Graph Algorithms in Parallel and Heterogeneous External Memory Model

Anonymous Author(s)

Abstract

Efficient algorithms for graph connectivity (including connected components and biconnected components) and minimum spanning tree (MST) computations are vital in diverse domains, from medical diagnostics to chip design. Yet, as graph sizes grow to billions of edges, existing approaches often fail due to limited device memory. This challenge demands innovative rethinking of both algorithmic design and implementation strategies.

We address this need by presenting a novel Parallel Heterogeneous External Memory (PHEM) framework, which harnesses both multicore CPU and GPU resources to tackle large-scale graph processing. Our method minimizes communication overhead while maximizing computational throughput through efficient copy- computation strategies. Experiments on a variety of real-world and synthetic benchmarks demonstrate that our algorithms for connected components, biconnected components, and minimum spanning tree in the PHEM model outperform current state-of-the-art solutions, significantly advancing the capability to process massive graphs.

CCS Concepts

• **Theory of computation** → **Massively parallel algorithms**; **Shared memory algorithms**.

Keywords

Large-scale graphs, CC, BCC, MST, Heterogeneous, Out-of-core processing, GPU, Batch processing

1 Introduction

Over the past decade, substantial research has been done on large graph analytics using parallel architectures and accelerators, such as multi-core CPUs and GPUs. This research has resulted in an extensive literature, including parallel algorithms, implementations, libraries, Domain Specific Languages, benchmark datasets, and programming frameworks. This progress has enabled the execution of various applications from diverse domains such as social networks, computational epidemiology, scientific applications, and the like. We refer the interested reader to recent surveys, viz. [32, 33], for a comprehensive summary.

One noticeable aspect in recent years is that the size of graph datasets is ever-increasing, with graphs containing billions of nodes and edges becoming commonplace. For example, the HyperLink graph representing web pages and hyperlinks between web pages has 3.5 billion nodes and 128 billion edges, requiring more than 1000 GB of space to store the graph! A significant challenge is effectively handling such large real-world graphs.

While modern server-class CPUs have RAM capacities in the range of hundreds of gigabytes, the memory on current-generation accelerators, such as GPUs, is limited to tens of gigabytes*. Despite these memory constraints, GPUs have emerged as a powerful solution for general-purpose computation over the last two decades due to their massive parallelism and high memory bandwidth. These architectural advantages also make GPUs exceptionally attractive for graph algorithms through careful parallelism-friendly algorithmic solutions that focus on mitigating high global memory latency, thread synchronization, load imbalance, and data transfer overheads. As the size and complexity of graph-structured data continue to grow, addressing the limited on-board memory of GPUs through innovative *out-of-memory* strategies presents new challenges.

Recent advances in large graph processing span heterogeneous computation, out-of-core methods, and distributed paradigms, yet each has trade-offs. Heterogeneous execution models [27] enable overlapping CPU and GPU computation but often require the entire graph (or significant portions) to reside in GPU memory, incurring overheads from partitioning and risking capacity limits. Approaches like Wang et al. [40] and Sengupta et al. [34] circumvent GPU space constraints by iteratively streaming subgraphs to the GPU, they neither exploit heterogeneous execution (which can yield 20–30% performance gains [4, 42]) nor mandate that partial outputs also fit in limited GPU memory.

GPU-accelerated systems support large input sizes through the Universal Virtual Memory (UVM), which brings GPU and host memories into a single address space. UVM allows transparent migration of data between the two memories using paging. Several works have proposed techniques for efficient management of large data transfers over UVM for graph algorithms [1, 28]. The zero-copy mechanism allows data transfer at a granularity level of cache lines and a minimum of 32 bytes. The zero-copy mechanism requires applications to pin memory allocations on the host side to prevent them from being swapped out. Min et al. [23] investigate using the zero-copy mechanism for out-of-memory graph algorithms. UVM and zero-copy, while providing significant abstractions to manage large data inputs, still inherently depend upon the interconnect bandwidths and do not provide mechanisms at the hardware level for hiding the transfer latencies. Specifically, this problem becomes more pronounced for irregular memory accesses commonly observed in graph applications.

The other research direction that has seen efforts at processing large graphs is the work of GraphChi [17], and Blelloch et al. [5], on external memory algorithms. These frameworks focus more on optimizing aspects such as external memory management, cache coherence, and memory wear. Further, their execution is mainly centered around multi-core CPUs and does not cater to heterogeneity.

ICPP'25, San Diego, CA
2025. ACM ISBN 978-1-4503-XXXX-X/2018/06
https://doi.org/10.1145/nnnnnnn.nnnnnnn

*For example, the recent NVIDIA H100 GPU offers 80 GB of VRAM.

The above situation calls for a comprehensive study of large graph algorithms that utilize the benefits of parallelism and heterogeneity and develop systematic techniques that look at computations and memory management in a unified manner. In this paper, we present a model, the *Parallel Heterogeneous External Memory* (PHEM) model for processing large graphs in a heterogeneous system of a multi-core CPU and a GPU, and seek algorithms for popular graph problems in this model. In particular, we present algorithms in the PHEM model for three popular graph problems: connected components (CC), biconnectivity (BCC), and minimum spanning tree (MST). For each of these problems, we design efficient PHEM algorithms and their implementations. In addition, we demonstrate that our implementations outperform comparable state-of-the-art implementations on large graphs on two heterogeneous systems. The entire source code datasets for our result and the extended version of this work with proofs are available at https://github.com/PHEM-Graph/PHEM-Graph.

1.1 Related Work

We review some prominent frameworks for large graph processing aimed at CPU and GPU-based systems such as HyPar [27], EMOGI by Min et al. [23], and GraphReduce [34] exploit GPU resources to accelerate graph computations. HyPar exploits heterogeneous execution and partitions the graph into two parts, one part each for processing on the CPU and the GPU, but requires that the GPU part fit in the GPU memory. Due to the space limitations of the GPU and its higher throughput, it can complete computations on its assigned part much faster than the CPU. This is a drawback because the GPU waits for the CPU to complete its execution on its assigned graph. GraphReduce dynamically partitions graphs into shards that GPUs can process, offloading excess data to host memory, but does not use heterogeneity. The EMOGI framework focuses more on optimizing the transfer of graph shards via novel enhancements to the zero-copy transfer mechanism and does not employ heterogeneous execution.

Frameworks such as GraphChi [17] and ROSE [5] focus on optimizing external memory management and do not address execution on accelerators or heterogeneous execution. Popular distributed graph processing platforms, such as PowerGraph [13] and Pregel [21] introduced distributed graph processing paradigms aimed at scaling graph analytics across multiple machines. While these systems distribute computational loads across multiple nodes, they focus on optimizing communication across processing nodes.

Table 1 summarizes the various features of existing works on large graph algorithms in the parallel setting. We notice from Table 1 that even as existing systems demonstrate significant advances, they fail to address the need for a more comprehensive, flexible, and scalable solution that leverages external memory and heterogeneous resources such as CPUs and GPUs. This situation calls for a comprehensive solution that addresses recent challenges in large graph algorithms coupled with the features of existing parallel architectures.

