Work Efficient Parallel Algorithms for Large Graph Exploration

Dip Sankar Banerjee, Shashank Sharma, Kishore Kothapalli
International Institute of Information Technology, Hyderabad
Gachibowli, Hyderabad, India 500 032.
{dipsankar.banerjee@research., shashank.sharma@research., kkishore@}iiit.ac.in

***Abstract*—Graph algorithms play a prominent role in several fields of sciences and engineering. Notable among them are graph traversal, finding the connected components of a graph, and computing shortest paths. There are several efficient implementations of the above problems on a variety of modern multiprocessor architectures.**

It can be noticed in recent times that the size of the graphs that correspond to real world data sets has been increasing. Parallelism offers only a limited succor to this situation as current parallel architectures have severe short-comings when deployed for most graph algorithms. At the same time, these graphs are also getting very sparse in nature. This calls for particular work efficient solutions aimed at processing large, sparse graphs on modern parallel architectures.

In this paper, we introduce graph pruning as a technique that aims to reduce the size of the graph. Certain elements of the graph can be pruned depending on the nature of the computation. Once a solution is obtained for the pruned graph, the solution is extended to the entire graph.

We apply the above technique on three fundamental graph algorithms: breadth first search (BFS), Connected Components (CC), and All Pairs Shortest Paths (APSP). To validate our technique, we implement our algorithms on a heterogeneous platform consisting of a multicore CPU and a GPU. On this platform, we achieve an average of 35% improvement compared to state-of-the-art solutions. Such an improvement has the potential to speed up other applications that rely on these algorithms.

I. Introduction

Graph algorithms find a large number of applications in engineering and scientific domains. Prominent examples include solving problems arising in VLSI layouts, phylogeny reconstructions, data mining, image processing, and the like. Some of the most commonly used graph algorithms are graph exploration algorithms such as Breadth First Search (BFS), finding the connected components, and computing shortest paths. As the current real life problems often involve the analysis of massive graphs, it is often seen that parallel solutions provide an acceptable recourse.

Parallel computing on graphs however is often very challenging because of their irregular nature of memory accesses. This irregular nature of memory access also stresses the I/O system of most modern parallel architectures. It is therefore not surprising that most of the recent progress in scalable parallel graph algorithms is aimed at addressing these challenges via innovative use of data structures, memory layouts, and SIMD optimizations [21], [13], [23]. Recent results have been able to make efficient use of modern parallel architectures such as the Cell BE [23], GPUs [21], [14], [13], Intel multi-core architectures [7], [30], [2] and the like. Algorithms running on GPUs have shown standout performance amongst these because of their massive parallelism.

Heterogeneous algorithms that aim to utilize all the computational devices in a commodity heterogeneous platform have also been designed for graph breadth-first exploration [14], [21], [11]. Most of these use platforms consisting of multicore CPUs and GPUs. All of the above-cited works show an average of 2x improvement over pure GPU algorithms.

Most of the above works in general aim at data structure optimizations but largely run classical algorithms on the entire input graph. These algorithms are designed for general graphs whereas the current generation graphs possess markedly distinguishable features such as being large, sparse, and large deviation in the node degrees. In Figure 1, we show some of the real-world graphs taken from [1]. These graphs are commonly constructed from real life systems or phenomenon. For example the web-Google graph represents the Google web graph where each of the nodes are web pages and the edges between them are the hyper-links between them. As can be seen from Figure 1, these graphs have several nodes of very low degree, often as low as 1. For instance, in the case of the graph web-Google, 14% of the nodes have degree 1. Table I lists other properties of a few real world graphs from [1].

Current parallel algorithms and their implementations [21], [11], [23], [14], [28] do not take advantage of the above properties. For instance, in a typical implementation of the breadth-first search algorithm, one uses a queue to store the nodes that have to be explored next. But, a node v of degree 1 that is in the queue will not lead to the discovery of any yet undiscovered nodes. So, the actions of BFS with respect to v such as adding it to the queue, dequeue it, and then realize that there are no new nodes that can be discovered through node v are all unnecessary. These actions unfortunately can be quite expensive on most modern parallel architectures as one has to take into account the fact that the queue is to be accessed concurrently. Similarly, other operations such as checking of the status of a node, may be quite disposable.

In light of the above paragraph, we envisage that new algorithms and implementation strategies are required for efficient processing of current generation graphs on modern multicore architectures. Such strategies should help algorithms and their implementations benefit from the properties of the graphs. In this paper, we propose graph pruning as a technique

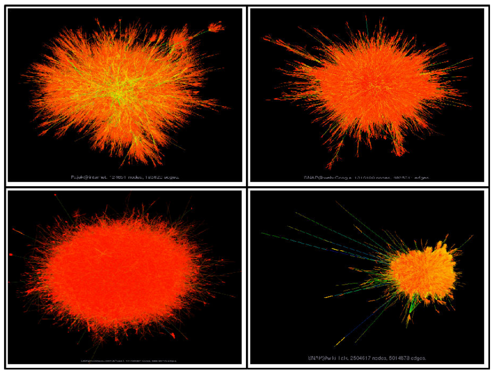


Fig. 1. A sample of four real world graphs from [1]. On the top-left corner is the graph internet, top-right is the graph web-google, bottom left is the graph webbase_1M, and the bottom-right is the graph wiki-Talk.

in this direction. Graph pruning aims to reduce the size of the graph by pruning away certain elements of the graph. The required computation is then performed on the remaining graph. The result of this computation is then extended to the pruned elements, if necessary.

In this paper, we apply the graph pruning technique to three important graph algorithms: Breadth-first search, connected components, and All-pairs-shortest-paths (APSP). In each case, we show that pruning pendant nodes iteratively can result in reducing the size of the graph on real-world datasets, by as much as 25% in some cases. This reduction in size helps us achieve remarkable improvements in speed for the above three workloads by an average of 35%.

A. Related Work

Many recent works in parallel computing have focused on graph algorithms. Few among them include [7], [14], [29], [21]. The work of Scarppaza et al. [23] demonstrates the use of an all-to-all exchange of visited nodes information across the eight SPUs of a Cell BE. One of the first results of BFS using GPUs is the work of Harish et al. [13]. Subsequent improvements to [13] centered around the use of heterogeneous computing. In [14], Olukotun et al. use a CPU+GPU platform where the levels of the BFS with fewer discovered nodes are processed on the CPU and levels with large number of discovered nodes are processed on the GPU. Using such a heterogeneous strategy, they achieve a throughput of 0.4 Beps (Billion edges per second) on Erdos-Renyi random graphs. These are improved further by Bader et al. [21]. Some of the prominent works on multicore CPUs include [7] where the primary goal is to map the data structures to the cache hierarchy so as to improve cache hit rates. A recent work [11] partitions the graph so that low degree nodes are processed on the GPU and the high degree nodes are processed on the CPU.

