Accepted Manuscript

Work efficient parallel algorithms for large graph exploration on emerging heterogeneous architectures

Dip Sankar Banerjee, Ashutosh Kumar, Meher Chaitanya, Shashank Sharma,
Kishore Kothapalli

PII: S0743-7315(14)00224-X
DOI: http://dx.doi.org/10.1016/j.jpdc.2014.11.006
Reference: YJPDC 3377

To appear in: *J. Parallel Distrib. Comput.*

Received date: 25 March 2014
Revised date: 17 November 2014
Accepted date: 25 November 2014

Please cite this article as: D.S. Banerjee, A. Kumar, M. Chaitanya, S. Sharma, K. Kothapalli, Work efficient parallel algorithms for large graph exploration on emerging heterogeneous architectures, *J. Parallel Distrib. Comput.* (2014), http://dx.doi.org/10.1016/j.jpdc.2014.11.006

Work Efficient Parallel Algorithms for Large Graph Exploration on Emerging Heterogeneous Architectures[1]

Dip Sankar Banerjee[a,*], Ashutosh Kumar[a], Meher Chaitanya[a], Shashank Sharma[a], Kishore Kothapalli[a]

[a] *International Institute of Information Technology, Hyderabad, Gachibowli, Hyderabad, India 500 032.*

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 solution strategies 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 on the pruned graph, the solution is extended to the entire graph. Towards, this end we investigate pruning based on two strategies that justifies their use in current real world graphs.

We apply the above technique on three fundamental graph algorithms: breadth first search (BFS), Connected Components (CC), and All Pairs Shortest Paths (APSP). For experimentations, we use three different sources for real world graphs. 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 reliant on these algorithms.

Keywords: graph algorithms, irregular computations, heterogeneous computing, input pruning

*Supported by Tata Consultancy Services Ltd. through the TCS Research Scholarship Program.

Email addresses: dipsankar.banerjee@reasearch.iit.ac.in (Dip Sankar Banerjee), ashutosh.kumar@students.iit.ac.in (Ashutosh Kumar), meher.c@reasearch.iit.ac.in (Meher Chaitanya), shashank.sharma@students.iiit.ac.in (Shashank Sharma), kkishore@iiit.ac.in (Kishore Kothapalli)

1. A part of this work has appeared previously in Proceedings of The 20th IEEE International Conference on High Performance Computing (HiPC), 2013

1. 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), computing components, and finding 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 stresses the I/O systems 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 [38, 22, 41]. Recent results have been able to make efficient use of modern parallel architectures such as the Cell BE [41], GPUs [38, 23, 22], Intel multicore architectures [14, 49, 3] and the like. Algorithms running of GPUs have shown standout performance amongst these because of its massive parallelism.

Another recent development in parallel computing is the design and engineering of heterogeneous algorithms that are aimed at heterogeneous computing platforms. Heterogeneous computing platforms consist of tightly coupled heterogeneous devices including CPUs and accelerator(s). One such example is a collection of a CPU coupled with a graphics accelerator (GPU). Heterogeneous algorithms for CPU+GPU based computational platforms have been designed for also graph breadth-first exploration [23, 38, 20]. 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 and memory layout 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 vertex degrees. In Figure 1, we show some of the real-world graphs taken from [2]. As can be seen from Figure 1, these graphs have several vertices of very low degree, often as low as 1. For instance, in the case of the graph web-Google, 14% of the vertices have degree 1. Table 1 lists other properties of a few real world graphs from [2].

Current parallel algorithms and their implementations [38, 20, 41, 23, 47] 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 vertices that have to be explored next. But, a vertex v of degree 1 that is in the queue will not lead to the discovery of any yet undiscovered vertices. 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 vertices that can be discovered through vertex 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 vertex, may be quite disposable.

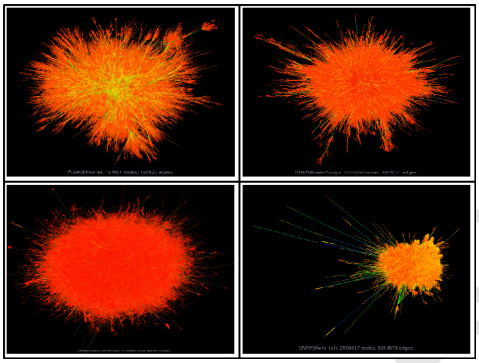

Figure 1: A sample of four real world graphs from [2]. On the top-left corner is the graph internet, top-right is the graph web-google, bottom left is the graph webbase_1M, and the bottom-right is the graph wiki-Talk.

In light of the above paragraph, we posit 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 *input pruning* as a technique in this direction. Input 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 input 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 degree one (or *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%. In addition, we also perform pruning based on articulation points, where we identify biconnected component of the graph and the suitable articulation points at which the smaller components can be pruned. We use this strategy to then perform APSP on each of the smaller components. The results are then merged to provide the final shortest path of the actual graph. This approach gives a 1.57x speedup over the state of the art results.

1.1. Related Work

Algorithm or implementation decisions based on the nature of the graph is an emerging area of research. In [19],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 close to n and 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 [40], Pattabiraman et al. show novel pruning techniques that solves the maximum clique problem on large sparse graphs. Their main idea is to prune the vertices that strictly have fewer neighbors than the size of the maximum clique already computed. These are the vertices 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.

In another work by Cong et al.[16], the authors explore an experimental technique for the computation of biconnected components on symmetric multiprocessors. Towards this, the authors propose a modification of the well known Tarjan's algorithm [45] for computing biconnected components. The authors show how to find and hence remove *non-essential* edges that do not affect the biconnected components of the graph. By removing such non-essential edges, the time taken to find the biconnected components can reduce vastly as is shown by Cong and Bader [16]. In a more recent work [24], the authors show a pruning based strategy for identifying the strongly connected components (SCCs) of a directed graph. In this paper, the authors propose a trimming technique that works on small world graphs based on their properties. It is observed such graphs tend to possess one gaint SCC, with a size that is $\Theta(n)$, and a lot of SCCs that have very small sizes, including SCCs with size 1, and size 2. The authors of [24] essentially identify the small-sized SCCs quickly and "trim" the input graph accordingly. The actual algorithms is then run on the remaining graph thereby increasing the overall performance.

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 [4], the optimal merging algorithm [15], the optimal range minima algorithm [42], and so on. In all of these cases, the size of the input is reduced by a non-constant factor 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.

Many recent works in parallel computing have focused on graph algorithms. Few among them include [14, 23, 48, 38, 11]. The work of Scarppaza et al. [41] demonstrates the use of an all-to-all exchange of visited nodes information in a BFS execution across the eight SPUs of a Cell BE. One of the first results of BFS using GPUs is the work of Harish et al. [22]. Subsequent improvements to [22] centered around the use of heterogeneous computing. In [23], Hong 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. [38]. Another work on parallel BFS was presented by Beamer et al. [11], where the authors show an improvement of upto 4.8x on real world graphs based on edge contraction. Some of the prominent works on multicore CPUs include [14] where the primary goal is to map the data structures to the cache hierarchy so as to improve the cache hit rates. A recent work [20] partitions the graph so that low degree vertices are processed on the GPU and the high degree vertices 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 [43] and its variants by Greiner [21]. On GPUs, a variant of Shiloach and Vishkin [43] is used by Soman et al. [44]. A heterogeneous execution of this algorithm on a CPU+GPU platform with an improvement of 35% on average is shown in [9].