‡‡As the GPU may get a smaller portion of work, the GPU may be idle until the CPU finishes its work.

Table 1: Table shows the features of various approaches to deal with large graphs in parallel.

Work	Execution	Handles graph that do not fit in GPU	GPU Data Fetching
[34, 40]	GPU only	Yes	On-demand
[27]	Heterogeneous	Yes	One time ‡‡
[18]	CPU	N-A	External Memory
[20, 23, 31]	GPU only	Yes	Zero Copy/UVM
Ours	**Heterogeneous**	**Yes**	**Batched**

The three graph algorithms we consider in this paper have been well-studied in the extant literature. In the following, we note some of these works in the context of heterogeneous algorithms.

Past Work on Connected Components. Some popular works using GPUs include the early work of GPU-CC [37], GSWITCH [22], GConn [15], and ECL-CC [16]. These algorithms employ a variety of techniques, including the Shiloach-Vishkin hooking and pointer-jumping method [35], the union-find data structure, and implementation optimizations such as lock-free asynchrony with load balancing [16]. The current state-of-the-art algorithm for GPUs is the GConn algorithm of Hong et al. [15]. Similarly, some popular works on multi-core CPUs include LACC by Zhang et al. [43], the ConnectIt algorithm of Dhulipala et al. [9], and Ligra by Shun et al. [36]. These algorithms use approaches ranging from the union-find data structure and label propagation to graph traversal and exploiting linear algebraic properties of graphs.

Past work on Biconnected Components. On the other side of the design spectrum, given a graph G on n vertices and m edges, many parallel algorithms for BCC construct a skeleton graph G' from G and obtain BCC of G from CC of G'. In Tarjan-Vishkin's algorithm for BCC, vertices in G' correspond to edges in G. In the FAST-BCC algorithm by Dong et al. [11], the skeleton graph G' is a subgraph of G. The key merit of their algorithm is that it requires $O(n)$ auxiliary space. However, the number of edges of the skeleton graph G' in the above two works is $\Theta(m)$ and need not fit in the GPU. Early work on parallel BCC computation, such as Wadwekar and Kothapalli's GPU-based approach, uses a BFS-based strategy to construct an auxiliary graph.

Past Work on MST. Most parallel MST algorithms, e.g., [38, 41], use the algorithm of Boruvka's [24]. This algorithm iteratively adds edges to a growing Minimum Spanning Forest (MSF) by selecting the lightest edge from each vertex in each iteration. This process effectively reduces the number of trees in the forest by a factor of at least 2, hence converging in $O(\log n)$ iterations.

1.2 Our contributions

From the preceding discussion, we posit that designing a heterogeneous, scalable framework for processing massive graphs poses significant challenges. This paper addresses these challenges by first introducing a novel algorithmic framework. Our key contributions are summarized as follows:

- We model, design, and implement a scalable, modular framework for out-of-memory graph processing computation, enabling efficient processing of massive graphs that do not fit into GPU memory. **(cf. Section 2)**
- We use parallel dynamic graph connectivity algorithms to design an algorithm for CC in the PHEM model. **(cf. Section 3)**

- We design a novel PHEM algorithm for BCC by characterizing a small sub-graph G' of G, such that $size(G') = O(n)$ and the biconnected components of G can be obtained from the connected components of G', where $n := |V(G)|$. **(cf. Section 4)**
- We adapt the algorithm of Boruvka [24] to design a PHEM algorithm for MST. **(cf. Section 5)**
- Experimental results of our algorithms on diverse, massive real-world datasets indicate that our algorithms outperform the state-of-the-art comparable algorithms. **(cf. Section 6)**

2 The PHEM Model and A Generic PHEM Algorithm

This section explains the proposed Parallel Heterogeneous External Memory (PHEM) model. In the PHEM model, we consider a computing platform equipped with a host, such as a CPU, and an auxiliary device, such as a GPU, connected to the CPU. The device(s) operate on data already in their local memory. The CPU has space of M_{CPU} Bytes while the auxiliary device has space of M_{GPU}. We assume that $M_{\text{CPU}} \gg M_{\text{GPU}}$. We also assume that the input has a size N Bytes with $N < M_{\text{CPU}}$ [†].

As the input does not fit entirely on the device, algorithms designed in the PHEM model should be able to execute on the device with a partial input. Further, the (partial) output produced on the device must also fit within the device's space. For most non-trivial computations, we require the PHEM algorithm to read and process the entire input at least once on the CPU or GPU. For an algorithm in the PHEM model, one *pass* constitutes the reading/processing of the entire input once. Since these passes have to execute one after the other sequentially, algorithms in the PHEM model should aim to minimize the number of passes of the input needed to complete the execution. Figure 1 shows how to view a PHEM algorithm with $r \geq 1$ passes. For clarity of exposition, only Pass i is shown in detail. The algorithms we present in this paper use either a single pass or two passes of the input.

As this paper deals with graph algorithms, we focus on how the model impacts graph inputs. Let the input be a graph G with n vertices and m edges. We assume that the M_{GPU} is $O(n)$ [‡] space. One can view this limit on the space on the auxiliary device as having the ability to store $O(1)$ data items, on average, including input, auxiliary space, and output, for every vertex of the graph. With this limit, an algorithm in this model allows storing $O(1)$ neighbors of every vertex in the device or storing the level number of a node according to a breadth-first search of the graph. For clarity, we denote by G_{CPU} and G_{GPU} the subgraph of the input graph that the CPU and GPU process, respectively.

Thus, algorithms that work in the PHEM model navigate many challenges to be efficient in practice. Firstly, these algorithms must utilize the available parallelism across heterogeneous computing devices while minimizing the number of passes of the graph. Secondly, the algorithms must work with partial inputs and possibly

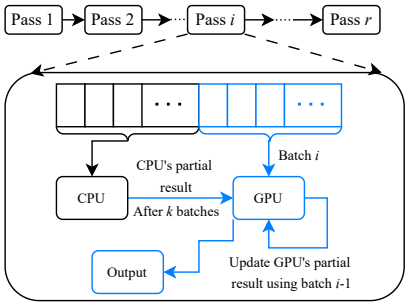


Figure 1: PHEM pipeline

partial outputs since the entire input is never available to any device. Third, these algorithms must introduce techniques to hide data transfer latency across computing devices. Finally, the data structures that these algorithms use would also likely have to be created and maintained across the device boundaries, thereby requiring novel designs.

Multiple parameters exist to measure an algorithm's performance in the PHEM model. Some of these include the number of passes of the input needed, the overall execution time, and the throughput of the algorithm. The time spent by the algorithm is the largest of the time spent by the algorithm utilizing any resource: computation on the CPU or the GPU, or data transfer.

2.1 A Generic PHEM Algorithm

This section provides a generic template of algorithms that run in the PHEM model. An algorithm in the PHEM model executes various steps simultaneously on all the computing devices of the computing platform. The algorithm may make multiple passes over the input, and the portions of the input the algorithm assigns to the computing devices can change over each pass. In any pass, the computation on any device uses only the portion of the input that is resident in the memory of that device to produce a partial output. These partial outputs may be combined as needed to produce an output at the end of each pass or all the passes.