Finding the connected components of a graph also is an important primitive and hence has attracted a lot of attention within the parallel computing community. Popular parallel algorithms in the PRAM model include the algorithm of Shiloach and Vishkin [25] and its variants by Greiner [12]. On GPUs, a variant of Shiloach and Vishkin algorithm [25] is used by Soman et al. [26]. A heterogeneous execution of this algorithm on a CPU+GPU platform with an improvement of 35% on average is shown in [6].

The all-pairs-shortest-paths (APSP) problem is yet another fundamental graph algorithm with several applications. One of the earliest works on parallel shortest path problem was proposed by Micikevicius et al. in [20]. In this work, the author proposed a parallel implementation of the popular Floyd-Warshall algorithm on an early generation FX5900 GPU which showed speedups up to 3x over sequential CPU code. In a more recent work [27], Venkatraman et al. proposed a blocked parallel implementation of the APSP problem. Here, the authors showed a more cache efficient algorithm of the problem that utilizes the cache hierarchies present in the CPUs to provide a 1.6x to 1.9x speedup. In [17], the authors show a new APSP implementation on the GPU where the authors utilizes the available shared memory efficiently on a G80 GPU. They employ a transitive closure based technique where they adapt the Floyd-Warshall algorithm across single and multiple GPUs. This is the currently known best result of the APSP problem on GPUs. Matsumoto et al. propose an hybrid APSP work based on the work in [27] in [18]. Here the authors used a block based structure for the minimization of communication overheads between CPU and GPU and is hence more efficient. However, the work does not experiment on massive graphs.

Algorithm or implementation decisions based on the nature of the graph is an emerging area of research. In [10], the authors propose a Distributed Leaf Pruning (DLP) strategy that helps in achieving a significant speedup over distributed communication networks. In this work, the authors noticed that in many real life networks, like CAIDA, the average node degree of a graph with n nodes is very less than n and the number of nodes with a unitary degree is typically high. So, pruning these nodes from the graph, provided a much better performance in packet forwarding strategies over the entire network.

In [22], the authors show novel pruning techniques that solves the maximum clique problem on large sparse graphs. Their main idea is to prune the nodes that strictly have fewer neighbors than the size of the maximum clique already computed. These are nodes that can be exempted from the computation as, even if a new clique is found, its size would not be greater than the maximum one that is already computed.

Input pruning has been used as a technique in the design of *work-optimal* parallel algorithms in the PRAM model. Popular examples include the list ranking algorithm of Anderson and Miller [3], the optimal merging algorithm [8], the optimal range minima algorithm [24], and so on. In all of these cases, the size of the input is reduced to an extent after which a slightly non-optimal algorithm is employed. In a post-processing phase, the results on the reduced input is extended to obtain a result for the entire input.

B. Our Results

In this paper, we focus on graph BFS, connected components, and all-pairs-shortest-paths. For these three graph algorithms, we first show that a similar preprocessing phase can help reduce the size of the graph by an average of 25%

on a wide variety of real-world graphs. This helps us to obtain an average of 40% speed-up compared to the best known implementations for the above problems on similar platforms.

Our preprocessing simply involves removing pendant nodes from the graph. This is done iteratively so that nodes on pendant paths are also removed during preprocessing. In the post-processing phase, we show that extending the output of the computation on the smaller graph can be done in a very straight-forward and quick manner.

Some of our specific contributions are as follows:

- Our results improve the state-of-the-art for graph BFS by 35%. We achieve an average throughput of 2 billion edges per second on a wide range of data sets including graphs from the University of Florida collection [1], and graphs generated using the Recursive Matrix Model (R-MAT). The R-MAT generator is efficiently implemented in the GTGraph suite[5].
- On the connected components problem, we get an average 20% improvement over the best known result on an identical platform [6]. A small change to the algorithm can also build a spanning tree of a graph with very little extra time.
- For computing the shortest path between all pairs of nodes, we achieve an average of 44% improvement compared to the best known result of [17] on a similar platform.

II. A Brief Overview of our Experimental Platform

In this section, we briefly describe our hybrid computing platform. Our hybrid platform is a coupling of the two devices described above, the Intel i7 980 and the Nvidia GTX 580 GPU. The CPU and the GPU are connected via a PCI Express version 2.0 link. This link supports a data transfer bandwidth of 8 GB/s between the CPU and the GPU. To program the GPU we use the CUDA API Version 4.1. The CUDA API Version 4.1 supports asynchronous concurrent execution model so that a GPU kernel call does not block the CPU thread that issued this call. This also means that execution of CPU threads can overlap a GPU kernel execution.

The GTX 580 GPU is a current generation Fermi micro-architecture from NVidia that has 16 symmetric multi-processors (SM) with each SM having 32 cores for a total of 512 compute cores. Each compute core is clocked at 1.54 GHz. Each SM has a hardware scheduler which schedules 32 threads at a time. This group is called a warp and a half-warp is a group of 16 threads that execute in a SIMD fashion. Each of the cores of the GPU now has a fully cached memory access via an L2 cache, 768 KB in size. In all, the GTX 580 has a peak single precision performance of 1.5 TFLOPS.

Along with the GTX 580, we use an Intel i7 980x processor as the host device. The 980x is based on the Intel Westmere micro-architecture. This processor has each core running at 3.4 GHz and with a thermal design power of 130 W. The i7-980X has six cores and with active SMT (hyper-threading) can handle twelve logical threads. The L3 cache has is of 12 MB and L2 is of 256 KB in size. Other features of the Core i7 980 include a 32 KB instruction and a 32 KB L1 data cache per core and the L3 cache is shared by all 6 cores.

III. Our Approach

In this section, we present a three phase technique, outilned in Algorithm 1, for scalable parallel graph algorithms of real world graphs. In the first phase, called the *preprocessing phase*, we reduce the size of the input graph by removing *redundant* elements of the graph. Once the graph size reduces, the second phase involves using existing algorithms to perform the computation on the smaller graph. In a final phase, we then extend the result of the computation to the entire original graph via quick post-processing, if required.

Let G be a large, sparse graph. As mentioned in Algorithm 1, let Prune(G) be a function that can prune certain elements of G. Let G' be the graph that remains after Prune(G). Let A be an algorithm that can compute the desired solution. We then use algorithm A on the graph G'. Let O' be the solution on G'. In a post-processing third phase, we extend the solution O' to a solution O of the entire graph G.

Algorithm 1 $ProcessGraph(Graph\ (\mathcal{V}, \mathcal{E}))$

```
1: /* Phase I – Prune */
2: $G' = \text{Prune}(G)$
3: /* Phase II – Compute */
4: $O' = A(G')$
5: /* Phase III – Extend */
6: $O = \text{Extend}(G, O')$
```

We note that if Phase I prunes only a constant fraction of the size of the graph, and one uses a standard algorithm in Phase II, then the asymptotic runtime using the above technique is still unchanged. However, even such a constant fraction reduction in size can have a considerable impact on the experimental efficacy.