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 [36]. In this work, the author proposed a parallel implementation of the popular Floyd-Warshall algorithm on an early generation FX5900 GPU which showed speedups upto 3x over sequential CPU code. In a more recent work [46], 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 [28], the authors show a new APSP implementation on the GPU where the authors utilizes the available shared memory efficiently using a G80 GPU. They employ a transitive closure based technique whereby adapt the Floyd-Warshall algorithm across single and multiple GPUs. This is the current known best result of the APSP problem on GPUs. Matsumoto et al. proposed an hybrid APSP work based on the work in [46] in [35]. 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.

More recent works in this direction are summarized below. Djidjev et al. [18] use graph decomposition via Parmetis [27], compute shortest paths within the partitions and extend the same to paths across partitions. The success of their approach depends on two factors: the ability to find a good partition, and the ability to find paths across partitions quickly. In their work, they work mostly with planar graphs to ensure a good partition.

1.2. 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 35% 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.

In Figure 2, we show the overall improvements achieved from our implementations on graphs from Table 1. Further results on some more graphs from the UFL Sparse Matrix Collection [2] and on random matrices generated using the R-MAT synthetic graph generator [13] are shown in our previous work [8].

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

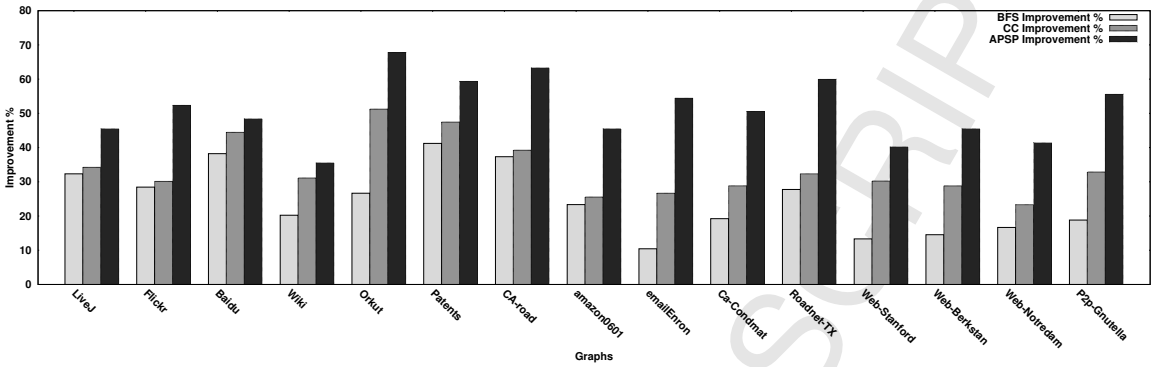

Figure 2: Overall improvements due to pruning pendant nodes on the graphs from Table 1 over the previous best known results.

sets including graphs from the University of Florida collection [2], and graphs generated using the Recursive Matrix Model (R-MAT).

- On the connected components problem, we get an average 20% improvement over the best known result on an identical platform [9]. 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 [28] on a similar platform.
- Using the second pruning strategy based on biconnected components we achieve a speedup of 1.57x on an average over similar graphs.

2. 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

SNAP Graphs					
Graph	**Description**	**Nodes**	**Edges**	**Pendant Vertices**	**r**
amazon0601	Amazon product co-purchasing network [31]	403,394	3,200,490	38,121 **(9.45%)**	3
email-Enron	Email communication network from Enron [29]	36,692	367,662	2,069 **(5.64%)**	2
ca-Condmat	Collaboration n/w of Condensed Matter [25]	23,133	186,936	3,338 **(14.43%)**	2
Roadnet-TX	Road network of Texas [33]	1,393,383	3,843,320	170,271 **(12.22%)**	4
Web-Stanford	Web graph of Stanford.edu [33]	281,903	2,312,497	21,819 **(7.74%)**	3
Web-Berkstan	Web graph of Berkeley and Stanford [33]	685,230	7,600,595	60,506 **(8.83%)**	3
Web-Notredam	Web graph of Notre Dam [33]	325,729	1,497,134	33,322 **(10.23%)**	2
p2p-Gnutella	Gnutella peer to peer network [30]	62,586	147,892	9,738 **(15.56%)**	2
LiveJ	Links in Live Journal[6]	4,848,571	68,993,773	403,401 **(8.3%)**	4
Flickr	Connection among Flickr users [37]	2,302,925	33,140,018	488,450 **(21.2%)**	3
Baidu	Links in Baidu Chinese online encyclopedia [39]	2,141,300	17,794,839	266,592 **(12.4%)**	5
Wiki	Links in english wikipedia [5]	15,172,740	131,166,252	1,195,612 **(7.8%)**	6
Orkut	Connection of Orkut users [50]	3,072,627	11,718,583	464,274 **(15.1%)**	4
Patents	Citations among US patents [32]	3,774,768	16,518,948	691,160 **(18.3%)**	2
Roadnet-CA	Road network of California [34]	1,965,206	5,533,214	228,357 **(11.6%)**	3

Table 1: The SNAP [1] 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 vertices.

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 from the Intel family with 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 a size of 12 MB. The L1 cache size is 64 KB per core and L2 is 256 KB. Other features of the Core i7 980 include a 32 KB instruction and a 32 KB data L1 cache per core and the L3 cache is shared by all 6 cores.