We show the high-level details of one pass in our generic algorithm in Algorithm 1. We represent the input graph G as a sequence $(e_1, e_2, \ldots e_m)$ of edges in the edge-list format. At the beginning, the entire sequence appears in CPU. We now describe the heterogeneous way of processing all edges in the given sequence using CPU and GPU as follows. We partition the sequence of edges into two disjoint sequences E_{CPU} and E_{GPU} in such a way that $\alpha \times m = |E_{GPU}|$ and $(1-\alpha) \times m = |E_{CPU}|$, where α denotes the fraction of edges that appear in GPU. We handle the edges in E_{CPU} and E_{GPU} in CPU and GPU, respectively. Due to space constraints on the GPU, the edges in E_{GPU} are further divided into batches $B_1, \ldots, B_k$ so that B_i, $i \geq 1$, fits in GPU. This algorithm framework allows for two parallel executions, Lines 4–6 on the CPU and Lines 7–13 on the GPU, on different parts of the input. In particular, while CPU is processing on E_{CPU} in parallel, GPU processes the current batch in parallel, and simultaneously, the next batch is getting copied from CPU to GPU.

We describe the details of processing E_{CPU} in CPU and a batch B_i in GPU in CPU-COMPUTE and GPU-COMPUTE functions, respectively.

[†]With current CPUs having RAM of the order of hundreds of GBs, the assumption that $N < M_{\text{CPU}}$ allows for graphs with hundreds of billions of edges also to be stored on the CPU RAM.

[‡]Notice that with modern GPUs having space of the order of tens of GBs, a graph with tens of billions of vertices can be accommodated within these constraints.

One can think of designing the GPU-COMPUTE function as a variant of incremental algorithms. In particular, for $i \geq 1$, GPU-COMPUTE function can take as input the output up to the i^{th} batch, the edges in the i^{th} batch, and any other auxiliary information and produce the output at the end of the i^{th} batch and the necessary auxiliary information required for the next batch[¶]. After CPU computation, the partial solution computed in the CPU is treated as another batch and copied to GPU in Line 6. Finally, in the last two lines of the algorithm, we process the last batch in the GPU using GPU-MERGE, and the solution is copied back to CPU.

Algorithm 1: PHEM Framework for Graph Algorithms

Input: A graph $G = (V, E)$ represented as sequence of edges

1 Partition E into E_{CPU} and E_{GPU} such that $|E_{\text{GPU}}| = \alpha \cdot |E|$;
2 Divide E_{GPU} into k batches $B_1, \ldots, B_k$, so that B_i fits in GPU;
3 **In Parallel**
4 **CPU Block**
5 $B_{k+1} \leftarrow$ `CPUCompute` (E_{CPU});
6 `CopyToGPU(CPUFinalList,` B_{k+1}`)`;
7 **GPU Block**
8 `AsyncCopyToGPU(CurrentList,` B_1`)`;
9 **for** $i \leftarrow 2$ **to** k **do in parallel**
10 $D \leftarrow$ `GPUCompute(CurrentList,` D`)`;
11 `AsyncCopyToGPU(NextList,` B_i`)`;
12 `Swap(CurrentList, NextList)`;
13 $D \leftarrow$ `GPUCompute(CurrentList,` D`)`;
14 $D \leftarrow$ `GPUMerge(CPUFinalList,` D`)`;
15 `CopyToCPU(Output,` D`)`;

The PHEM framework incorporates multiple novel aspects that enable efficient large graph processing. A key advantage of the PHEM framework is its flexibility: both the CPU and the GPU can utilize any algorithm, including incremental algorithms on the GPU. Our framework requires users to specify just the three subroutines GPU-Compute, CPU-Compute, and GPU-Merge to enable any graph computation to use the generic algorithm framework and arrive at efficient solutions for large graph processing. The framework is inherently heterogeneous in nature. The framework also limits the amount of auxiliary space that GPU algorithms can use so that they continue to run on any large input. Further, the framework extends to multiple devices and to multi-GPU settings seamlessly.

3 First Application: Connected Components and Spanning Forest

This section focuses on finding connected components (CC) for massive graphs that exceed on-board memories. Throughout this paper, we use G to denote a graph formed on (V, E), where V and E denote the set of vertices and edges, respectively. We use m and n to denote the number of edges and vertices in G, respectively. Given an undirected graph $G = (V, E)$, the *Connected Components* (CC) problem partitions V into disjoint subsets $V_1, V_2, \ldots, V_k$, where each (V_i, E_i) is a *maximally connected* subgraph of G. The objective is to identify these connected components and label each vertex by its component's unique label. Almost all CC algorithms can construct a spanning tree (or spanning forest) as a byproduct with some minimal bookkeeping [15, 29, 37].

3.1 PHEM-CC Algorithm

One common strategy is to process the graph in smaller batches, each of which fits in memory. We apply partial connectivity or union-find techniques to build or maintain a spanning forest during each batch. After processing all batches, we merge the forests into a single spanning tree (or forest). This approach necessitates that GPU-COMPUTE can work in the presence of only partial inputs and a partial output. At the same time, CPU-COMPUTE bypasses this prerequisite as the graphs fit in the main memory (see Section 2). Thus, a static connectivity algorithm best suits CPU-COMPUTE, while an incremental connectivity algorithm works best for GPU-COMPUTE.

We now describe the steps involved in our **PHEM-CC Algorithm** to solve connected components problem in a heterogeneous manner. The process begins by partitioning the input edges based on a parameter α, which specifies the fraction of edges that the GPU processes, with the remaining edges assigned to the CPU. Given the potentially large data size, we further split the GPU edges into k batches, where each batch of edges is loaded incrementally and processed.

Procedure GPU-COMPUTE. We process each batch of edges in the GPU using the GPU-COMPUTE subroutine and update the existing spanning forest maintained in GPU. A GPU-optimized incremental connected components algorithm, GConn [15], helps to achieve the functionality of GPU-COMPUTE. To address the high cost of data transfers on GPU, the PHEM framework employs a key optimization: overlapping data transfer and computation. While the GPU processes one batch, the next batch is transferred simultaneously, effectively hiding the copy time with computation.

Procedure CPU-COMPUTE. In parallel, the CPU constructs a partial spanning forest using the CPU-COMPUTE subroutine from its assigned edges. We use the parallel static connected components algorithm from ConnectIt [10] to achieve this. As a result, the CPU and the GPU construct two partial spanning forests independently.

Final Merge. As part of the final merge process, the CPU's spanning forest is transferred to the GPU, where the two forests are merged into a single spanning tree/forest using the *Merge* subroutine.

4 Second Application: Biconnected Components

In the previous section, we showed how to use the PHEM framework to adopt an existing connected component (CC) algorithm. As discussed earlier, while existing approaches for finding BCCs are well understood and efficient in RAM-based or parallel shared-memory settings, their direct adoption in PHEM is not straightforward due to associated space constraints. In light of this challenge, we propose a BCC algorithm specifically tailored to the PHEM model.