We envisage that different graph algorithms can benefit from corresponding pruning processes in Phase I. Further, step 1 may also be performed iteratively. Each iteration may prune some nodes after which more nodes may become candidates for pruning in the next iteration. We refer the reader to Algorithm 2 for an illustration. In Algorithm 2, $\mathcal{P}$ refers to a property that nodes that are pruned will satisfy. Similarly, the post-processing in Phase III can also be based on the problem at hand. If Phase I is spread over multiple iterations, then Phase III may also be spread over multiple iterations, possibly in the reverse order of iterations of Phase I.

Algorithm 2 $Prune(Graph\ (\mathcal{V}, \mathcal{E}))$

```
1: for $i = 1\ to\ r$ iterations do
2:     for each node $v \in \mathcal{G}$ do
3:         if $v$ has property $\mathcal{P}$ then
4:             Remove $v$, and all edges incident on $v$.
5:             Store $(v, i)$ for future re-insertion step.
6:         endif
7:     endfor
8: endfor
```

It is important to note that the property $\mathcal{P}$ can be evaluated quickly. This helps keep the overall time for Phase I small. The time taken by a graph algorithm using our technique will depend on the extent of pruning achieved in Phase I and also the time taken in Phase I and III. As can be noticed, in most cases, there will also be a trade-off between time taken by Phase I and III and that of Phase II. In fact, such a trade-off is observed in the case of list ranking [6].

Some of the properties that may be of interest are the following.

- Pendant nodes: Let us call a node v in a graph G as a pendant node if the degree of v is G is 1. For the three workloads we consider in this paper, we show that a simple pruning based on removal of pendant nodes suffices. This is also the pruning technique used in [10].
- Independent nodes: A subset of nodes is called as an independent set of nodes if they are mutual non-neighbors. This has been used in list ranking algorithms [3] and its recent heterogeneous implementation [29], [6].
- Graph partitioning: Graph partitioning calls for partitioning a graph G into a specified number k equal partitions such that the number of edges that have end points in different partitions is minimized. In an influential work, Karypis and Kumar [16], introduce the coarsening-refinement approach. During the coarsening step, a matching of the current graph is computed and prunes matched nodes.

The above examples indicate that various properties $\mathcal{P}$ can have applicability to different problems. Thus, our approach is quite general. It must be noted that all the above examples are not implemented on modern parallel architectures. In this paper, we show that our technique can be used on modern parallel architectures too.

IV. Breadth First Search

Breadth First Search (BFS), is one of the most widely used graph algorithms and finds massive applications in the domains of state space partitioning, graph partitioning, theorem proving, and networks. The problem statement of the BFS is: given an undirected, unweighted graph $\mathcal{G}(\mathcal{V}, \mathcal{E})$, and a source node $\mathcal{S}$, compute the minimum number of edges that are needed to reach every node of $\mathcal{G}$ from $\mathcal{S}$. The optimal sequential solution to this problem runs in $\mathcal{O}(\mathcal{V} + \mathcal{E})$ time [9].

The well known sequential algorithm maintains a queue where the newly discovered nodes are are inserted at the rear. Current nodes are deleted from the front of the queue and this process continues until the queue is depleted. All the newly visited nodes are constantly enqueued along the way. For representation of the graph in the memory, we use the compact adjacency list which is more popularly known as the compact sparse representation (CSR) [19]. An example is shown in Figure 2. In CSR, all the adjacency lists are packed into a single large array. An array E_a is used to store the adjacency lists where the list for node $i+1$ immediately follows node i, for all the nodes in $\mathcal{G}$. An array V_a, stores the starting indices of the corresponding adjacency lists in E_a. Each of the indices of V_a acts as the node number of the graph. The key advantage

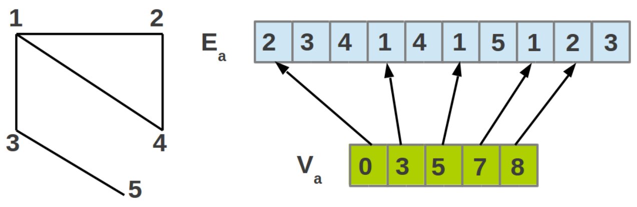


Fig. 2. The CSR format for representation.

of using this representation is that the graph is stored in a contiguous memory locations and no long strides are required to go from a neighbor of a certain node. This helps in reducing the memory access irregularity and hence boosts the overall performance of the BFS implementation.

A. Implementation

The basic approach of our algorithm is to first perform a pruning step where pendant nodes are removed iteratively. This is followed by an efficient parallel execution of BFS on the CPU+GPU hybrid platform from Section II. Finally, in a post-processing step, the level number for nodes that were removed initially will be computed. Our algorithm is described in Algorithm 3.

Algorithm 3 $BFS(Graph\ (\mathcal{V}, \mathcal{E})), VertexS$

```
1: Call Algorithm 2
2: Perform BFS in hybrid on GPU and CPU (See Algorithm 4)
3: for $i = r$ to 1 repeat
4:     Re-insert removed nodes according to $(i, v)$ information previously stored.
```

a) Phase I:: The first phase of removing the pendant nodes is done entirely on the GPU as it is a purely parallel step with no irregular memory operations involved. Hence, a hybrid implementation of this step puts an unnecessary overhead of data transfer. We however add the following optimizations.

- To reduce the time spent in Phase I, we use the CSR representation and identify pendant nodes as follows. Consider nodes u and v numbered consecutively. Then, u is a pendant node if $V_a[u]$ and $V_a[v]$ differ by 1. If node u is numbered n, then the above rule has to be modified to say that $V_a[n] = |E_a|$. See Figure 2 for an illustration where node 5 is a pendant node, and also $V_a[5] = 10$. In essence, threads need not read the E_a array, and also do not have uncoalesced accesses.
- Notice that if a node v that is removed in iteration i, then its only neighbor can now become a pendant node in iteration $i+1$. Further, in iteration $i+1$, we need to check only such nodes w. Therefore, we mark such w in iteration i, and do not check other nodes in iteration $i+1$. This helps in reducing the time spent in Phase I across each iteration. For illustration, see Figure 2. If node 5 is removed in the first iteration, then node 3 is marked as a potential pendant node in that iteration. Since the

remaining degree of node 3 is now 1, also node 3 can be removed in the second iteration.

