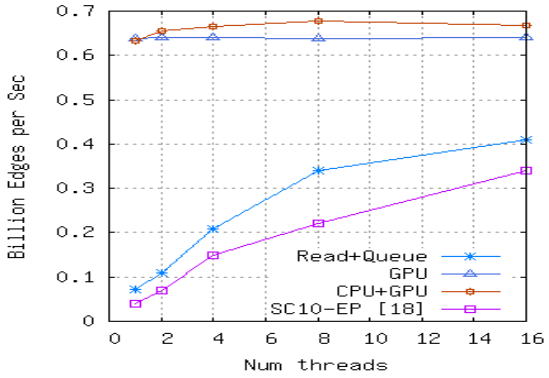


(b) RMAT

Fig. 6. Performance of BFS implementation methods measured on Nehalem CPU.



(a) Uniform



(b) RMAT

Fig. 7. Performance of BFS implementations on Fermi GPU.

Next we measure BFS performance of the same graph instances on our Fermi GPU machine (Table III) using a GPU-only execution and Hybrid CPU and GPU method. The GPU-only execution is based on Hong et al's [19] implementation. The result is shown in Fig 7, where two CPU results (Hybrid Read and Queue, and SC10-EP) are also displayed for comparison. From Fig 7, it is immediately clear that GPU execution still performs better than CPU execution, even with our new best CPU implementation. More detailed study on such architectural effects will be presented later in this section. The Hybrid CPU and GPU method provides extra performance benefits, but it is again marginal.

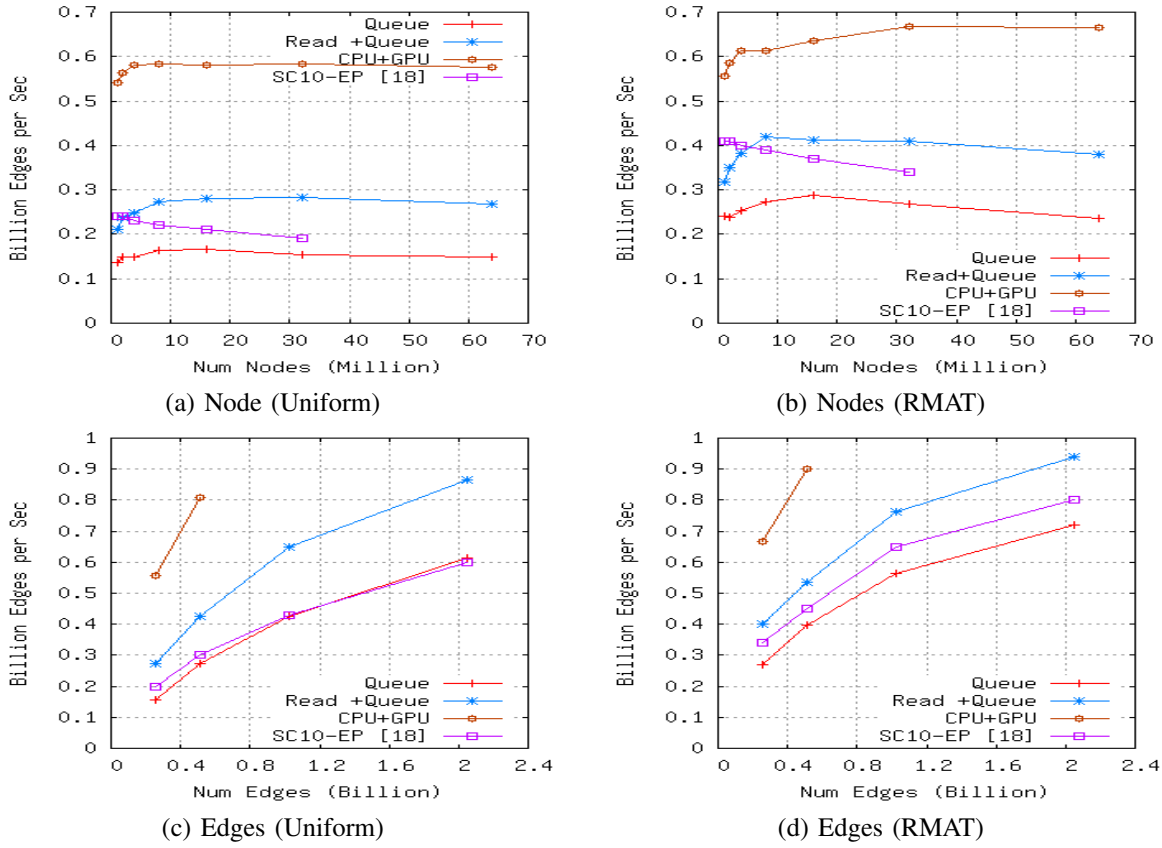
Next,we explore the effect of graph size. Using the same Uniform and RMAT graph generator, we scale the size of the graphs and measure the performance on both CPU and GPU. The results are shown in Fig 8. In Fig 8, graphs (a) and (b) change the number of nodes from 1 million to 64 million, while keeping the average degree of the graph as 8–i.e., M equals to 8 * N. To represent larger graphs, (c) and (d) change the number of edges from 256 million to 1 billion while the number of nodes is fixed at 32 million. We omit the results of GPU-only and Read-based method for brevity, as they show similar relative performance as in Fig 6 and 7.