3. Our Approach

In this section, we present a three phase technique, outlined 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 output of A on G'. In a post-processing third phase, we extend the solution O' on G' 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 vertices 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 $Prune(Graph\ (\mathcal{V}, \mathcal{E}))$

```
for i = 1 to r iterations do
    for each vertex v ∈ G do
        if v has property P then
            Remove v, and all edges incident on v.
            Store (v, i) for future re-insertion step.
        endif
    endfor
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 [9].

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 [19].

- 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 [4] and its recent heterogeneous implementation [48, 9].

- 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 [27], introduce the coarsening-refinement approach. During the coarsening step, a matching of the current graph is computed and prunes matched vertices.

- Connected Components: Graph components often provide several core properties of real world graphs. Biconnected components in undirected graphs or strongly connected components in directed graphs can be investigated for pruning and subsequent processing of the compressed graph. In a latest work [24], the authors have shown precisely this sort of an investigation where several trivial components has been trimmed from the actual graph for the computation of strongly connected components on the remainder graph.

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.

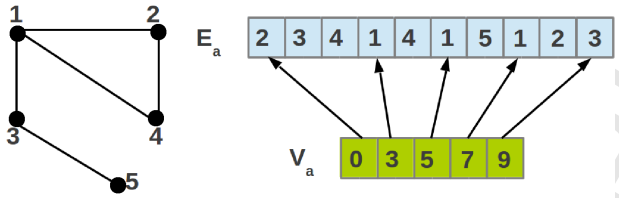

Figure 3: The CSR format for representation.

4. 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 vertex $\mathcal{S}$, compute the minimum number of edges that are needed to reach every vertex of $\mathcal{G}$ from $\mathcal{S}$. The optimal sequential solution to this problem runs in $\mathcal{O}(\mathcal{V} + \mathcal{E})$ time [17].

The well known sequential algorithm maintains a queue where the newly discovered vertices are are inserted at the rear. Current vertices are deleted from the front of the queue and this process continues until the queue is depleted. All the newly visited vertices 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). An example is shown in Figure 3. 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 vertex $i + 1$ immediately follows vertex i, for all the vertices 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 vertex number of of the graph. The key advantage 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 vertex. This helps in reducing the memory access irregularity and hence boosts the overall performance of the BFS implementation.

4.1. Implementation

The basic approach of our algorithm is to first perform a pruning step where pendant vertices are removed iteratively. This is followed by an efficient parallel execution of BFS on the CPU+GPU hybrid platform from Section 2. Finally, in a post-processing step, the level number for vertices that were removed initially will be is computed. Our algorithm is described in Algorithm 3.

Phase I:. The first phase of removing the pendant vertices 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.

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.

- To reduce the time spent in Phase I, we use the CSR representation and identify pendant vertices as follows. Consider vertices u and v numbered consecutively. Then, u is a pendant vertex if $V_a[u]$ and $V_a[v]$ differ by 1. If vertex u is numbered n, then the above rule has to be modified to say that $V_a[n] = |E_a|$. See Figure 3 for an illustration where vertex 5 is a pendant vertex, 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 vertex in iteration $i + 1$. Further, in iteration $i + 1$, we need to check only such vertices w. Therefore, we mark such w in iteration i, and do not check other vertices in iteration $i+1$. This helps in reducing the time spent in Phase I across each iteration. For illustration, see Figure 3. If vertex 5 is removed in the first iteration, then vertex 3 is marked as a potential pendant vertex in that iteration. Since the remaining degree of vertex 3 is now 1, also vertex 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 vertex $\mathcal{S}$
Output: Level numbers of all the vertices.
1: Set a threshold for separating the vertex set between CPU and GPU.
2: Create G_c graph for CPU and 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 [17].
7: Check NFR array of GPU and CPU for termination
8: Set FR := NFR
9: **endwhile**
10: Consolidate LEVEL values

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. [38]. 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 vertices, the current frontier vertices, and the next frontier vertices,

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

Input: A graph $\mathcal{G}(\mathcal{V}, \mathcal{E})$ and starting vertex $\mathcal{S}$
Output: Level numbers of all the vertices.