Algorithm 4 $PhaseII(Graph\ (\mathcal{V},\mathcal{E}), Vertex\ S)$

```
Input: A graph $\mathcal{G}(\mathcal{V},\mathcal{E})$ and starting node $\mathcal{S}$
Output: Level numbers of all the nodes.
1: Set a threshold for separating the node set between CPU and GPU.
2: Create the graph $G_c$ for the CPU and the graph $G_g$ for GPU.
3: Create initial FR array with $S$
4: while FR ! $= \phi$
5:     GPU :: Call GPU_BFS$((\mathcal{G}',\mathcal{S}), Array\ FR)$ (Algorithm 5)
6:     CPU :: Perform CPU BFS [9].
7:     Check NFR array of GPU and CPU for termination
8:     Set FR := NFR
9: endwhile
10: Consolidate LEVEL values
```

Algorithm 5 $GPU\_BFS\ (Graph\ (\mathcal{V},\mathcal{E}),\ Vertex\ S,\ FR)$

```
Input: A graph $\mathcal{G}(\mathcal{V},\mathcal{E})$ and starting node $\mathcal{S}$
Output: Level numbers of all the nodes.
1: tid=threadID
2: Initialize LEVEL[S]=1;
3: Set NFR to NULL
4: Calculate $range$ in FR based on size of $E_g$ and threads
5: Find $start$ and $stop$ of index of nodes from $range$
6: for $i\ = start\ to\ stop$
7:      VISITED[i]=1
8:     for i $\in$ all neighbors of $i$ do
9:         if(VISITED[i] == FALSE)
10:            LEVEL[i]=LEVEL[i]+1;
11:            Add $i$ to NFR;
12:        endif
13:    endfor
14: endfor
```

Graph	Nodes	Edges	Pendant Nodes	r
UF Sparse Matrix Collection				
internet	124,651	207,214	56,959 **(45%)**	2
dblp_2010	326,183	807,700	87,881 **(26%)**	3
watson_1	386,992	1,055,093	44,637 **(11%)**	2
webbase_1M	1,000,005	3,105,536	80,053 **(8%)**	4
wiki-Talk	2,394,385	5,021,410	176,617 **(7.37%)**	4
web-Google	916,428	5,105,309	134,452 **(14.6%)**	3
rail2586	923,269	8,011,362	117,342 **(12.7%)**	4
tp-6	1,014,301	11,537,419	194,764 **(19%)**	4
R-MAT Graphs				
rmat_1	131,072	256,784	43,556 **(33.2%)**	1
rmat_2	263,144	554,678	67,345 **(25.6%)**	2
rmat_3	525,288	1,048,515	55,453 **(10.5%)**	2
rmat_4	1,056,534	2,097,345	78,234 **(7.4%)**	3
rmat_5	2,045,266	4,190,223	92,345 **(4.5%)**	4
rmat_6	3,074,344	8,456,240	105,117 **(3.4%)**	3
rmat_7	3,156,834	12,673,552	132,443 **(4.2%)**	4
rmat_8	3,765,223	16,673,993	153,442 **(4.1%)**	4

TABLE I
THE GRAPHS USED FOR EXPERIMENTATIONS AND THEIR PROPERTIES. THE COLUMN HEADING r IN THE LAST COLUMN INDICATES THE NUMBER OF ITERATIONS REQUIRED TO REMOVE ALL PENDANT NODES.

b) Phase II: We now present our detailed algorithm in Algorithm 5 that is used in Phase II. Algorithm 4 is similar in spirit to the one used by Munguia et al. [21]. The label CPU:: and GPU:: in Algorithm 4 refer to steps executed on the CPU and the GPU respectively. The CPU and the GPU maintain array VISITED, FR, and NFR which contain the visited nodes, the current frontier nodes, and the next frontier nodes, respectively. The array VISITED is shared between the two devices so that the status of a node can be polled whenever required. The array LEVEL is used to store the number of edges in a shortest path from s to every other node in the graph. This array is maintained locally at both the CPU and the GPU and is merged at the end of each iteration. An iteration corresponds to exploring all the nodes in the current frontier, given by the FR array. Once either the CPU or the GPU checks that the NFR array that it is maintaining is already explored by the other device, the execution stops and the LEVEL array is transferred from the GPU to the CPU.

The entire algorithm executes until the current FR array is not empty, that is all the nodes has been visited. To this end, both the CPU and the GPU work in a synchronous manner to perform the exploration. The GPU does a thread based partitioning of the adjacency list E_a. The CPU on the other hand does a more coarse grained execution on the E_c portion using the same algorithm using all the threads available to it with simultaneous multithreading. The GPU BFS part maintains a frontier array FR, which is the queue where it continually deletes elements from and also maintains a NFR which is the next frontier to be visited. Both the CPU and the GPU maintains this NFR information locally so as to minimize the communication overheads. To further minimize the cost of communication, we transfer the NFR in an asynchronous

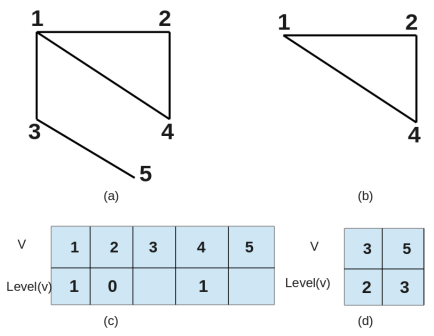


Fig. 3. An example run of our algorithm on the graph in part (a). Part (b) is the graph obtained after removing pendant nodes, (c) shows the result of Phase II, and (d) shows the result of Phase III.

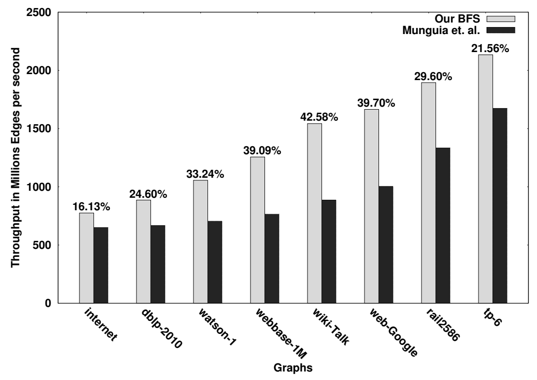


Fig. 4. Performance of BFS on the UF Sparse Matrix dataset. The numbers show the percentage improvement.

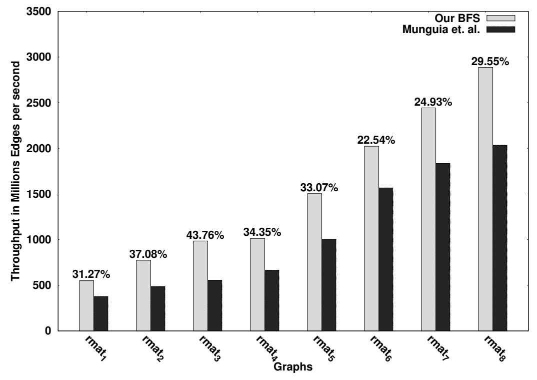


Fig. 5. Performance of BFS on the R-MAT dataset. The numbers show the percentage improvement.

manner so that maximum amount of overlap can be achieved with the communication and the devices stay idle for the minimum amount of time. After each iteration, both the CPU and the GPU communicates this NFR array and sets it as its current FR if it has not been already visited by the other device. When the BFS algorithm on both CPU and the GPU terminates, the two devices consolidates their LEVEL array.

c) Phase III: In Phase III of the algorithm, we re-insert the nodes that were removed in Phase I as follows. Let v be a node removed in iteration i of Phase I, and let w be the only neighbor of v prior to its removal. Then, the LEVEL number of v is set to be one more than the LEVEL number of w. Further, nodes removed in iteration i are processed before those removed in iteration $i-1$.