Both Fig 8 (a) and (b) show that the GPU execution sustains a constant performance level even for large numbers of nodes, while CPU-based methods tend to drop as the number of nodes increases in the graph. This is because, as the size of the graph grows, more memory requests escape the cache and are serviced by DRAM. In addition, the plots also show that our method (Hybrid Read and Queue) performs better than Agarwal et al's method when the size of graph is large. The performance difference becomes even larger as the size of graph grows. We speculate that the performance of Agarwal et al's work would converge to that of the Queue-based method when the graph size gets large enough.

This phenomenon can be explained as the following: it is beneficial to minimize coherence misses among the last level caches as long as most data fits in those caches. But this benefit is greatly reduced when the data size outgrows the cache and the execution time is dominated by capacity misses. Our method makes use of the unified array to completely eliminate the cache/memory bandwidth used for queue management. This saved memory bandwidth is especially beneficial when the execution time is dominated by main memory accesses.

Similar trends can be also observed in (c) and (d), where we scaled up the number of edges: As the size of graph grows, the performance gap between our method and the previous work widens. Note that the graph did not fit in GPU memory when the number of edges was more than 512 million.

We now analyze more thoroughly the impact of our hybrid methods on random (i.e. low-diameter) graphs. Fig 9 shows the level-wise break down of the execution time for each BFS method, obtained from a specific run for the 32 million node RMAT instance. The number of nodes in each level can be found in Table I, which reveals that most of the nodes belong to only two levels (level 4 and 5). Since the execution time of



(a) Node (Uniform) (b) Nodes (RMAT)

(c) Edges (Uniform) (d) Edges (RMAT)

Fig. 8. Effect of Graph Size Scaling.

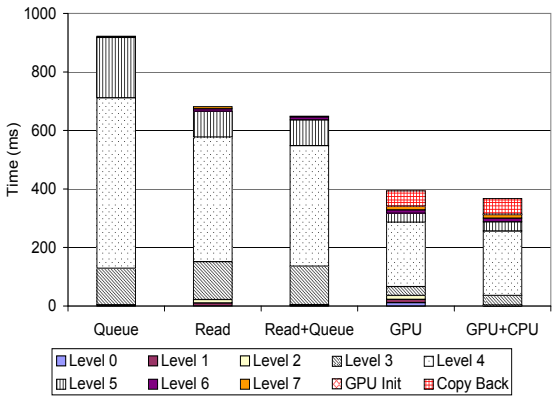


Fig. 9. Breakdown of Execution Time for Various Methods.