```
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 vertices from range
6: for i = start to stop
7:     VISITED[i]=1
8:     for i ∈ 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
```

respectively. The arrays VISITED is shared between the two devices so that the status of a vertex 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 vertex 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 vertices 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 vertices 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 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 arrays.

The consolidation of the LEVEL arrays created in both CPU and GPU is done using the information that is kept during the partitioning the edge set between the two processors. It is to be kept in mind that we employ static partitioning for the purpose of

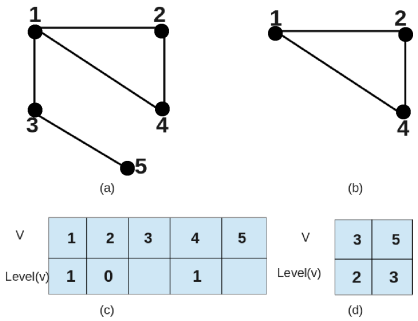

Figure 4: An example run of our algorithm on the graph in part (a). Part (b) is the graph obtained after removing pendant nodes, (c) shows the result of Phase II, and (d) shows the result of Phase III.

our execution. Static partitioning of work is a popular technique which is broadly used in several libraries like ScaLapack [12]. The technique is primarily aimed at creating independent sub tasks so that they can be efficiently processed with minimum overheads associated with communication and synchronization. For our purposes, when we partition the edge set between the CPU and the GPU, we also keep track of the set of *cross-edges* between the two processors. We define cross-edges as the set of edges which has one vertex on one processor and the other vertex on the other. This way, when the LEVEL arrays are consolidating the values, the cross-edges provides the point of connection. We can very easily identify the vertex from the LEVEL array obtained from the GPU, which has connection to the CPU LEVEL array. This way, starting with that unique vertex (or vertices), we can simply add an extra level to all the values currently existing on the CPU side LEVEL array to get the consolidated levels.

Phase III. In Phase III of the algorithm, we re-insert the vertices that were removed in Phase I as follows. Let v be a vertex 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 $i-1$. It can be seen easily that our approach does not affect the correctness of a breadth-first traversal. An example run of our algorithm is presented in Figure 4 for the graph from Figure 3.

4.2. Results

In this section, we present the results of our implementation. We compare the results with those of [38]. The results of [38] are the currently reported best results for graph breadth first search on identical platforms.

The earlier results on the R-MAT datasets and the datasets obtained from the University of Florida Sparse Matrix Collection has been reported in [8]. 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 5 for three sample

graphs: Baidu, Orkut and Patents from the SNAP dataset. Figure 5 indicates that significant performance gains can be achieved even as a small percentage of vertices are pruned from the input graph. The improvement can be attributed to the lesser number of operations required in our implementation as pruned vertices do not enter/exit arrays FR, VISITED, and NFR. As we can notice, there is a significant improvement in performance which is achieved as a result of pendants pruning from the input graphs. Another thing that can be noticed from the experimentations on the SNAP graphs is that due to the smaller size of graphs actually used to perform BFS, there is a higher amount of scalability that can be achieved. This is due to the fact that along with a reduction in work complexity, there is a significant reduction in the space complexity too that is obtained as a result of pruning. Pruning helps in better use of the FR, VISITED and NFR data structures. Higher scalability is also achieved in a similar fashion while experimenting on the other work loads too which are discussed in the future sections.

Figure 6 shows the results of the above experiment on the Baidu graph from the [1] dataset. This figure shows the trade-off between the number of iterations of Phase I and the overall runtime. As we have also noticed earlier, that Phase I time decreases over successive iterations. This can be attributed to the vertex removal technique which we have proposed in [8]. Hence, similar results are observed in case of the SNAP datasets as well. In Figure 6, we can observe the Phase I time on the right Y-axis and can note how the overall time plateaus out after five iterations. Hence, it is worthwhile to stop the removal of the pendant vertices after a certain small threshold.

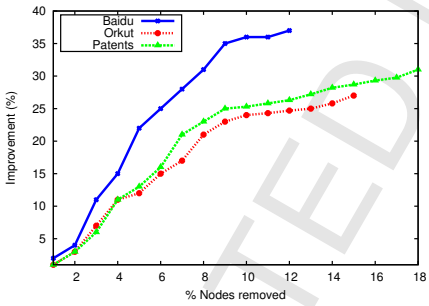

Figure 5: Percentage improvement for SNAP graphs Baidu, Orkut and Patents.

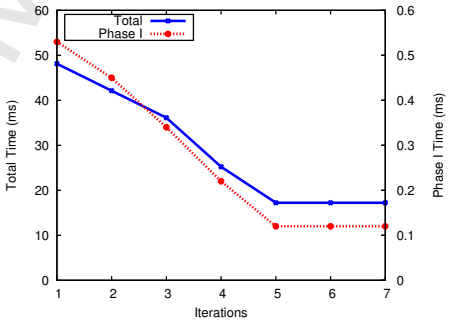

Figure 6: Trade-off in BFS between Phase I and the overall runtime of BFS for Baidu graph from SNAP.

5. 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 vertices 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) [17] 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 [43], and the algorithm of Greiner et al. [21]. 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 [7], Cray and CM2 [21], GPUs [44], and also on CPU+GPU systems [9].

In this section, we apply techniques from Section 3 and show that highly efficient hybrid algorithms can be designed for this workload on the CPU+GPU hybrid platform described in Section 2. Our solution can be broadly outlined in the following steps and follows the algorithm used in [9].

5.1. Implementation

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 vertex 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 [43], 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.

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.

In the next steps, we partition the graph according to a predetermined 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 the 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 [43] for the GPU computation and DFS on the individual CPU cores.

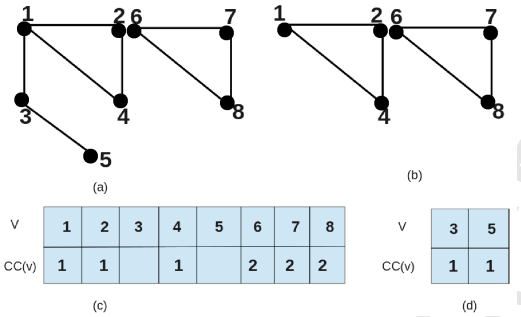

Figure 7: An example run of our algorithm on the graph in part (a). Part (b) is the graph obtained after removing pendant nodes, (c) shows the result of Phase II, and (d) shows the result of Phase III.

GPU and CPU Optimizations : The Shiloach–Vishkin [43] 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 SV 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 [44].

For finding the connected components of the graph G_i for $1 \leq i \leq c$ on the ith core of the CPU, we use the standard DFS algorithm [17]. This is motivated by the fact that since 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 vertex v removed in the ith iteration, let w be the only neighbor of v. Then the vertex v is said to belong to the component that w belongs to. In the above, we process vertices in the opposite order of their removal in Step 1. An example run of our algorithm is presented in Figure 7 for the graph from Figure 3.

5.2. Results

In this section, we present the results of our implementation. We compare the results with those of [9]. The results of [9] are currently the best reported results for finding the connected components of a given graph on a CPU+GPU platform.