An example run of our algorithm is presented in Figure 3 for the graph from Figure 2.

B. Results

In this section, we present the results of our implementation. We compare the results with those of [21] which are the currently reported best results for graph breadth first search on identical platforms.

In Figure 4, we show the results of our implementation on a sample of eight real world graphs from the University of Florida dataset [1]. As shown in Figure 4, our results outperform the results of [21]. Similar effect is seen also on random graphs generated using the R-MAT generator [5] as shown in Figure 5. These graphs are generated with the default values for $a = 0.45$, $b = 0.15$, $c = 0.15$, and $d = 0.25$ of R-MAT as they represent many real world graphs. All the experiments were carried out on the same platform using the code that was obtained from the respective authors of the papers. Also, the results reported are averaged over multiple executions.

To analyze the results we obtain we perform two further experiments. We study the percentage improvement of our implementation as a function of the percentage of nodes that Phase I can remove. The results of this experiment are shown in Figure 6 for the graphs webbase_1M, wiki-Talk and web-Google from the dataset of [1]. Figure 6 indicates that significant performance gains can be achieved even as a small percentage of nodes are pruned from the input graph. The improvement can be attributed to the lesser number of operations required in our implementation as pruned nodes do not enter/exit arrays FR, VISITED, and NFR.

We also experimented on the trade-off between the number of iterations of Phase I and the overall runtime. As an example, consider that the graph webbase_1M from the [1] dataset. Notice from Figure 7 that in the sixth iteration of Phase I, only 134 pendant nodes can be removed. The question we seek to answer is whether one should run one more iteration of Phase I to remove these 134 nodes, or move on to Phase II.

Figure 7 shows the results of the above experiment on the webbase_1M graph from the [1] dataset. The time for Phase I is shown on the right-side Y-axis of Figure 7. It can be noticed that the time taken for each iteration of Phase I does decrease. This is attributed to the fact that successive iterations do not test each node whether it can be removed. Rather, we flag potential pendant nodes in an iteration so that only those nodes are checked in the next iteration. The number of nodes that are removed in each of the iterations are plotted over the Phase I curve. The overall runtime decreases initially as we expect to remove more pendant nodes in the first few iterations of Phase I. The important observation we can note from Figure 7 is that the overall runtime plateaus after five iterations of Phase I for the graph considered. Therefore, it is worthwhile to stop Phase I once the number of remaining pendant nodes reduces below a small threshold of say 500.

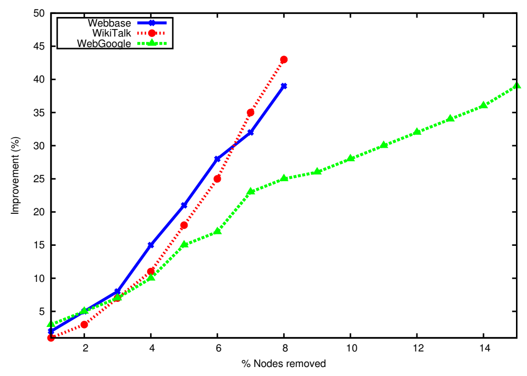


Fig. 6. Percentage improvement of performance of BFS as a function of the percentage of pendant nodes removed in Phase I. The graphs webbase_1M and wiki-Talk have under 8% pendant nodes.

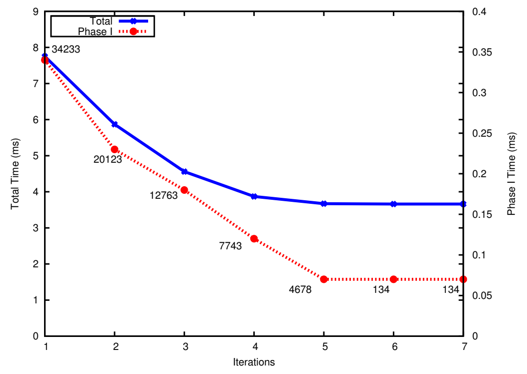


Fig. 7. Trade-off between Phase I and the overall runtime of BFS.

V. Connected Components

Finding the connected components of a graph is of fundamental importance to graph algorithms. Given a graph $G = (V, E)$, the problem is to find a partitioning of V into disjoint sets $V_1, V_2, \cdots$, so that nodes u and v are in the same set if and only if there is a path between u and v in G. Well known sequential algorithms such as the Depth First Search algorithm (DFS) [9] run in $O(n + m)$ time. Several efficient parallel algorithms in the PRAM model have been proposed. Popular among them are the algorithms of Shiloach and Vishkin [25], and the algorithm of Greiner et al. [12]. However, because of the irregular nature of operations involved, this workload is often difficult to implement on most modern parallel architectures. Efficient implementations of the Shiloach and Vishkin algorithm are known to exist for a variety of parallel architectures including symmetric multiprocessors [4], Cray and CM2 [12], GPUs [26], and also on CPU+GPU systems [6].

In this section, we apply techniques from Section III and show that highly efficient hybrid algorithms can be designed for this workload on the CPU+GPU hybrid platform described in Section II. Our solution can be broadly outlined in the following steps and follows the algorithm used in [6].

A. Implementation

The main steps of our implementation is outlined in Algorithm 6. In Step 1 of Algorithm 6, we *prune* the pendant nodes in the given graph iteratively. This is done by the sequence of steps as outlined in Algorithm 2.

Algorithm 6 *Connected_Components*$(Graph\ (\mathcal{V}, \mathcal{E}))$

```
Input: A graph $\mathcal{G}(\mathcal{V}, \mathcal{E})$. Here $|V| = n$.
Output: Labels of each node identifying its component.
1: Call Algorithm 2
2: Initialize LEVEL[S]=1;
3: Set a threshold t.
4: CPU :: $n_{cpu} = \frac{nt}{100}$
5: CPU :: Partition $E$ into $E_{cpu}$ and $E_{gpu}$ where $E_{cpu}$ is
   the edges corresponding to $n_{cpu}$ nodes and $E_{gpu}$ is the
   rest.
6: Find connected components of the graph $G(V - n_{cpu}, E_{gpu})$ on the GPU using the Shiloach-Vishkin
   algorithm [25], and the connected components of
   $G(n_{cpu}, E_{cpu})$ on the CPU using DFS. The graph $G[n_{cpu}]$
   is further divided into $c$ equal partitions where $c$ is the
   number of threads run on the CPU.
7: Use the cross-edges recorded during the partition phase to
   compute the final components.
8: Re-insert the edges removed in Step 1.
```

In Steps 3–5, we partition the graph according to a pre-determined threshold t. The optimal value of t is later on determined experimentally. We divide the edges of the graph so that the first $t\%$ is in one partition and the rest in the other. We allocate the smaller partition to the CPU and the other to the GPU. The partitioning strategy that we follow is can be defined as a case of MIMD data parallelism where different functions are applied to different data sets. This is because the algorithm that is well suited for the CPU can be different from that of the GPU. We use the GPU friendly Shiloach-Vishkin (SV) algorithm [25] for the GPU computation and DFS on the individual CPU cores.