Throughout this section, we use G to denote an unweighted, undirected, and connected graph. In a graph G, v is a *cut vertex* if $G - v$ is disconnected, and e is a *cut-edge* if $G - e$ is disconnected. A *biconnected component* of G is a maximal subgraph without any cut

[¶]An incremental graph algorithm usually has access to the entire input and no space constraints to store a data-structure, whereas our GPU-COMPUTE function has access only to the current batch.

vertex. Given an undirected graph G, the *Biconnected Components* (BCC) problem partitions the *edges* of G into maximal subgraphs $G_1, G_2, \ldots, G_k$ such that each G_i is a biconnected component.

We begin by defining relevant building blocks (Section 4.1) that are important to understand our work, then describe our BCC algorithm to work in PHEM model (Section 4.2).

4.1 Building Blocks

We use the following subroutines in our PHEM algorithm for BCC.

Euler Tour. We use the well-known Parallel Euler Tour Technique [29] to convert an unrooted tree into a rooted one and to generate an Euler tour starting from a specified root. We use T to denote the resulting rooted tree. Instead of edges, we represent the Euler tour as a sequence of vertices. For each vertex u in T, $first[u]$ and $last[u]$ indicate the positions of its first and last appearances in the tour. Figure 2(a) illustrates a rooted tree with its Euler tour, with each vertex labeled with its corresponding first and last indices.

Low-High Tags. For a vertex u, let T_u denote the subtree of T rooted at u. We then write $low[u]$ to denote the smallest $first[\cdot]$ value of all the reachable vertices via a non-tree edge from vertices in T_u. Similarly, $high[\cdot]$ is the maximum value. Formally, both $low[\cdot]$ and $high[\cdot]$ can be written as:

$$low[u] = \min\{w_1[u] \mid u \in V(T_v)) \text{ where,}$$
$$w_1[u] = \min\left(\{first[u]\} \cup \{first[u'] \mid (u,u') \in E(G) - E(T)\}\right)$$
$$high[u] = \max\{w_2[u] \mid u \in V(T_v)) \text{ where,}$$
$$w_2[u] = \max\left(\{first[u]\} \cup \{first[u'] \mid (u,u') \in E(G) - E(T)\}\right)$$

Types of edges. In a graph G with a rooted spanning tree T, edges fall into four categories. **Tree edges** are those in T, while **non-tree edges** are all others. A non-tree edge (u, v) is a **back edge** if one of the vertices is an ancestor of the other in T (satisfies $first[u] < first[v]$ and $last[v] < last[u]$ (or) vice versa); otherwise, it is a **cross edge**. For a vertex v, let $par(v)$ denote its parent. We further divide tree edges into fence edges and plain edges. A tree edge $(par(v), v)$ is a **fence edge** if every non-tree edge from the subtree rooted at v stays within the subtree rooted at $par(v)$ (satisfies $first[par(v)] \le low[v]$ and $high[v] \le last[par(v)]$); otherwise, it is a **plain edge**. Figure 2(b) illustrates these edge types.

BCCs in Sparse Representation. We use two arrays, $label[\cdot]$ and $componentHead[\cdot]$ ([3, 11]), to represent the BCCs of a graph in a space-efficient manner. For a graph G and a rooted spanning tree T of G, for each non-root vertex u in T, $label[u]$ denotes a label from $\{1, \ldots, k\}$, where k denotes the number of BCCs in G. For each label $i \in \{1, \ldots, k\}$, $componentHead[i]$ indicates a special vertex called component head or BCC head. All vertices with the same label and component head together form a BCC.

FAST-BCC Algorithm [11]. The most important idea in this algorithm is to obtain a skeleton graph G' from G, with the plain edges of T and all cross edges in $G - T$, and obtain a sparse representation of BCC using the connected components of G'.

4.2 PHEM-BCC Algorithm

We start this section with the necessary notation and key ideas in our PHEM algorithm for BCC. Later, we describe our two-pass algorithm to solve BCC, in which we process all edges in the input graph in **two passes**, where the first pass **P1** constructs a spanning tree. The second pass **P2** obtains a spanning forest out of cross edges and low/high tags.

Algorithm 2: PHEM-BCC ALGORITHM

Input: An undirected graph $G(V, E)$
Output: labels $l[\cdot]$ and componentHead[$\cdot$]

1 **P1-** CPU+GPU: Spanning tree edges $E_T \leftarrow$ PHEM-CC(G)
2 GPU: $(T, first[\cdot], last[\cdot]) \leftarrow$ EULER-TOUR(E_T)
3 Copy $(T, first[\cdot], last[\cdot])$ from GPU to CPU
4 **P2-** CPU+GPU: $(low[\cdot], high[\cdot], F_c) \leftarrow$ COMPUTE-SF-TAGS(G)
/* All subsequent steps are executed on GPU. */
5 $G' \leftarrow$ PLAINEDGE(T) $\cup$ F_c // Form the skeleton graph
6 $l[\cdot] \leftarrow$ CC-LABELS(G') // Assign vertex labels $\forall v \in V$
7 **for** *each vertex* $v \in V \setminus \{r\}$ **do in parallel**
8 **if** $l[v] \neq l[par[v]]$ **then**
9 $componentHead[l[v]] \leftarrow par[v]$

Notation. We use E_c and E_b to denote the sets of back edges and cross edges of $G - T$, respectively, where T is an arbitrary rooted spanning tree of G. Additionally, F_c denotes a spanning forest of the edges in E_c. We refer to the edges in F_c as *essential edges*.

Key Ideas. From Theorem 8.4, we know that two vertices belong to a biconnected component in G if and only if they belong to a biconnected component in $T \cup F_c \cup E_b$. Based on this, we construct a skeleton graph G' of G in a heterogeneous way, where G' consists of the plain edges of T and all the cross edges in F_c, and $|E(G')|$ is $O(n)$. We then apply the ideas from [3, 11] to obtain biconnected components of G in sparse representation from connected components of G', with the help of labels and component heads.