As in the case of BFS, results on the graphs obtained from R-MAT and UFL repository has been already discussed in [8]. Similar to the BFS implementation, we also study the 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 8 for the graphs Baidu, Orkut and Patents from the SNAP dataset. As mentioned earlier, we adopt the Shiloach-Vishkin [43] algorithm for the GPU implementation. The several steps of the algorithm has been mostly discussed in a previous work [9]. Due to the pruning mechanism, we are working on a graph which has a significantly lesser number of edges. This gives a big advantage from the point of thread divergence which our implementation [9] takes advantage of. Additionally, memory irregularities involved in the step of hooking of smaller subtrees is significantly reduced.

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 4.2. Figure 9 shows the results of the above experiment for the graph Baidu 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 9.

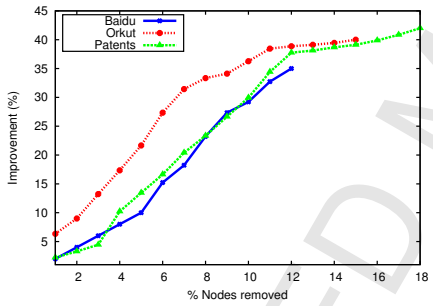

Figure 8: Percentage improvement of Connected Components for SNAP graphs Baidu, Orkut and Patents.

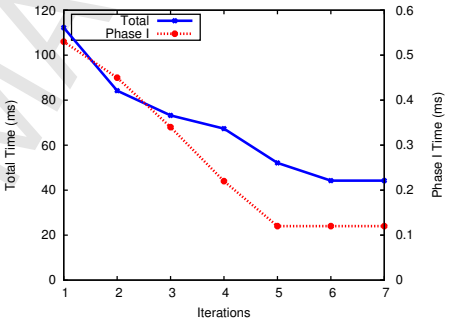

Figure 9: Trade-off in CC computation between Phase I and the total time for Baidu graph from SNAP.

6. All Pairs 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 vertices 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 vertices 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 suited for sparse graphs, there are special algorithms designed for

sparse graphs [26]. On GPUs, there are very few reported implementations. Notable among these are those of Harish et al. [22], Katz et al. [28], and [46]. In Harish et al. [22], the problem is solved by running a parallel Dijkstra's algorithm [17]. This is shown to be better for sparse graphs.

6.1. Implementation

In this case too, we still prune the pendant nodes in the graph iteratively. Notice that for a pendant vertex v with w as its only neighbor, the shortest path from v to any other vertex u will always pass through w. Therefore, pendant vertices 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 3, the shortest path from vertex 5 to vertex 4 has to necessarily go through the only neighbor of vertex 5, that is vertex 3. Further, if vertex 3 is removed in the second iteration of Phase I, then the shortest path from vertex 3 to any other vertex has to necessarily go through its only remaining neighbor, i.e., vertex 1. Hence, in the first phase we prune the pendant vertices. We remove all the pendant vertices 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 [22], we run a single-source-shortest-paths (SSSP) algorithm from each vertex in graph $G(V, E, W)$. A parallel implementation of Dijkstra's algorithm [17] is used to solve SSSP. The basic step is to select a vertex 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). The computation is spread across two GPU kernels so as to avoid read/write inconsistencies. We refer the reader to [22] for more details of the implementation. An example run of our algorithm is presented in Figure 10 for the graph from Figure 3.

6.2. Results

In this section, we present the results of our implementation. We compare our implementation results with those of [28]. The results of [28] are the currently reported best results for finding the connected components of a given graph on identical platforms.

Our previous experimentations on the graphs obtained from R-MAT and the UFL collection has been reported in [8]. In lines of the experiments carried out in the previous implementations of BFS and CC, we also study the percentage improvement of our APSP implementation as a function of the percentage of nodes that Phase I can remove. The results of this experiment are shown in Figure 11 on the graphs Baidu, Orkut and Patents from the SNAP dataset. 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 improvements from Figure 2. We can see that we get a speedup of 44% on an average. Hence, pruning even a small fraction of the nodes results can potentially decrease the runtime by a large percentage.

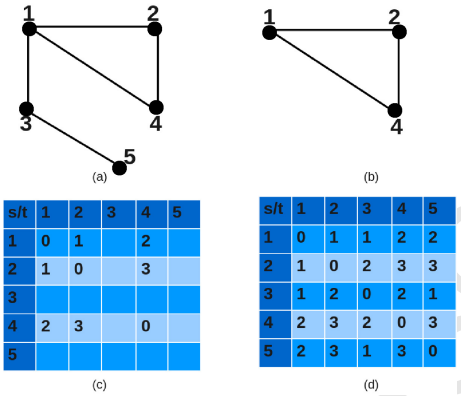

Figure 10: An example run of our algorithm on the graph in part (a). Part (b) is the graph obtained after removing pendant nodes, (c) shows the result of Phase II, and (d) shows the result of Phase III.

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 4.2. Figure 12 shows the results of the above experiment on the graph Baidu from SNAP the dataset. 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 12.

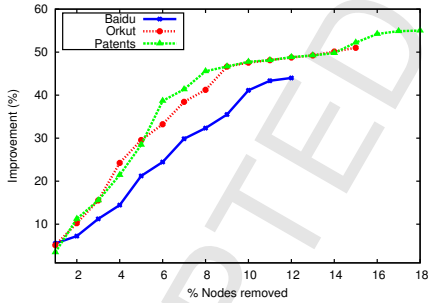

Figure 11: Percentage improvement of APSP for SNAP graphs Baidu, Orkut and Patents.

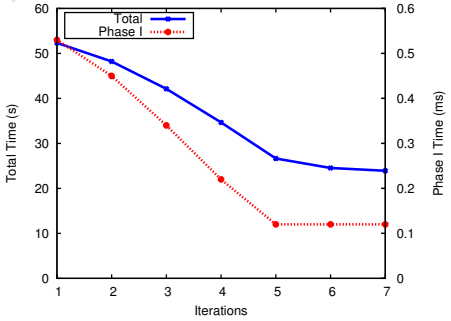

Figure 12: Trade-off in APSP between Phase I and the total time for Baidu graph from SNAP.

7. Pruning Based on Bridges and Articulation Points

In this section we introduce a pruning technique that prunes bridges of the graph. Recall that a bridge in a graph G is an edge whose removal disconnects G. Similarly, an articulation point in a graph G is a vertex whose removal disconnects the graph.

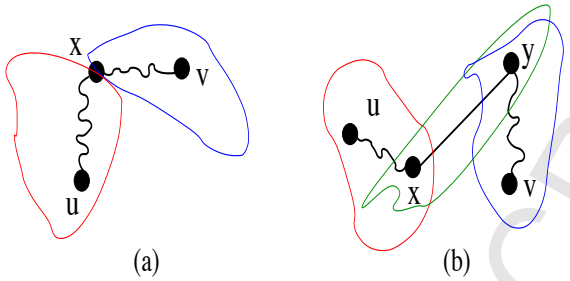

Figure 13: Figure (a) shows how to extend shortest paths between nodes u and v across biconnected components connected by an articulation point x. Figure (b) shows such an extension when an multiple biconnected components separate the two nodes u and v. Each biconnected component is shown with different color.

Bridges and articulation points can be used to partition the edges of the graph G into maxmial 2-connected subgraphs that are also called as the biconnected components (BCCs) of G.

For the case of All-Pairs-Shortest-Paths, such a decomposition into BCCs can be extremely helpful. As shown in Figure 13(a), for vertices u and v in distinct biconnected components separated by an articulation point x, the shortest path from u to v can be constructed as the shortest path from u to x, and the shortest path from x to v. Similar rules can be designed for nodes in two distinct biconnected components that are separated by multiple biconnected components as shown in a preliminary fashion in Figure 13(b).

Further, such a decomposition based on BCCs is also relevant since real-world graphs are sparse and tend to have many biconnected components. Figure 14 shows evidence for the same for the graph "internet" from the dataset of [10]. Notice from Figure 14, that there is one BCC with about 100 K edges in this graph. The remaining 3K BCCs are of varying sizes. The graph has 182 K edges, which means that the largest BCC has 56% of the entire edges of the graph. Further, there are several BCCs with sizes varying between 2 to 250 as indicated in the pie-chart in Figure 14. Since one is now solving the problem in each BCC followed by post-processing, it is seen that there will be performance gains also.

7.1. Our Approach for APSP

Based on the observations from Section 2, we propose Algorithm 7 for APSP.

7.1.1. Phase I: Prune Bridges and Articulation Points

In Phase I, we identify the bridges and the articulation points of the input graph G and use them to partition the edges of G into biconnected components, each of which is maximally 2-connected. To identify the bridges and the articulation points, we use the standard sequential algorithm of Tarjan [17]. We note that as the actual computation takes lot more time than the identification of bridges and articulation points, we do not

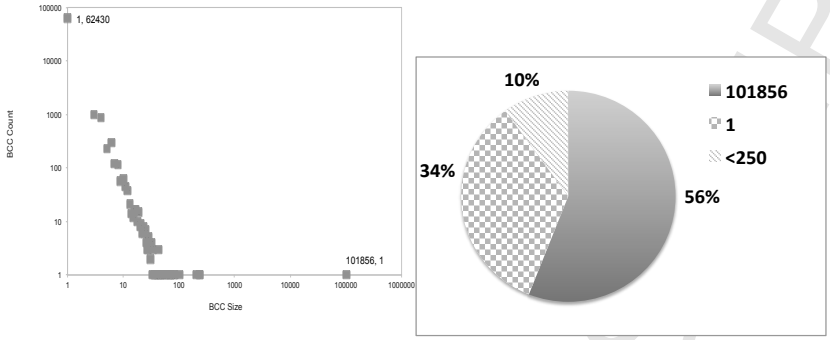

Figure 14: Histogram of BCC sizes for one sample graph. The figure on the right shows the same data in a pie chart. The sizes of the BCCs are clustered into 3 groups: 101856, 1, and < 250.

Algorithm 7 APSP(G)