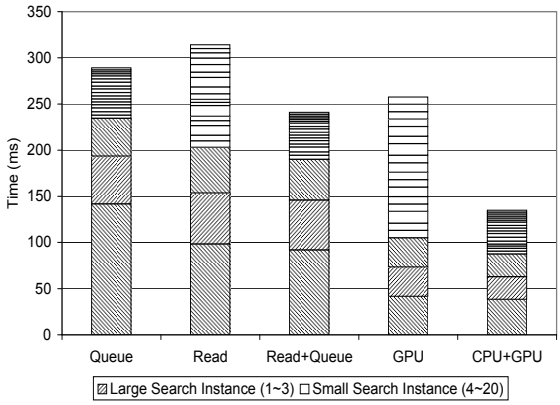


Fig. 10. Accumulated Execution Time of 20 BFS runs performed on a tree: Small and Large search instances were intentionally mixed.

each level is determined by the number of nodes in the current level and the next level, the total execution time is naturally governed by the processing times of level 3, 4, and 5. As seen in Fig 9, compared to the Queue-based method, the Read-based method performs better for these critical levels due to its efficient utilization of memory bandwidth. For non-critical levels, however, the Read-based methods performs worse. Similar observations can be made for GPU-only execution, when compared to CPU-based methods. Note that our hybrid methods reduce the execution time for non-critical levels (e.g. level 1, 2). However, since the fraction of these levels to the total execution time is little, the gains from Hybrid methods became marginal.

Despite its marginal benefit on low-diameter graphs, the hybrid method remains very valuable since it acts as a safety net to prevent performance degradation for worst-case inputs: small (sub-)graphs or high-diameter graphs. To illustrate this point, we first synthesized a complete 4-ary tree (5.5 million nodes, 11 levels) and performed 20 BFS executions on it. We started three BFS runs from the nodes in top two levels and the remaining 17 from the bottom three levels except the leaf

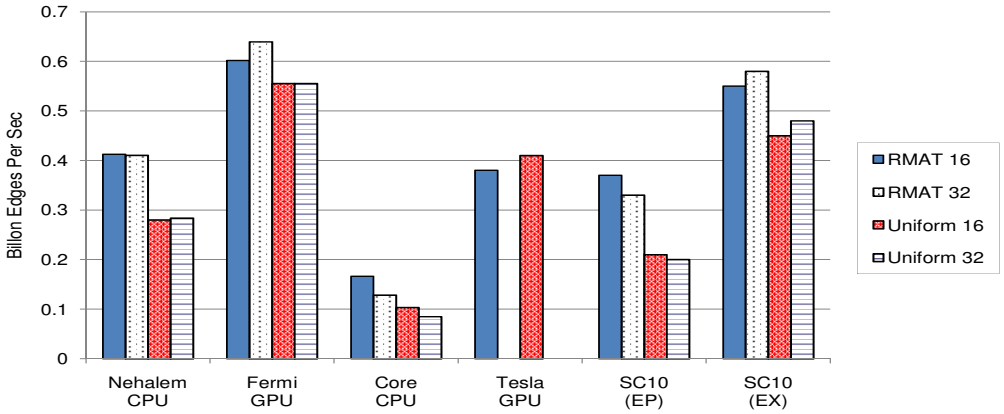


Fig. 11. BFS Execution Performance on Various Machines: RMAT and Uniform is the type of the graph. Number 16 and 32 stand for number of nodes of the graph, in millions. Each graph instance has the same average degree of 8.

Method	Normalized Execution Time
Queue	1.00
Read	12.63
Queue+Read	1.01
GPU	15.23

TABLE IV
NORMALIZED BFS EXECUTION TIME ON A 2-D MESH.

level; this setup dictates that the first three BFS executions visit at least a quarter of all nodes in the tree while the remaining 17 only visit a couple. The accumulated execution time of these searches is shown in Fig 10. The figure shows that the Read-based method or pure GPU execution suffer from huge fixed overhead for small search instances (solid boxes) while performing well for large ones (checker boxes). The Hybrid methods perform better than those methods because an appropriate iteration method is chosen for each instance size.

Similarly, we synthesized a 2-D mesh (4,000 x 4,000) to represent low-diameter graphs and measured BFS performance using various methods. Table IV shows execution time for each method, normalized by the execution time of the queue-based method. We omitted the Hybrid CPU and GPU result since it is essentially the same as the Hybrid Queue and Read method in this case. Note that there are $O(\sqrt{N})$ BFS levels in this graph. As can be seen in the table, performance of the Read and GPU methods suffers due to repeated unnecessary memory accesses while the hybrid method avoids this pitfall. The GPU also suffers especially since the degree of parallelism in each level is low for these kinds of graphs.