PHEM-BCC Algorithm. The PHEM-BCC algorithm proceeds in five main steps. First, we construct a spanning tree from the given graph G in the first pass, using PHEM-CC algorithm in Section 3. The PHEM-CC algorithm solves the connected components problem and finds a spanning forest of a graph. Since the input graph G is connected, PHEM-CC produces a spanning tree of G. The second step converts an unrooted spanning tree to a rooted spanning tree T using the EULER TOUR TECHNIQUE ([29]) and computes $first[\cdot]$ and $last[\cdot]$ for all vertices in T. The second step requires $O(\log n)$ depth. During the third step, we apply COMPUTE-SF-TAGS to construct a spanning forest F_c on all cross-edges and compute the tags $low[\cdot]$ and $high[\cdot]$ for all vertices using the non-tree edges in F_c and E_b. We defer the details of the algorithm COMPUTE-SF-TAGS that processes the entire graph in the second pass to the following subsection. In the fourth step, we run CC-LABELS algorithm ($O(\log n)$ depth [35]) on the skeleton graph formed by plain edges in T and cross-edges in F_c. The CC-LABELS assign labels to all vertices such that all vertices in the same connected component receive the same label. Finally, in the fifth step (constant depth), we obtain a component head for each label by processing all non-root vertices of T in parallel. For a non-root vertex v, we update the componentHead of $label[v]$ with its parent. All vertices with the same label and their component head together form a BCC. We emphasize that two subroutines PHEM-CC and COMPUTE-SF-TAGS in Line 1 and Line 4 are performed in a

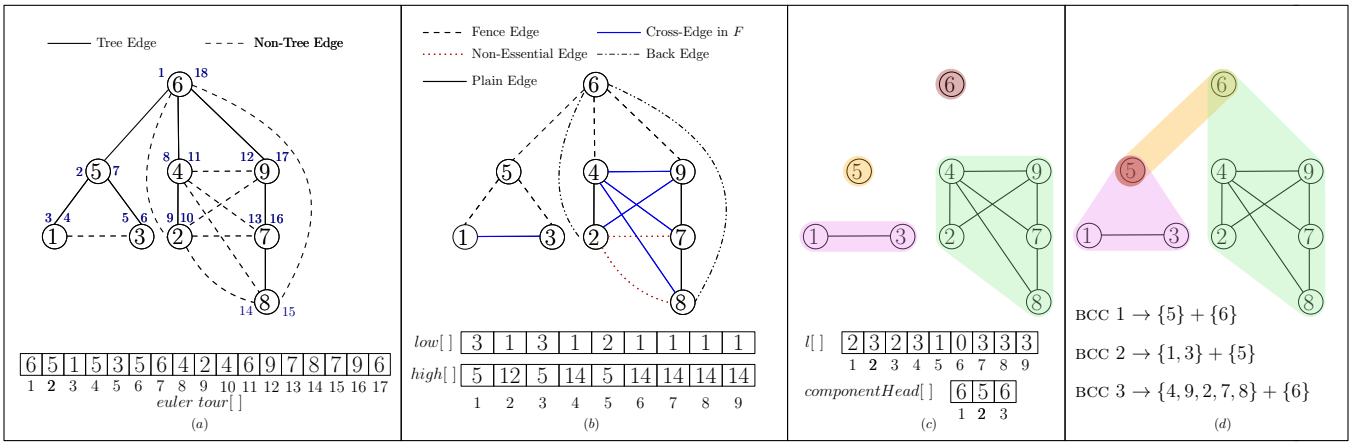


Figure 2: Steps of Algorithm 2: (a) Lines 1-2: (First pass P1) Obtain a rooted spanning tree T of G, its Euler tour, and compute first/last tags. (b) Line-4: (Second pass P2) Compute low/high tags using all back edges and cross edges in F_c. (c) Lines 5-9: Construct a skeleton graph G' with plain edges in T and edges in F_c, find labels for connected components in G', and obtain componentHead for each label except for the label of the root vertex. (d) BCCs of G are shown in sparse representation.

heterogeneous manner, using both CPU and GPU as the complete input graph does not fit on the GPU. In contrast, the rest of the lines are executed on GPU. Figure 2 illustrates the steps of PHEM-BCC algorithm using an example.

Algorithm 3: GPU-Merge-SF-Tags

Input: $w1_{cpu}, w2_{cpu}, w1_{gpu}, w2_{gpu}, F_{cpu}, F_{gpu}, first, last$
Output: Spanning Forest F_c of cross edges, $low[\cdot]$ and $high[\cdot]$

1 F_c = SPANNINGFOREST($F_{cpu} \cup F_{gpu}$)
2 update $w1_{gpu}$ and $w2_{gpu}$ using F_c edges.
3 **for** *each vertex u* **do in parallel**
4 $\quad w1[u]$ = min($w1_{cpu}[u]$,$w1_{gpu}[u]$)
5 $\quad w2[u]$ = max($w2_{cpu}[u]$,$w2_{gpu}[u]$)
6 Obtain $low[\cdot], high[\cdot]$ using $w1$, $w2$, $first$ and $last$

4.2.1 ***COMPUTE-SF-TAGS (Line 4, Algorithm 2).*** We recall that the purpose of this subroutine is to construct a spanning forest F_c on all cross-edges and compute the tags $low[\cdot]$ and $high[\cdot]$ for all vertices using the non-tree edges in F_c and E_b. To achieve this, we use the rooted spanning tree T represented by the parent array $par[\cdot]$ and two arrays, namely $first[\cdot]$ and $last[\cdot]$, which are accessible on both CPU and GPU. At the beginning of this subroutine, the edges in G appear in CPU in edge-stream format. The second round of heterogeneous processing of all edges in G occurs in this subroutine. At a high level, we follow all the steps described in Algorithm 1. The algorithm details of CPU-COMPUTE, GPU-COMPUTE, and GPU-MERGE that correspond to the computation of spanning forest and tags are described below.

GPU-COMPUTE. The GPU processes a batch of edges in parallel and classifies them as a cross edge, back edge, or a tree edge using the $first[\cdot]$, $last[\cdot]$, and $parent[\cdot]$ arrays. We then construct a spanning forest F_{gpu} on the identified cross edges using the incremental spanning forest algorithm [15]. Later, we sort all the back edges (u, v), where v is an ancestor of u, by considering the first endpoint using [26]. We then perform segmented min-reduction on the sorted edges by considering $first[\cdot]$ values of second endpoints, where all edges with the same first endpoint are treated as segments. These two steps help to obtain $w1_{gpu}[\cdot]$ by considering all edges in the current batch, where $w1_{gpu}[u]$ denotes the minimum of all first values of second endpoints of the back edges going out from the subtree rooted at T_u and $first[u]$. $w2_{gpu}[\cdot]$ array is analogues to $w1_{gpu}[\cdot]$, except that it focuses on maximum value. All operations in GPU-COMPUTE take $O(\log n)$ depth.

CPU-COMPUTE. We apply the same steps described in GPU-COMPUTE on the edges assigned to CPU. While processing the edges in CPU, we update spanning forest F_{cpu}, and tags $w1_{\text{cpu}}$, and $w2_{\text{cpu}}$ incrementally. This CPU-COMPUTE consumes $O(\log n)$ depth.

GPU-Merge-SF-Tags. The steps of subroutine are described in Algorithm 3. The arrays $w1_{\text{cpu}}, w2_{\text{cpu}}, F_{\text{cpu}}$, computed by the CPU, are transferred to the GPU for the merge operation. The global spanning forest F_c is generated by combining F_{cpu} and F_{gpu} through a parallel spanning forest computation on the union of their edges, as shown in Line 1 of Algorithm 3. (Line 2) Once F_c is constructed, the arrays $w1_{\text{gpu}}$ and $w2_{\text{gpu}}$ are updated based on the edges in F_c, similar to the procedure used for back edges earlier. For each vertex u, the values $w1[u]$ and $w2[u]$ are derived by combining the CPU and GPU results (Lines 3-5). In the Euler tour $ET[.]$ of T, vertices in u's subtree appear consecutively between $first[u]$ and $last[u]$. We compute $low[u]$ as the minimum $w1$ value in u's subtree using range minimum query on the $ET1[.]$ array, where $ET1[i] = w1[ET[i]]$. Similarly, $high[u]$ is the maximum $w2$ value, computed using a range maximum query. We implemented GPU-optimized Parallel Range Minimum Query, which uses the sparse table approach [6], providing $O(\log n)$ preprocessing and query times with $O(n)$ space.