GPU and CPU Optimizations : The Shiloach–Vishkin [25] algorithm is a well suited algorithm for parallel implementation. However, we need to make some additional optimizations in order to make it more efficient for the hybrid platform. Some of the major bottlenecks that we address are that of the atomics, memory latencies and reducing divergence. Towards implementing these modifications for the GPU, we perform the Shiloach-Vishkin algorithm in three steps. In the first step, called the *hooking* step, we hook subtrees of the graph whose root has a lower label to a tree with a higher label whenever there is an edge uv such that u and v are in different subtrees. In the second step, we do *pointer jumping* where by we shrink the existing trees to a rooted star so that the nodes are present only at two levels: either at the root level or at the leaf level. In the final step we perform *edge hiding*. In this step, we stop processing the edges of the smaller sub trees once their hooking step is over. This reduces the thread divergence to a great extent and also reduces data movement. More details of these optimizations are described in [26].

For finding the connected components of the graph G_i for $1 \le i \le c$ on the ith core of the CPU, we use the standard DFS algorithm [9]. This is motivated by the fact that since

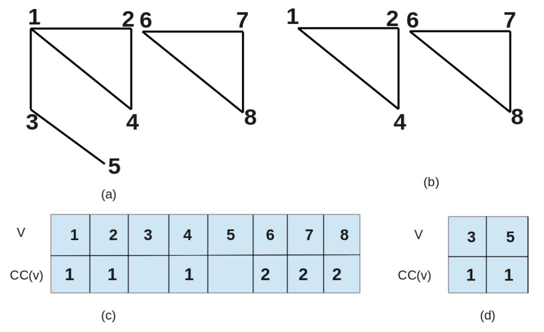


Fig. 8. An example run of our algorithm on the graph in part (a). Part (b) is the graph obtained after removing pendant nodes, (c) shows the result of Phase II, and (d) shows the result of Phase III.

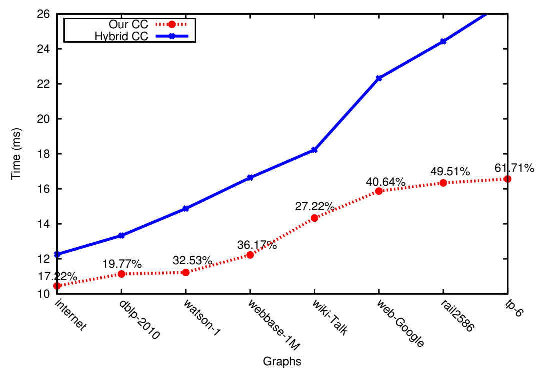


Fig. 9. Performance of Connected Components the UF Sparse Matrix dataset [1]. The numbers show the percentage improvement.

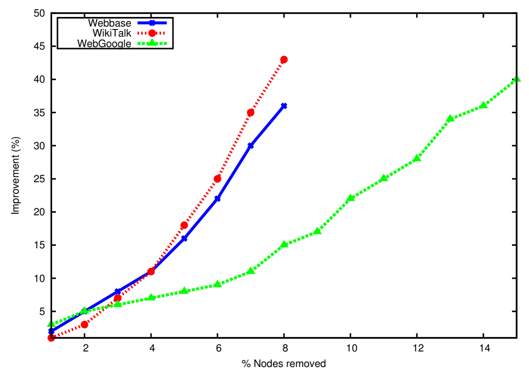


Fig. 10. Performance improvement of Connected Components with the removal of pendant nodes.

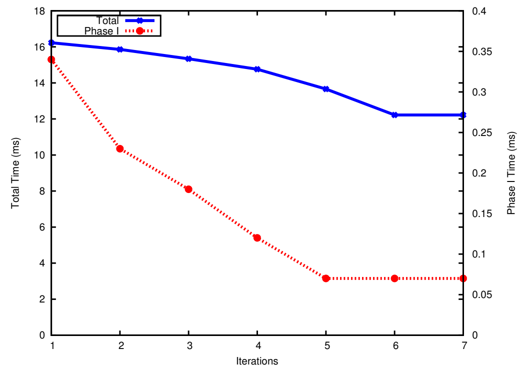


Fig. 11. Trade-off between Phase I and the total time.

the available parallelism on the CPU is small, highly data parallel algorithms do not make a good fit. Further, each CPU core can run independently minimizing any overheads in synchronization and communication. The output of this step is that each CPU core labels the components identified uniquely.

In the final step, Step of Algorithm 6, the following post-processing is done. For a pendant node v removed in the ith iteration, let w be the only neighbor of v. Then the node v is said to belong to the component that w belongs to. In the above, we process nodes in the opposite order of their removal in Step 1. An example run of our algorithm is presented in Figure 8 for the graph from Figure 2.

B. Results

In this section, we present the results of our implementation. We compare the results with those of [6]. The results of [6] are currently the best reported results for finding the connected components of a given graph on a CPU+GPU platform.

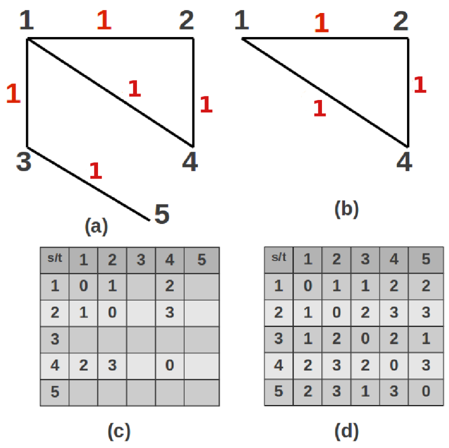
In Figure 9, we show the results of our implementation on a sample of eight real world graphs from the University of Florida dataset [1]. We see that on an average we achieve a speedup of around 42%. The improvement we obtain can be attributed to the fact that the pointer jumping operation in the Shiloach-Vishkin algorithm is a highly irregular operation, and hence the fewer such operations the better.

Similar to the BFS implementation, we also study the percentage improvement of our implementation as a function of the percentage of nodes that Phase I can remove. The results of this experiment are shown in Figure 10 for the graphs webbase_1M, wiki-Talk and Web_Google from the dataset of [1]. Due to the irregular nature of memory accesses, pointer jumping is the predominantly costly operation in the Shiloach-Vishkin [25] algorithm on the GPU. Since our technique reduces the number of these operations, there is a significant impact on the overall runtime.

Finally, we also study the trade-off between the number of iterations of Phase I and the overall runtime. This study is motivated by similar reasons as explained in Section IV-B. Figure 11 shows the results of the above experiment for the graph webbase_1M from the dataset of [1]. It can be noticed that the overall time taken decreases over iterations of Phase I and plateaus off at about five iterations for the graph under consideration. The time for Phase I is shown on the right-side Y-axis of Figure 11.