```
PruneBridges(G) /* Phase I*/
for i = 1 to k in parallel /* Phase II*/ do
    APSP(G_i)
end for
for each pair of distinct vertices u, v in parallel /* Phase III*/ do
    if u and v are in different BCCs of G_1 then
        Compute the shortest uv-path
    end if
end for
```

parallelize the computation in this phase. As we will see in Figure 16, the time spent in this phase is often under 3% of the overall time on average.

We also note that unlike earlier sections, we do not remove the pendant nodes before identifying the bridges. It can be seen that edges with one end point as a pendant node also will be identified as bridges. So, we need not explicitly remove the pendant nodes in this phase.

7.1.2. Phase II: Paths Between Nodes within Same BCCs

In this phase, we also use CPU and GPU parallelism. On the CPU and the GPU, we run multiple independent iteration of Dijkstra's algorithm on each of the BCCs obtained in Phase I. We let the GPU start processing the BCC with the largest size. The CPU starts with the BCC of the smallest size. We implemented Dijkstra's algorithm on the CPU using the heap data structure. On the GPU, we use the program developed by Harish et al. [22]. The computation in this phase is similar to the computation in Section 6.1.

7.1.3. Phase III: Paths Between Nodes Across Two BCCs

The computation in Phase III involves the following steps. Let us define the graph H as the block graph of G. In H, there is a node for each BCC of G. Two nodes s

Graph name	Articulation Points	#BCCs	Largest BCC (in % of edges)
amazon0601	12,542	13331	71%
email-Enron	1,391	12, 093	89%
ca-CondMat	2,906	3, 343	92%
Roadnet-TX	259,925	293, 694	82%
Web-Stanford	16,337	29, 470	72%
Web-Berkstan	25,468	63,968	77%
Web-Notredame	21,780	167, 543	55%
p2p-Gnutella31	12,254	9, 326	82%
LiveJ	7,237	125,231	65%
Flickr	46,059	456,144	72%
Baidu	20,694	84,528	49%
Wiki	5,847	1,769,165	74%
Orkut	112,289	332,184	51%
Patents	250,073	678,954	81%
Roadnet-CA	327,864	381, 366	56%

Table 2: List of sparse graphs that we use in our experiments.

and t in H are neighbors if and only if the BCCs of G corresponding to s and t share an articulation point in G. Notice that H as defined above is not necessarily a tree. (Consider a star graph of 3 edges. Each edge is in a distinct BCC. The block graph would be a cycle of 3 nodes.)

For each pair of distinct vertices s and t in H, we find the sequence of articulation points that the BCCs of G corresponding to s and t share. We can visualize this also as a (shortest) paths problem on H. In the second step, we have to compute the shortest path between every pair of nodes in G that lie in separate BCCs of G as follows.

Let C_i and C_j be two distinct biconnected components of G. We let u, v be any two nodes in G with $u \in C_i$ and $v \in C_j$. Let nodes s and t represent the components C_i and C_j in H. Suppose that the shortest path between s and t in H is of length 1. In this case, let x be the articulation point that C_i and C_j share in G. The shortest uv-path in G can be obtained by joining the shortest path between u and x from C_i and the shortest path from x to v in C_j.

The above can be extended to handle the case when the shortest path between nodes corresponding to C and C' in H has a length more than 1. In this case too, the shortest uv-path can be obtained as follows. Let $P = C_1 = x_0 x_1 \cdots x_r = C_2$ be the shortest path in H between C_1 and C_2. Let $(x_{i-1}x_i) = \langle a_i, c_{i-1}, c_i \rangle$ for $i = 1, 2, \cdots, r$. Then, the shortest uv-path in G is obtained by concatenation of the shortest paths between u to $a_1, a_2, a_3 \cdots$, and a_i, v.

Since articulation points in G belong to possibly multiple BCCs, we need to apply the above computation to only every pair of distinct nodes that lie in separate BCCs.

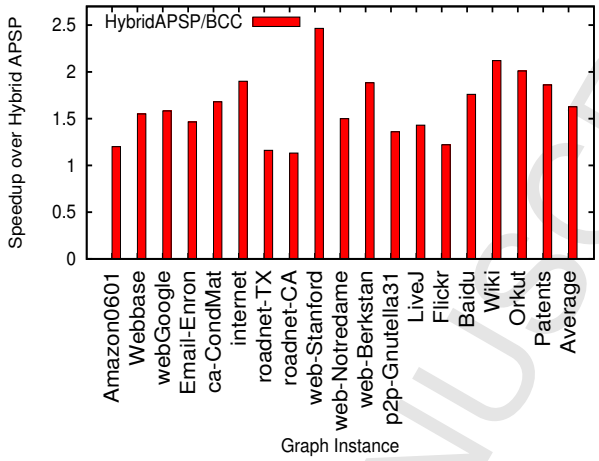

Figure 15: Comparing the overall performance improvement of our approach with respect to a baseline implementation. The Y-axis shows the ratio of the time taken by the baseline to our implementation. The last point on the X-axis is the average of the dataset.

7.2. Experimental Results

In Table 2, we show the graphs that we consider in our experiments along with the number of BCCs that these graphs have and the size of the largest BCC. It can be observed that the graphs chosen for our experiments have a large number of biconnected components and also that the size of the largest biconnected component ranges between 50% to 90% of the edges. Thus, our dataset is fairly rich in its variety. We study the results of our approach by conducting the following experiments.

7.3. Overall Improvement

We consider the overall improvement obtained by our approach compared to the current best known implementation for APSP on the same platform. On the graphs from Table 2, we show the relative improvement of our approach compared to an heterogeneous implementation which is presented in Section 6.2. The results are shown in Figure 15. In Figure 15, the text Hybrid APSP refers to the baseline implementation as described above, and the text BCC refers to our implementation of Algorithm 7.

As can be observed, our approach offers an average improvement of 1.57x compared to the baseline implementation. The overall results are shown in Figure 15. The improvement can be noticed even in cases where the graph has a giant biconnected component, of the order of 80% of the overall edges. A giant BCC does not hinder our program as we utilize data parallelism to find the shortest paths between pairs of vertices within the giant BCC. Further, as we use both the CPU and the GPU, once the CPU finishes working on the small BCCs, we are able to achieve work balance between the CPU and the GPU.

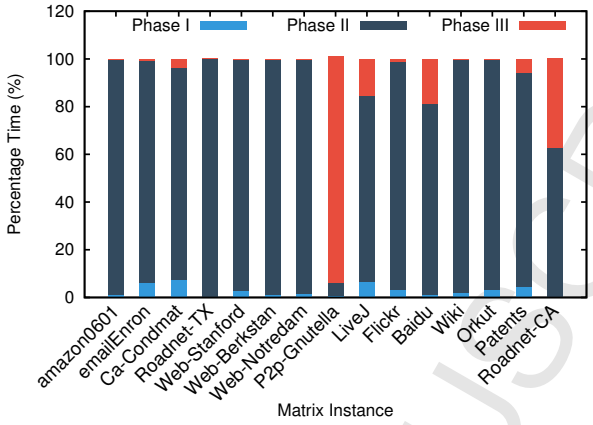

Figure 16: Profile of time taken across various phases of Algorithm 7.

7.4. Profiling

We now study the percentage of time our implementation spends across the various phases of our algorithm described in Algorithm 7. This study will also help in justifying some of our choices during the implementation. We refer to the time taken by Phase I of Algorithm 7 as the preprocessing time, the time taken by Phase II of Algorithm 7 as the compute time, and the rest of the time as the post-processing time. The post-processing time therefore includes the time taken to find the shortest paths in the block graph H and also the time taken to find shortest paths between pairs of vertices in different BCCs of G.

The results of this study are shown in Figure 16. As can be seen, the time taken for preprocessing is really a small fraction of the overall time. (The Yaxis in Figure 16 uses a logarithmic scale.) Thus, the choice of using the sequential algorithm of Tarjan [17] to find the bridges and articulation points is justified. In all cases, finding the shortest paths between pairs of vertices in each BCC consumes the largest amount of time.