Processing a set of edges assigned to CPU, as well as processing a batch of edges in GPU, requires $O(\log n)$, as $O(\log m) = O(\log n)$ in both passes. Thus, the total depth is bounded by $O(k \log n)$, where k denotes the number of batches. All the data structures used in our algorithm in GPU consume $O(n)$ space, as each batch fits in GPU and GPU holds $O(n)$ space. Although Algorithm 2 works on disconnected graphs, and we use disconnected graphs in our implementation, in the pseudo-code, it is assumed that the input graph is connected for simplicity. The correctness of our algorithm follows from the theorem below.

THEOREM 4.1. *Two vertices are biconnected in G if and only if they are biconnected in $T \cup F_c \cup E_b$.*

5 Third Application: Minimum-weight Spanning Tree

This section studies the Minimum Spanning Tree (MST) problem on a weighted undirected graph. For a graph $G = (V, E, w)$ with $n := |V|$ vertices, $m := |E|$ edges, and $w(e)$ as the weight of edge $e \in E$, the MST problem seeks a subgraph G' of G that spans all vertices in V with minimal total weight $\sum_{e \in E(G')} w(e)$ [§].

5.1 PHEM Algorithm for MST

Following Algorithm 1, we describe Algorithm PHEM-MST by providing details of the subroutines GPU-COMPUTE and CPU-COMPUTE. We use E to denote the list of edges and B_i to denote the edges in i^{th} batch in the graph. F_i denotes the minimum spanning forest that the GPU-COMPUTE builds on the GPU on the set of edges in $\cup_i B_i$, and F_{cpu} represents the minimum spanning forest computed using CPU-COMPUTE.

GPU Compute. To process the i^{th} batch of edges, B_i, the GPU COMPUTE routine combines F_{i-1} with B_i to obtain the forest F_i. In other words, we set $B_i = B_i \cup F_{i-1}$. Initially, F_0 is set to $\emptyset$. This iterative process continues until we obtain the MSF F_{GPU} for G_{GPU}.

For obtaining F_i, our GPU-COMPUTE uses ideas from Zhou et al. [44] that use Boruvka's algorithm in edge list representation. We adapt their multicore program to the GPU. To this end, for each vertex u in B_i, we find the vertex $bestNeighbor[u]$ that shares the least weighted edge from u in B_i. We use the least vertex id neighbor to resolve ties. We use the array $R[u]$ to keep track of the representative of each vertex u. The array $R[\cdot]$ is initialized with each vertex being its representative. Next, we add all edges found using $bestNeighbor[\cdot]$ where both u and $R[u]$ have different representatives to the MSF F_i. To resolve double counting, we also add the condition $u < bestNeighbor[u]$. Afte this, we update $R[\cdot]$ using a shortcutting method [35] until $R[R[u]] = u$. We remove the edges in B_i where both endpoints have the same $R[\cdot]$. Removing such edges helps reduce the search space for future iterations. The process repeats by finding $bestNeighbor$ for the remaining set of edges, and we continue until there are no more edges to obtain F_i.

The GPU-COMPUTE introduces additional techniques for faster computation and efficient storage. We represent each edge uv and its weight $w(uv)$ only once and do not include its symmetric counterpart vu in our edge list. This saves space and does not require the graph representation to be symmetric. This technique contrasts most implementations, including [12] and [2]. Our implementation is, therefore, memory efficient at the expense of carrying out another operation to find the best neighbor for a vertex. Further, to reduce the memory accesses while obtaining $bestNeighbor[\cdot]$, we use a relabeling step as discussed in [44] and [7].

We note that for CC and BCC, we use an incremental approach to combine the partial results in successive batches. However, in the case of MST, we chose an iterative approach where we use a Boruvka-style algorithm to combine the edges in F_{i-1} with those in B_i to obtain the MSF F_i. We choose this since typical incremental dynamic algorithms for MSF work best when the number of edges in the update is small. In our setting, the number of edges in the update is of the same order of magnitude as the current MSF. The correctness of such an approach follows from the fact that there are no edge deletions or removals.

CPU Compute. We use a variant of the code in [2] modified to work in the PHEM model, and we offload a portion of the MSF computation. The MSF is found using the parallel Filter Kruskal method. As with the other workloads we study in this paper, we refactor the source code from [2] to work with a subset of the edges concurrently with GPU-COMPUTE.

Final Merge. The above two subsections show that the CPU and the GPU computations produce two MSFs, F_{CPU} and F_{GPU}, respectively. We transfer F_{CPU} to the GPU as part of the final merge step. Subsequently, we combine F_{CPU} and F_{GPU} to produce the MSF F of G.

6 Experimental Results

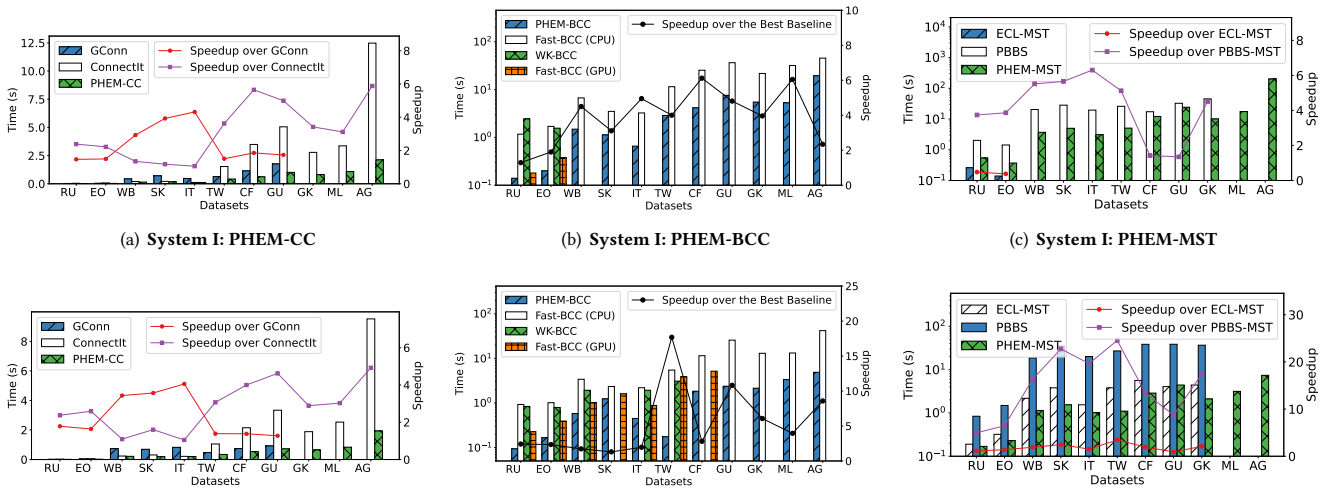
This section presents a detailed evaluation of our proposed PHEM algorithms. We describe the two systems used for our experiments and compare performance results across various datasets and baselines.