Now, we study the effect of the machine architectures on BFS execution. For this purpose, we executed our best BFS implementation on all of our available machines (first four columns in Table III): For CPU, we used the Hybrid Read and Queue method and for GPU, we used pure GPU execution. We used two differently-sized graphs (16 million nodes with 128 million edges and 32 million nodes with 256 million edges) of both graph types (Uniform and RMAT).

We summarize the result of our experiment in Fig 11. The first four sets of columns in the figure display our measured performance on each system. The remaining two are performance values reported in the previous work of Agarwal et al's [18], which was said to outperform all previous results prior to it [12]–[14], [16].

We first compare the difference between the CPU architectures: Nehalam CPU, Core CPU, SC10-EP and SC10-EX. First, the performance difference between Core CPU and Nehalam CPU is noticeably large – more than 2x difference. We can attribute this performance gap to the difference in memory bandwidth of these machines (Table II) rather than cache size if we compare RMAT16 performance on Core CPU and RMAT32 performance on Nehalem CPU. Similarly, the performance gap between SC10-EP and SC10-EX is also closely correlated with the bandwidth difference between these machines, rather than the cache size difference (6x difference) or number of processors (4x difference).

Next we compare the difference between two GPU architectures: Tesla GPU and Fermi GPU. The Tesla GPU was unable to accommodate large graph instances. For smaller instances, Fermi GPU performed around 60% better than Tesla GPU, even though both have comparable memory bandwidth. [2] This improvement can be attributed to the shared last level (L2) cache, newly adopted in Fermi GPU, as we explain in the following experiment.

To better understand the GPU performance, we measured the difference in BFS performance on the Fermi GPU as we changed its cache configuration using the `-dlcm` compiler flag. We used graph instances with 32 million nodes and 256 million edges. The result is shown in Fig 12. Enabling the L2 cache alone yielded the best performance for BFS execution on the GPU, while disabling the cache resulted in performance comparable to Tesla GPU's. Interestingly,

[2]As a side note, we also observed that the negative effect of workload imbalance due to skewed degree distribution [19] was less severe in Fermi GPU than in Tesla GPU.

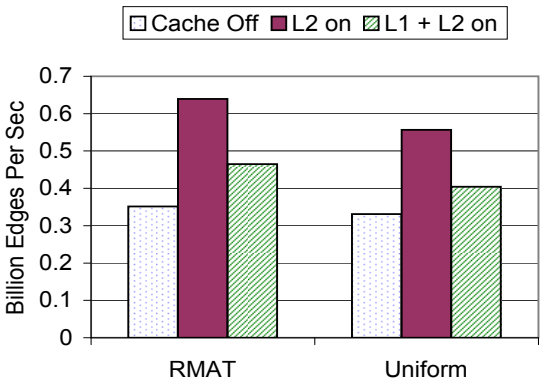


Fig. 12. Effect of GPU cache on BFS execution.

enabling L1 cache along with the L2 cache negatively affected performance, which could be due to an unrelated side-effect: disabling the L1 cache using `-ldcm` automatically reduces the transfer granularity from the L2 from 128 bytes to 32 bytes, thereby degrading bandwidth utilization. All our performance measurements for Fermi GPU in this paper used only the L2 cache.

Lastly, we compare the best GPU performance (Fermi GPU, single socket) and best CPU performance (SC10-EX, quad-socket). The performance between the two machine is quite comparable; although Fermi GPU still performs better, we expect that SC10-EP performance would increase if our methods were used on that machine [3].

We believe the final judgement between CPU systems and GPU systems would be individual since it should consider many other factors such as power consumption, cost of the system and the size of target graph instances as well as absolute performance number; each reader might suggest different weights on these factors.

VI. Related Works

Large graphs are drawing more attention due to the increasing number and importance of graph-based applications, such as social networking services. However, efficient processing of large graphs is still considered challenging [6]; one reason is the natural random memory access patterns exhibited in graph traversal. To accurately capture the computational requirements of large graph applications, a benchmark called Graph500 [11] has been created.

Much research has been conducted on efficient parallel implementations of BFS, targeting various computer architectures [12]–[19]. Among these works, we have extensively discussed the most recent work for multi-core CPU [18], and GPU [19] in Section III since they outperformed the others on those machines.