8. 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. We received good speedups in all the workloads that we experimented with. This result clearly proves the need to perform data centric pre-processing and modifications that can lead to huge benefits.

In the applications we studied in this paper, we needed to prune the pendant vertices. We also studied an application where one can prune bridges and articulation points that induce partitions in the graph. In future, we wish to study further properties that will lead to the discovery of other pruning strategies.

From the implementations side, we wish to explore the possible ways of designing better hybrid algorithms which will span not only a single CPU+GPU platform but will have multiple CPUs and GPUs connected together.

9. References

[1] Stanford network analysis project. http://snap.stanford.edu/.

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

[3] AGARWAL, V., PASETTO, F. P. D., AND BADER, D. A. Scalable graph exploration on multicore processors. In *Proc. of ACM SC* (10), p. 111.

[4] ANDERSON, R. J., AND MILLER, G. L. A Simple Randomized Parallel Algorithm for List-Ranking. *Information Processing Letters 33*, 5 (1990), 269–273.

[5] AUER, S., BIZER, C., KOBILAROV, G., LEHMANN, J., CYGANIAK, R., AND IVES, Z. DBpedia: A Nucleus for a Web of Open Data. In *Proceedings of the 6th International The Semantic Web and 2Nd Asian Conference on Asian Semantic Web Conference* (Berlin, Heidelberg, 2007), ISWC'07/ASWC'07, Springer-Verlag, pp. 722–735.

[6] BACKSTROM, L., HUTTENLOCHER, D., KLEINBERG, J., AND LAN, X. Group Formation in Large Social Networks: Membership, Growth, and Evolution. In *Proceedings of the 12th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining* (New York, NY, USA, 2006), KDD '06, ACM, pp. 44–54.

[7] 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.

[8] BANERJEE, D., SHARMA, S., AND KOTHAPALLI, K. Work efficient parallel algorithms for large graph exploration. In *20th International Conference on High Performance Computing (HiPC), 2013* (Dec 2013), pp. 433–442.

[9] BANERJEE, D. S., AND KOTHAPALLI, K. Hybrid Algorithms for List Ranking and Graph Connected Components. In *Proc. of 18th Annual International Conference on High Performance Computing (HiPC)* (2011).

[10] BATAGELJ, V., AND MRVAR, A. Pajek network: connectivity of internet routers.

[11] BEAMER, S., ASANOVIĆ, K., AND PATTERSON, D. Direction-optimizing Breadth-first Search. In *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis* (Los Alamitos, CA, USA, 2012), SC '12, IEEE Computer Society Press, pp. 12:1–12:10.

[12] BLACKFORD, L. S., CHOI, J., CLEARY, A., D'AZEUEDO, E., DEMMEL, J., DHILLON, I., HAMMARLING, S., HENRY, G., PETITET, A., STANLEY, K., WALKER, D., AND WHALEY, R. C. *ScaLAPACK user's guide*. Society for Industrial and Applied Mathematics, Philadelphia, PA, USA, 1997.

[13] CHAKRABARTI, D AND ZHAN,Y AND FALOUTSOS,C. R-MAT: A recursive model for graph mining. In *Proceedings of 2004 SIAM International Conference on Data Mining*, SDM '04.

[14] CHHUGANI, J., SATISH, N., KIM, C., SEWALL, J., AND DUBEY, P. Fast and Efficient Graph Traversal Algorithm for CPUs: Maximizing Single-Node Efficiency. In *Parallel Distributed Processing Symposium (IPDPS), 2012 IEEE 26th International* (2012), pp. 378–389.

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

[16] CONG, G., AND BADER, D. A. An Experimental Study of Parallel Biconnected Components Algorithms on Symmetric Multiprocessors (SMPs). In *Proceedings of the 19th IEEE International Parallel and Distributed Processing Symposium (IPDPS'05) - Papers - Volume 01* (Washington, DC, USA, 2005), IPDPS '05, IEEE Computer Society, pp. 45.2–.

[17] CORMEN, T., LEISERSON, C., RIVEST, R., AND STEIN, C. Introduction to algorithms, 2001.

[18] DJIDJEV, H., THULASIDASAN, S., CHAPUIS, G., ANDONOV, R., AND LAVENIER, D. Efficient multi-gpu computation of all-pairs shortest paths. In *Proc. of IEEE IPDPS* (2014). To Appear.

[19] 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.

[20] GHARAIBEH, A., COSTA, L. B., SANTOS-NETO, E., AND RIPEANU, M. On Graphs, GPUs, and Blind Dating: A Workload to Processor Matchmaking Quest. In *Proceedings of the 2013 IEEE 27th International Symposium on Parallel and Distributed Processing* (Washington, DC, USA, 2013), IPDPS '13, IEEE Computer Society, pp. 851–862.

[21] GREINER, J. A comparison of parallel algorithms for connected components. In *Proceedings of the sixth annual ACM symposium on Parallel algorithms and architectures* (1994), SPAA '94, ACM, pp. 16–25.

[22] HARISH, P., AND NARAYANAN, P. J. Accelerating Large Graph Algorithms on the GPU using CUDA. In *Proc. of HiPC* (2007).

[23] HONG, S., OGUNTEBI, T., AND OLUKOTUN, K. Efficient parallel graph exploration for multi-core CPU and GPU. In *IEEE Parallel Architectures and Compilation Techniques (PACT)* (2011).

[24] HONG, S., RODIA, N. C., AND OLUKOTUN, K. On Fast Parallel Detection of Strongly Connected Components (SCC) in Small-world Graphs. In *Proceedings of SC13: International Conference for High Performance Computing, Networking, Storage and Analysis* (New York, NY, USA, 2013), SC '13, ACM, pp. 92:1–92:11.

[25] J. LESKOVEC, J. KLEINBERG, C. F. Graph evolution: Densification and shrinking diameters. *ACM Transactions on Knowledge Discovery from Data (ACM TKDD) 1*, 1 (2007).

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

[27] KARYPIS, G., AND KUMAR, V. Parallel multilevel k-way partitioning scheme for irregular graphs. In *Proceedings of the 1996 ACM/IEEE conference on Supercomputing* (Washington, DC, USA, 1996), Supercomputing '96, IEEE Computer Society.

[28] KATZ, G. J., AND KIDER, JR, J. T. All-pairs shortest-paths for large graphs on the GPU. In *Proceedings of the 23rd ACM SIGGRAPH/EUROGRAPHICS Symposium on Graphics Hardware* (2008), GH '08, Eurographics Association, pp. 47–55.

[29] LESKOVEC, J. Email communication network from enron.

[30] LESKOVEC, J. Gnutella peer to peer network.

[31] LESKOVEC, J., ADAMIC, L. A., AND HUBERMAN, B. A. The Dynamics of Viral Marketing. *ACM Trans. Web 1*, 1 (May 2007).

[32] LESKOVEC, J., KLEINBERG, J., AND FALOUTSOS, C. Graphs over Time: Densification Laws, Shrinking Diameters and Possible Explanations. In *Proceedings of the Eleventh ACM SIGKDD International Conference on Knowledge Discovery in Data Mining* (New York, NY, USA, 2005), KDD '05, ACM, pp. 177–187.

[33] LESKOVEC, J., LANG, K., DASGUPTA, A., AND MAHONEY, M. Community structure in large networks: Natural cluster sizes and the absence of large well-defined clusters.

[34] LESKOVEC, J., LANG, K. J., DASGUPTA, A., AND MAHONEY, M. W. Community Structure in Large Networks: Natural Cluster Sizes and the Absence of Large Well-Defined Clusters. *Internet Mathematics 6*, 1 (2009), 29–123.