6.1 Experimental Platforms

We run our experiments on two heterogeneous systems. The first system, labeled **System I**, has an NVIDIA L4, which is an Ada generation GPU with 24 GB on-board RAM that offers a 300 GB/sec memory bandwidth, 48 MB of L2 cache, and 128 KB of L1 cache per SM with a total of 60 Streaming Multiprocessors (SM). This GPU is attached to a dual socket AMD EPYC 7742 CPU with 64 cores each, 4 MB L1 cache, 32 MB L2 cache, and 256 MB L3 cache. With hyperthreading, the CPU supports 256 threads. The second system, labeled **System II**, has an NVIDIA A100, which is a Volta generation GPU with 40 GB on-board RAM that offers 2.04 TB/s memory bandwidth, 6MB of L2 cache, and 128KB of L1 cache per SM with a total of 80 SMs. This GPU is attached to an AMD EPYC 7742 CPU with 64 cores, 4 MB L1 cache, 32 MB L2 cache, and 256 MB L3 cache. With hyperthreading, the CPU supports 128 threads.

As our datasets have instances where the number of edges crosses a billion, our implementation uses a 64-bit unsigned integer data type to store the ID of an edge. For this reason, we also refactor the source code of implementations we compare by changing the data type for edge ID to a 64-bit unsigned integer. All our programs use CUDA. **PHEM Model Implementation.** We implement our PHEM model using pinned (page-locked) host memory (`cudaMallocHost`) to streamline data transfers between

[§] If the input graph is not connected, we obtain a minimum spanning forest.



(a) **System I: PHEM-CC** (b) **System I: PHEM-BCC** (c) **System I: PHEM-MST**

(d) **System II: PHEM-CC** (e) **System II: PHEM-BCC** (f) **System II: PHEM-MST**

Figure 3: Figures show the run time of Algorithm PHEM-CC, PHEM-BCC, and PHEM-MST on System I and System II along with speedup of these algorithms over state-of-the-art algorithms on instances from Table 2. The speedup line, anchored to the secondary Y-axis, shows the speedup of the PHEM algorithms. We do not show the run time and speedup numbers for instances that do not fit in the GPU memory. The run time on instances EO and RU is minimal owing to their small size.

the CPU and GPU. Using two CUDA streams, we enable concurrent data transfer and kernel execution. Specifically, two buffers are maintained: one actively involved in computation and the other dedicated to data transfers, thereby overlapping computation and communication.

6.2 Datasets

Dataset	$\|V\|$	$\|E\|$	$\#CC$	$\#BCC$	Acronym
Road Networks					
road_usa	23.94M	57.7M	1	7.3M	RU
europe_osm	50.9M	108.1M	1	43.4M	EO
Web Crawls					
Webbase-2001	118.14M	1.01B	744	30.4M	WB
SK-2005	50.64M	3.62B	126	4M	SK
IT-2004	41.29M	2.05B	979	4.9M	IT
Social Networks					
Twitter7	41.65M	2.41B	1	1.9M	TW
com-Friendster	65.61M	3.61B	1	14M	CF
Synthetic Graphs					
GAP-URAND	134.22M	4.29B	1	1	GU
GAP-Kron	134.22M	4.37B	71.1M	87.5M	GK
Biomedical Hypothesis Generation Systems					
Moliere-2016	30.24M	6.68B	25026	1.4M	ML
Agatha-2015	183.96M	11.6B	13	18.4M	AG

Table 2: List of graphs used in our experiments.

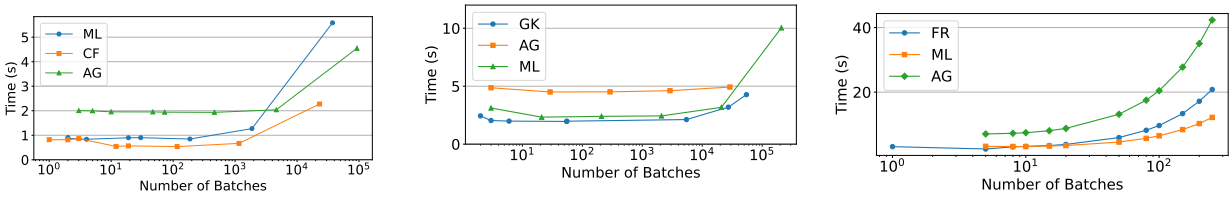
We evaluated our algorithms on eleven graph datasets sourced from DIMACS [30], the Galois framework [25], SNAP [19], and the SuiteSparse Matrix Collection [8]. We removed self-loops and duplicate edges when necessary and assigned random weights to unweighted graphs for MST computation. Table 2 presents the characteristics of each dataset, including the number of vertices ($|\mathbf{V}|$), edges ($|\mathbf{E}|$), and number of CCs (**#CC**) and BCCs (**#BCC**). Our dataset includes all the large graphs from these collections, i.e., graphs with at least 1 Billion edges. These graphs cover the different graph classes mentioned in Table 2.

6.3 Performance of PHEM Algorithms

This section analyzes the performance of the three PHEM algorithms across **System I** and **System II**. We repeat each experiment multiple times, and we report the median runtime. We execute the three PHEM algorithms using the optimal values of α and batch size, determined through experimentation.

6.3.1 Out-of-Memory. We note that the state-of-the-art algorithms, [11, 15, 16], require space not only to store the input graph but also to hold auxiliary data and output. For the three problems studied in this paper, these algorithms use $O(n+m)$ space. However, a closer look indicates that each vertex needs at least 8 bytes, and each edge needs at least 16 bytes to represent. Additionally, these algorithms typically allocate auxiliary space equivalent to 4–5 arrays of size proportional to the number of vertices. These observations explain that some existing in-memory GPU algorithms we refer to in this work fail to execute on certain larger instances.

6.3.2 Overall Performance: In Figs. 3(a) to 3(f) We compare our PHEM-based algorithms with the state-of-the-art approaches in GPU and multicore CPU. We compare our PHEM-CC for connected components against GConn [15] (GPU) and ConnectIt[10] (CPU). For biconnected components, we compare our PHEM-BCC against FAST-BCC [11] (CPU), FAST-BCC-GPU, and WK-BCC [39] (GPU)



(a) **Connected Components** (b) **Biconnected Components** (c) **Minimum Spanning Tree**

Figure 4: Figure shows how the run time of PHEM algorithms on System II for CC and BCC and on System I for MST varies over the number of batches on different instances.

and for Minimum Spanning tree we compare our PHEM-MST against PBBS [2] (CPU) and ECL-MST [12] (GPU). We use the best possible α and batch size to run the PHEM algorithms. Missing bars indicate instances that did not fit in the GPU memory of the given system for the given algorithm.