Although we focused on a level synchronous strategy for BFS (Section II), there are other studies that have used fixed-point strategies. Hassaan et al [25] compared fixed-point and level synchronous strategies and confirmed that level synchronous strategy allows for a sufficient degree of parallelism in BFS. Pearce et al [26] applied a fixed-point strategy for various graph algorithms including BFS, focusing on reducing synchronization overhead. Finally, Yoo et. al. [12] adopted a fixed-point strategy in order to implement BFS on a huge distributed system (64k cpu nodes) using a message passing library. Yoo et. al.'s performance on their distributed system was notably 5x less than on a four-socket multi-core system [18].

Distributed graph processing is considered to be challenging in general, due to the natural irregularity of the underlying graph [6]. Nevertheless, there is an important case in which distributed processing is mandatory: the graph does not fit in a single machine's memory. There are a few frameworks or libraries which aim to simplify graph processing in distributed envrionements. PBGL [27] is a message-passing implementation of the classic boost graph library. Pregel [28] is a distributed framework that encapsulates message passing and fault-tolerance in a similar manner to the MapReduce framework; traditional graph algorithms should be expressed in a description suitable for MapReduce, however.

There also has been developed supercomputers that are specially designed for graph processing [29], which features high memory bandwidth, huge memory capacity, and many cores that are heavily multi-threaded. Graph algorithms, including BFS, showed impressive performance on these machines [13], [16]. Unfortunately, such machines are rare and costly.

Some researchers [15], [19] used GPU to accelerate graph algorithms, because GPU shares many architectural properties of aforementioned supercomputers but is much economical. However the benefit of GPU execution is often limited by GPU's relatively small memory capacity. Our hybrid method utilizes GPU only when the graph size fits.

In this paper, we compared BFS performance across multiple CPU and GPU machines, studying their architectural effects. According to our observation, random memory access bandwidth was most critical to BFS performance. We refer the readers to recent papers regarding CPU vs. GPU debates [1], [30].

VII. Conclusion

In this paper, we propose new methods for parallel breadth-first search (BFS) implementations. Our multi-core CPU methodology is simple to apply yet efficient in utilizing memory bandwidth.Our method outperforms the state-of-the-art method by up to 45%, with the performance gap widening as the graph size grows. We also propose a hybrid method that dynamically chooses the best implementation for each BFS-level iteration–such a method benefits both large and small graphs while preventing poor worst case performance. Finally, our experiments showed that a single high-end GPU performs as well as a quad-socket high-end CPU system for BFS execution; the governing factor for performance was primarily random memory access bandwidth.

[3]We did not include multi-GPU systems in our experiment. However, considering the random access nature of this problem, the additional benefit of using a multi-GPU system is unclear.

Acknowledgements

This work is supported by DOE contract, Sandia order 942017; Army contract AHPCRC W911NF-07-2-0027-1; DARPA contract, Oracle order US103282; and Stanford PPL affiliates program, Pervasive Parallelism Lab: Oracle/Sun, NVIDIA, AMD, NEC, and Intel.

REFERENCES

[1] J. Nickolls and W. J. Dally, "The gpu computing era," *IEEE Micro*, vol. 30, no. 2, 2010.

[2] T. Coffman, S. Greenblatt, and S. Marcus, "Graph-based technologies for intelligence analysis," *Communications of the ACM*, vol. 47, no. 3, pp. 45–47, 2004.

[3] R. Sim and N. Roy, "Global a-optimal robot exploration in slam," in *Robotics and Automation, 2005. ICRA 2005. Proceedings of the 2005 IEEE International Conference on*, 2005, pp. 661–666.

[4] U. Brandes, "A faster algorithm for betweenness centrality," *The Journal of Mathematical Sociology*, vol. 25, no. 2, pp. 163–177, 2001.

[5] A. Tong, B. Drees, G. Nardelli, G. Bader, B. Brannetti, L. Castagnoli, M. Evangelista, S. Ferracuti, B. Nelson, S. Paoluzi *et al.*, "A combined experimental and computational strategy to define protein interaction networks for peptide recognition modules," *Science*, vol. 295, 2002.

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

[7] S. Skiena, *The algorithm design manual*. Springer, 1998, pp. 166–168.

[8] M. E. J. Newman and M. Girvan, "Finding and evaluating community structure in networks," *Physiscal Review E*, vol. 69, no. 2, 2004.