VI. All Pair Shortest Paths

In graph theory, finding shortest paths in a weighted graph is a fundamental and well researched problem. The problem seeks to find the shortest path between any two nodes of the graph such that the sum of the weights of the constituent edges is minimized. The All-Pairs-Shortest-Paths (APSP) problem is a generalization where one seeks to find the shortest path between every pair of nodes in the graph. The most popular solution of the APSP problem is the Floyd-Warshall algorithm which has a $\mathcal{O}(V^3)$ running time and a $\mathcal{O}(V^2)$ space complexity. As the Floyd-Warshall algorithm is generally not well



s/t	1	2	3	4	5
1	0	1		2	
2	1	0		3	
3					
4	2	3		0	
5					

s/t	1	2	3	4	5
1	0	1	1	2	2
2	1	0	2	3	3
3	1	2	0	2	1
4	2	3	2	0	3
5	2	3	1	3	0

Fig. 12. An example run of our algorithm on the graph in part (a). The labels on the edges are the weights of each edge. Part (b) is the graph obtained after removing pendant nodes, (c) shows the result of Phase II, and (d) shows the result of Phase III.

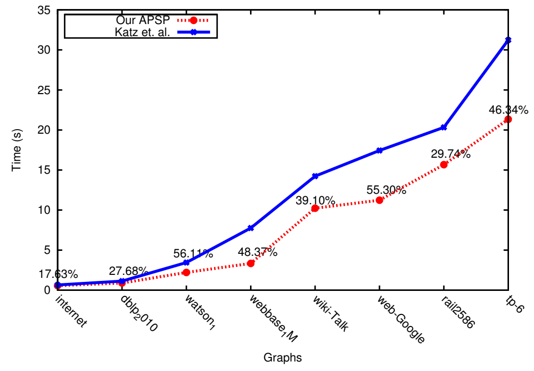


Fig. 13. Performance of APSP on the UF Sparse Matrix dataset. The numbers show the percentage improvement.

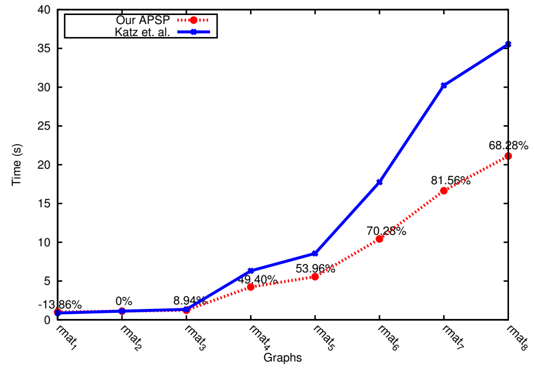


Fig. 14. Performance of APSP on R-MAT graphs from Table I. The numbers show the percentage improvement.

suited for sparse graphs, there are special algorithms designed for sparse graphs [15]. On GPUs, there are very few reported implementations. Notable among these are those of Harish et al. [13], Katz et al. [17], and [27]. In Harish et al. [13], the problem is solved by running a parallel Dijkstra's algorithm [9]. This is shown to be better for sparse graphs.

A. Implementation

Notice that for a pendant node v with w as its only neighbor, the shortest path from v to any other node u will always pass through w. Therefore, pendant nodes can be safely removed from the graph in the pruning step and the required shortest paths can be easily computed. For instance, for the graph in Figure 2, the shortest path from node 5 to node 4 has to necessarily go through the only neighbor of node 5, that is node 3. Further, if node 3 is removed in the second iteration of Phase I, then the shortest path from node 3 to any other node has to necessarily go through its only remaining neighbor, i.e., node 1. Hence, in the first phase we prune the pendant nodes. We remove all the pendent nodes of graph G and obtain G' over a few iterations along with the book keeping of removed nodes along with their iteration number.

In this implementation, we again use the compacted edge representation, and a separate weight array is maintained. The weight function W on the edges associates a random weight with each edge. As is done in [13], we run a single-source-shortest-paths (SSSP) algorithm from each node in graph $G(V, E, W)$. A parallel implementation of Dijkstra's algorithm [9] is used to solve SSSP. The basic step is to select a node and update it's neighbors depending upon the minimum cost. Shared memory is used to prefetch all the neighbors of a certain node while it is being processed. The entire execution happens only on the GPU for ease of comparison. (There are no hybrid implementations of APSP reported). We refer the reader to [13] for more details of the implementation. An example run of our algorithm is presented in Figure 12 for the graph from Figure 2.

B. Results

In this section, we present the results of our implementation. We compare our implementation results with those of [17].

In Figure 13, we show the results of our implementation on a sample of eight real world graphs from the University of Florida dataset [1]. As shown in Figure 13, our results are consistently better. This improvement can be attributed to the fact that APSP is a highly computationally intensive workload, roughly of cubic order. Hence, any decrease in the size of the graph reflects prominently in a corresponding decrease in the overall runtime. A similar experiment on the R-MAT graphs from Table I is shown in Figure 14. It can be noticed from Figure 14 that there is a considerable improvement of 44% on average.

Similar to our experiments on the BFS implementation, we also study the percentage improvement of our implementation as a function of the percentage of nodes that Phase I can remove. The results of this experiment are shown in Figure 15 on the graphs webbase_1M, wiki-Talk and Web_Google from the dataset of [1]. While there are not many irregular operations in the APSP implementation on a GPU, the workload is still computationally heavy. This can be observed from the total runtime from Figures 13 and 14, where the run time is of the order of one minute. Hence, pruning even a small fraction of the nodes results can potentially decrease the runtime by a large percentage.

Finally, we also study the trade-off between the number of

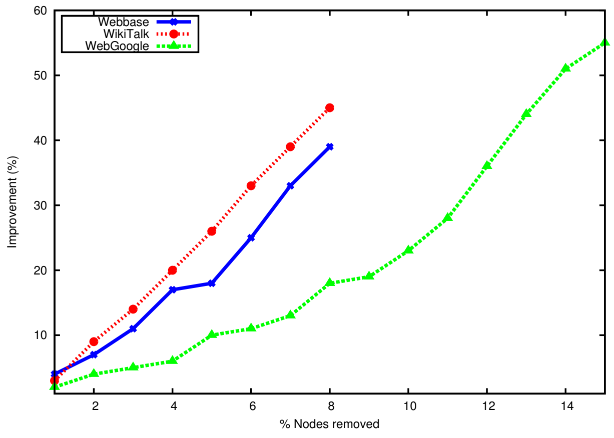


Fig. 15. Performance improvement of APSP with the removal of pendant nodes.