[35] MATSUMOTO, K., NAKASATO, N., AND SEDUKHIN, S. Blocked All-Pairs Shortest Paths Algorithm for Hybrid CPU-GPU System. In *2011 IEEE 13th International Conference on High Performance Computing and Communications (HPCC)* (2011), pp. 145–152.

[36] 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.

[37] MISLOVE, A., KOPPULA, H. S., GUMMADI, K. P., DRUSCHEL, P., AND BHATTACHARJEE, B. Growth of the Flickr Social Network. In *Proceedings of the First Workshop on Online Social Networks* (New York, NY, USA, 2008), WOSN '08, ACM, pp. 25–30.

[38] MUNGUIA, L.-M., BADER, D. A., AND AYGUADÉ, E. Task-based parallel breadth-first search in heterogeneous environments. In *HiPC* (2012), pp. 1–10.

[39] NIU, X., SUN, X., WANG, H., RONG, S., QI, G., AND YU, Y. Zhishi.Me: Weaving Chinese Linking Open Data. In *Proceedings of the 10th International Conference on The Semantic Web - Volume Part II* (Berlin, Heidelberg, 2011), ISWC'11, Springer-Verlag, pp. 205–220.

[40] PATTABIRAMAN, B., PATWARY, M., GEBREMEDHIN, A., LIAO, W.-K., AND CHOUDHARY, A. Fast Algorithms for the Maximum Clique Problem on Massive Sparse Graphs. In *Algorithms and Models for the Web Graph*, A. Bonato, M. Mitzenmacher, and P. Praat, Eds., vol. 8305 of *Lecture Notes in Computer Science*. Springer International Publishing, 2013, pp. 156–169.

[41] 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.

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

[43] SHILOACH, Y., AND VISHKIN, U. An O(log n) Parallel Connectivity Algorithm. *J. Algorithms* (1982), 57–67.

[44] 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.

[45] TARJAN, R. E., AND VAN LEEUWEN, J. Worst-case Analysis of Set Union Algorithms. *J. ACM 31*, 2 (Mar. 1984), 245–281.

[46] VENKATARAMAN, G., SAHNI, S., AND MUKHOPADHYAYA, S. A blocked all-pairs shortest-paths algorithm. *J. Exp. Algorithmics 8* (Dec. 2003).

[47] VILLA, O., SCARPAZZA, D., PETRINI, F., AND PEINADOR, J. Challenges in mapping graph exploration algorithms on advanced multi-core processors. In *Parallel and Distributed Processing Symposium, 2007. IPDPS 2007. IEEE International* (2007), pp. 1–10.

[48] WEI, Z., AND JAJA, J. Optimization of linked list prefix computations on multithreaded gpus using cuda. In *The 24th IEEE International Parallel and Distributed Processing Symposium (IPDPS)* (April 2010).

[49] XIA, Y., AND PRASANNA, V. K. Topologically Adaptive Parallel Breadth Frst Search on Multicore-Processors. In *in Proc. PDCS* (2009).

[50] YANG, J., AND LESKOVEC, J. Defining and Evaluating Network Communities Based on Ground-truth. In *Proceedings of the ACM SIGKDD Workshop on Mining Data Semantics* (New York, NY, USA, 2012), MDS '12, ACM, pp. 3:1–3:8.

Appendix-1

Absolute Performance Results For Each Implementation.							
Graph	**BFS (MTEPS)**		**CC (ms)**		**APSP (ms)**		
	Our Time	From [38]	Our Time	From [9]	Our Time	BCC Pruning	From [28]
amazon0601	1065	863	48.54	65.19	390.10	300.43	568.38
email-Enron	780	706	5.54	7.55	2.80	1.96	4.33
ca-Condmat	654	548	1.13	1.59	2.71	1.63	4.07
Roadnet-TX	1077	842	45.56	67.33	1980.23	1500.23	3167.15
Web-Stanford	1008	889	31.76	45.51	175.49	73.13	246.09
Web-Berkstan	884	771	107.56	151.09	774.28	430.12	1126.90
Web-Notredam	913	782	17.56	22.90	61.80	40.13	87.41
p2p-Gnutella	320	269	1.85	2.76	1.37	0.91	2.13
LiveJ	4125	3117	1013.44	1540.42	89508.16	62593.12	130281.28
Flickr	3356	2612	487.85	697.76	38058.77	31453.43	57975.69
Baidu	2255	1631	206.45	371.60	23896.09	13654.31	35471.45
Wiki	3867	3216	1941.12	2818.08	2445298.81	1153442.74	3314993.52
Orkut	2154	1700	183.22	375.69	24116.97	12431.34	40462.25
Patents	2231	1579	243.34	462.99	27220.96	14874.84	43372.24
Roadnet-CA	1173	854	52.23	85.94	5406.23	5300.23	8830.11

Table 3: The absolute results from our experiments on the graphs from Table 1. The metric MTEPS refers to Million Traversed Edges per Second, which is used for reporting results on graph traversals.

Author Bios

Dip Sankar Banerjee:

Dip Sankar Banerjee is presently a PhD student at the International Institute of Information Technology, Hyderabad, India where he is affiliated to the Center for Security Theory and Algorithmic Research. Prior to joining the doctoral he pursued his undergraduate studies in computer science engineering from the West Bengal University of Technology, India. His research interests are broadly in parallel algorithms, massive graph analysis, multicore and manycore computing and high performance computing.

Meher Chaitanya:

Meher Chaitanya is currently a MS by research student at the International Institute of Information Technology, Hyderabad, India where he is affiliated to the Center for Security Theory and Algorithmic Research. Prior to this he was a a software engineer at Nvidia. He completed his Mtech in Computer Science Engineering from the same institute. His current areas of research are studying various ways of separating graphs, and their cost-benefit analysis with respect to problems such as shortest paths, and betweenness centrality.

Ashutosh Kumar:

Ashutosh Kumar did his undergraduate studies in Computer Science and Engineering from IIIT Hyderabad and is currently working as a Software Development Engineer at Microsoft. His research interestes are on secure communication protocols in adversarial distributed networks and in parallel algorithms.

Shashank Sharma:

Shashank Sharma is currently a Masters student at the International Institute of Information Technology, Hyderabad, India where he is affiliated to the Center for Security Theory and Algorithmic Research. Prior to this he completed his undergraduate degree in Computer Science Engineering from the same institute. His major research interests are in parallel computing, GPU computing and graph theory.

Kishore Kothapalli:

Kishore Kothapalli is presently an Associate Professor at the International Institute of Information Technology, Hyderabad, where he is working since 2006. Prior to that, he obtained his doctoral degree in Computer Science from the Johns Hopkins University, USA, and his Master's degree in Computer Science from Indian Institute of Technology, Kanpur. His current research interests are in parallel algorithms for problems on graphs, sparse matrices, and the like. He is also interested in data structures for geometric problems.

Work Efficient Parallel Algorithms for Large Graph Exploration on Emerging Heterogeneous Architectures.

- Processing real world graphs in an efficient manner through input pruning.
- Two different pruning strategies based on 1-degree nodes and articulation points.
- Improvements upto 35% or 1.57x over current best known results.
- Experimental evaluation of algorithms proposed on several real world graphs.
- Heterogeneous multicore implementation provides better performance efficiency.