[9] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein, *Introduction to Algorithms*. MIT press and McGraw-Hill, 2001, pp. 651–665.

[10] D. Bader and K. Madduri, "Design and implementation of the hpcs graph analysis benchmark on symmetric multiprocessors," *High Performance Computing–HiPC 2005*, 2005.

[11] M. Anderson, "Better benchmarking for supercomputers," *Spectrum, IEEE*, vol. 48, no. 1, pp. 12–14, 2011.

[12] A. Yoo, E. Chow, K. Henderson, W. McLendon, B. Hendrickson, and U. Catalyurek, "A scalable distributed parallel breadth-first search algorithm on BlueGene/L," in *SC 2005 ACM/IEEE*.

[13] D. Bader and K. Madduri, "Designing multithreaded algorithms for breadth-first search and st-connectivity on the Cray MTA-2," in *ICPP 2006*. IEEE.

[14] D. Scarpazza, O. Villa, and F. Petrini, "Efficient breadth-first search on the cell/be processor," *IEEE Transactions on Parallel and Distributed Systems*, 2007.

[15] P. Harish and P. J. Narayanan, "Accelerating large graph algorithms on the gpu using cuda," in *HiPC*, vol. 4873. Springer, 2007.

[16] D. Chavarria-Miranda, A. Marquez, J. Nieplocha, K. Maschhoff, and C. Scherrer, "Early experience with out-of-core applications on the Cray XMT," in *IEEE IPDPS 2008*.

[17] D. Bader and K. Madduri, "Snap, small-world network analysis and partitioning: An open-source parallel graph framework for the exploration of large-scale networks," in *IEEE IPDPS*, 2008.

[18] V. Agarwal, F. Petrini, D. Pasetto, and D. Bader, "Scalable Graph Exploration on Multicore Processors," in *Supercomputing. Proceedings of the ACM/IEEE SC 2010 Conference*.

[19] S. Hong, S. Kim, T. Oguntebi, and K. Olukotun, "Accelerating CUDA graph algorithms at maximum warp," in *Proceedings of the 16th ACM symposium on Principles and practice of parallel programming*, 2011.

[20] Intel, "Intel microarchitecture codenamed nehalem," http://www.intel.com/technology/architecture-silicon/next-gen/index.htm?iid=nehalem.

[21] Nvidia, "Next generation cuda architecture, code named fermi," http://www.nvidia.com/object/fermi_architecture.html.

[22] D. Watts and S. Strogatz, "Collective dynamics of small-world networks," *Nature*, vol. 393, no. 6684, 1998.

[23] D. Chakrabarti, Y. Zhan, and C. Faloutsos, "R-mat: A recursive model for graph mining," in *SDM*, 2004.

[24] D. A. Bader and K. Madduri, "Snap: small-world network analysis and partitioning," http://snap-graph.sourceforge.net.

[25] M. Hassaan, M. Burtscher, and K. Pingali, "Ordered vs. unordered: a comparison of parallelism and work-efficiency in irregular algorithms," in *PPoPP*. ACM, 2011.

[26] R. Pearce, M. Gokhale, and N. Amato, "Multithreaded asynchronous graph traversal for in-memory and semi-external memory," in *Proceedings of the 2010 ACM/IEEE International Conference for High Performance Computing, Networking, Storage and Analysis*, 2010.

[27] D. Gregor and A. Lumsdaine, "The parallel bgl: A generic library for distributed graph computations," *Parallel Object-Oriented Scientific Computing (POOSC)*, 2005.

[28] G. Malewicz, M. H. Austern, A. J. Bik, J. C. Dehnert, I. Horn, N. Leiser, and G. Czajkowski, "Pregel: a system for large-scale graph processing," in *SIGMOD '10*. ACM.

[29] Cray, Inc., "Cray xmt," http://www.cray.com/products/xmt/.

[30] V. W. Lee, C. Kim, J. Chhugani, M. Deisher, D. Kim, A. D. Nguyen, N. Satish, M. Smelyanskiy, S. Chennupaty, P. Hammarlund, R. Singhal, and P. Dubey, "Debunking the 100x gpu vs. cpu myth: an evaluation of throughput computing on cpu and gpu," in *ISCA'10*.