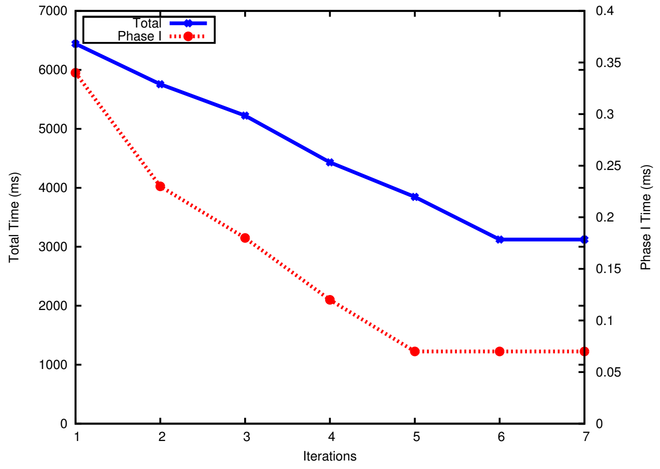


Fig. 16. Tradeoff between Phase I and the total time.

iterations of Phase I and the overall runtime. This study is motivated by similar reasons as explained in Section IV-B. Figure 16 shows the results of the above experiment on the graph webbase_1M from the dataset of [1]. It can be noticed that the overall time taken decreases over iterations of Phase I and plateaus off at about six iterations for the graph under consideration. The time for Phase I is shown on the right-side Y-axis of Figure 16.

VII. Conclusions

In this paper, we have proposed graph pruning as a technique to speed-up large graph algorithms on modern parallel architectures. We applied the technique to three important problems in graphs. Our results indicate that the technique is quite useful, especially for large sparse graphs.

In all the applications we studied in this paper, we needed to prune the pendant nodes. In future, we wish to study other problems that will lead to the discovery of other pruning strategies.

References

[1] The University of Florida Sparse Matrix Collection. http://www.cise.ufl.edu/research/sparse/matrices/.

[2] Agarwal, V., Pasetto, F. P. D., and Bader, D. A. Scalable graph exploration on multicore processors. In *Proc. of ACM SC* (10), p. 111.

[3] Anderson, R. J., and Miller, G. L. A Simple Randomized Parallel Algorithm for List-Ranking. *IPL 33*, 5 (1990), 269–273.

[4] Bader, D. A., and Cong, G. A fast, parallel spanning tree algorithm for symmetric multiprocessors (SMPs). *Journal of Parallel and Distributed Computing 65*, 9 (2005), 994 – 1006.

[5] Bader, D. A., and Madduri, K. GTgraph: A suite of synthetic graph generators.

[6] Banerjee, D. S., and Kothapalli, K. Hybrid Algorithms for List Ranking and Graph Connected Components. In *Proc. HiPC* (2011).

[7] Chhugani, J., Satish, N., Kim, C., Sewall, J., and Dubey, P. Fast and Efficient Graph Traversal Algorithm for CPUs: Maximizing Single-Node Efficiency. In *IPDPS* (2012), pp. 378–389.

[8] Cole, R. Parallel merge sort. *SIAM J. Comput. 17*, 4 (Aug. 1988), 770–785.

[9] Cormen, T., Leiserson, C., Rivest, R., and Stein, C. Introduction to algorithms, 2001.

[10] DAngelo, G., DEmidio, M., Frigioni, D., and Maurizio, V. A speed-up technique for distributed shortest paths computation. In *Computational Science and Its Applications-ICCSA 2011*. 2011, pp. 578–593.

[11] Gharaibeh, A., Costa, L. B., Santos-Neto, E., and Ripeanu, M. On Graphs, GPUs, and Blind Dating: A Workload to Processor Matchmaking Quest. In *in Proc. of IEEE IPDPS* (2013).

[12] Greiner, J. A comparison of parallel algorithms for connected components. In *Proc. SPAA* (1994), pp. 16–25.

[13] Harish, P., and Narayanan, P. J. Accelerating Large Graph Algorithms on the GPU using CUDA. In *Proc. of HiPC* (2007).

[14] Hong, S., Oguntebi, T., and Olukotun, K. Efficient parallel graph exploration for multi-core CPU and GPU. In *PACT* (2011).

[15] Johnson, D. B. Efficient algorithms for shortest paths in sparse networks. *J. ACM 24*, 1 (Jan. 1977), 1–13.

[16] Karypis, G., and Kumar, V. Parallel multilevel k-way partitioning scheme for irregular graphs. In *Proc. ACM SC* (1996).

[17] Katz, G. J., and Kider, Jr, J. T. All-pairs shortest-paths for large graphs on the GPU. In *Proceedings of the 23rd ACM Symp. Grap. Hard.* (2008), pp. 47–55.

[18] Matsumoto, K., Nakasato, N., and Sedukhin, S. Blocked All-Pairs Shortest Paths Algorithm for Hybrid CPU-GPU System. In *Proc. HPCC* (2011), pp. 145–152.

[19] Merrill, D., Garland, M., and Grimshaw, A. "scalable gpu graph traversal". *SIGPLAN Not. 47*, 8 (Feb. 2012), 117–128.

[20] Micikevicius, P. General Parallel Computation on Commodity Graphics Hardware: Case Study with the All-Pairs Shortest Paths Problem. In *PDPTA'04* (2004), pp. 1359–1365.

[21] Munguia, L.-M., Bader, D. A., and Ayguadé, E. Task-based parallel breadth-first search in heterogeneous environments. In *HiPC* (2012), pp. 1–10.

[22] Pattabiraman, B., Patwary, M. M. A., Gebremedhin, A. H., keng Liao, W., and Choudhary, A. N. Fast algorithms for the maximum clique problem on massive sparse graphs. *CoRR abs/1209.5818* (2012).

[23] Scarpazza, D. P., Villa, O., and Petrini, F. Efficient Breadth-First Search on the Cell/BE Processor. *IEEE Trans. Parallel Distrib. Syst. 19*, 10 (2008), 1381–1395.

[24] Schieber, B., and Vishkin, U. On finding lowest common ancestors: simplification and parallelization. *SIAM J. Comput. 17*, 6 (Dec. 1988), 1253–1262.

[25] Shiloach, Y., and Vishkin, U. An O(log n) Parallel Connectivity Algorithm. *J. Algorithms* (1982), 57–67.

[26] Soman, J., Kothapalli, K., and Narayanan, P. Some GPU Algorithms for Graph Connected Components and Spanning Tree. In *Parallel Processing Letters* (2010), vol. 20, pp. 325–339.

[27] Venkataraman, G., Sahni, S., and Mukhopadhyaya, S. A blocked all-pairs shortest-paths algorithm. *J. Exp. Algorithmics 8* (Dec. 2003).

[28] Villa, O., Scarpazza, D., Petrini, F., and Peinador, J. Challenges in mapping graph exploration algorithms on advanced multi-core processors. In *IPDPS 2007* (2007), pp. 1–10.

[29] Wei, Z., and JaJa, J. Optimization of Linked List Prefix Computations on Multithreaded GPUs Using CUDA. In *IPDPS* (April 2010).

[30] Xia, Y., and Prasanna, V. K. Topologically Adaptive Parallel Breadth First Search on Multicore-Processors. In *in Proc. PDCS* (2009).