For connected components, PHEM-CC achieves mean speedups of 2.4× over GConn and 3.17× over ConnectIt on System I, and 2.32× over GConn and 2.84× over ConnectIt on System II. In some instances, PHEM-CC is transfer-bound, so the possible speedup attained on the GPU is bounded by 2.0. On the other hand, in instances that are not transfer bound, Algorithm PHEM-CC achieves a higher speedup of up to 5.16x, including the advantage of heterogeneous execution.

For biconnected components, we observe a mean speedup of 4.9x over FAST-BCC-CPU, 1.6× over FAST-BCC-GPU, and 12.57× over WK-BCC on **System I**. On **System II**, We observe a mean speedup of 8.2× over FAST-BCC-CPU, 4.47× over FAST-BCC-GPU, and 7.8× over WK-BCC. Speedup is computed only for instances that the baseline algorithms (FAST-BCC-GPU & WK-BCC) fit on the GPU.

For minimum spanning tree, we observe a mean speedup of 1.8× over ECL-MST, 14.9× over PBBS on **System II** and a mean speedup of 4.16× over PBBS and a slowdown of 0.43× over ECL-MST (which runs only for the two smallest graphs in 2) on **System I**.

In our experiments, we observe that the performance of PHEM-BCC algorithm is 24% better on average than GPU alone implementation. Similarly, PHEM-CC and PHEM-MST perform 30% and 10% better on average than GPU alone implementation, respectively. The primary reason for this is that both multi-core CPU and GPU are being used effectively and simultaneously.

6.3.3 Variation with Batch Size. In this experiment, we study how the performance of PHEM algorithms varies as we vary the batch size. Recall that the batch size refers to the number of edges transferred from the CPU to the GPU in each iteration. For each of the PHEM algorithms, we consider various datasets from Table 2. At the best possible α for these datasets, we vary the batch size and measure the time the PHEM algorithms require on **System II**.

Fig. 4(a)–(c) show the run time of Algorithm PHEM-CC, PHEM-BCC, and PHEM-MST on **System II** on various instances as we vary the batch size. We observe from the figure that when the number of batches is too large, the overhead involved in multiple kernel calls and merge operations hampers the performance of PHEM algorithms. Suppose the number of batches is too low, indicating a large batch size. In that case, we see a slight reduction in the performance of PHEM algorithms, and this could be due to longer computation time for the last batch B_k of E_{gpu}, as there is no transfer to overlap this computation. Therefore, as Fig. 4 shows, PHEM algorithms achieve their best performance when the choice of batch size mitigates the above two factors.

6.4 Comparison with Other Frameworks

We now compare the performance of our algorithms with some of the other frameworks, such as EMOGI [23] and Subway [31].

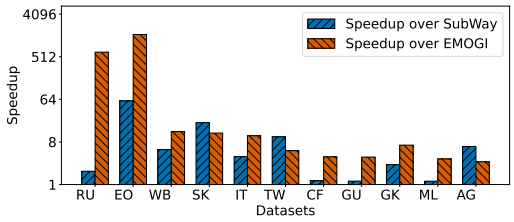


Figure 5: PHEM-CC outperforms EMOGI and Subway

EMOGI [23] handles graphs that do not fit in the GPU memory, but it does not utilize heterogeneity. Further, EMOGI relies on system-managed memory access mechanisms between the GPU and the CPU. From Figure 5, we note that as PHEM uses user-managed memory access mechanisms, Algorithm PHEM-CC outperforms the corresponding algorithm using the EMOGI framework. The median speedup of Algorithm PHEM-CC over the corresponding EMOGI algorithm is 6x. For the RU and EO road networks, EMOGI performs 640x and 1500x slower, respectively, owing to a combination of algorithmic and system-managed memory access inefficiencies.

Subway [31] uses fine-grained subgraph extraction to reduce the number of edges processed along asynchronous subgraph Processing to accelerate graph algorithms for out-of-memory graphs. Our PHEM-CC model performs 4 - 20x faster than Subway. Subway's slower performance is primarily due to the following reasons. The GPU waits for the CPU to copy the active edges from the CPU in each iteration. Also, the number of active edges in each iteration does not fully utilize the GPU memory. We only report comparisons for CC, as existing frameworks such as EMOGI and SUBWAY do not support BCC and MST workloads.

One can think of the HyPar* framework as running the PHEM algorithms with the best possible value of α such that procedure GPU-COMPUTE can run on the entire GPU part of the graph in a single batch. However, we believe the HyPar framework introduces inefficiency as the GPU memory dictates the value of α. On bigger instances, this inefficiency increases.

6.5 Finding Good Values for α and Batch Size

From the three workloads we studied in this paper, we notice that using the appropriate values of α and batch size results in better performance for the PHEM algorithms. Most heterogeneous algorithms, see also [4, 27], and other frameworks such as Liberator [20] require such hyperparameters. However, identifying these parameters a priori is non-trivial as the values of these parameters vary across workloads and instances given a workload. This observation is consistent with other works focusing on algorithms for large graphs, viz. [4, 20, 27]. We believe that estimating these hyperparameters is an orthogonal research problem where some early solutions [14] can be helpful.

7 Conclusion

This paper presented a comprehensive model for utilizing heterogeneity and parallelism while also addressing the needs of algorithms that work with large inputs. The algorithms for the three workloads that this paper studied outperform existing GPU algorithms when the data fits on the GPU and also complete their execution when the data does not fit in the GPU memory.

In the future, we plan to study graph problems that require more than $O(1)$ passes of the input in the PHEM model.

References

[1] N. Agarwal, D. Nellans, M. Stephenson, M. O'Connor, and S. W. Keckler. 2015. Page Placement Strategies for GPUs within Heterogeneous Memory Systems. In *ASPLOS*. 607–618.

[2] D. Anderson, G. E. Blelloch, L. Dhulipala, M. Dobson, and Y. Sun. 2022. The problem-based benchmark suite (PBBS), v2. In *Proc. ACM PPoPP*. 445–447.

[3] N. Ben-David, G. E. Blelloch, J. T. Fineman, P. B. Gibbons, Y. Gu, C. McGuffey, and J. Shun. 2018. Implicit Decomposition for Write-Efficient Connectivity Algorithms. In *IPDPS*. 711–722.

[4] A. Bhowmik and S. Vadhiyar. 2019. HyDetect: A Hybrid CPU-GPU Algorithm for Community Detection. In *HiPC*. 2–11.

[5] G. E. Blelloch, L. Dhulipala, P. B. Gibbons, Y. Gu, C. McGuffey, and J. Shun. 2021. The Read-Only Semi-External Model. In *APOCS*. SIAM, 70–84.

[6] G. E. Blelloch, J. T. Fineman, Y. Gu, and Y. Sun. 2020. Optimal parallel algorithms in the binary-forking model. In *SPAA*. 89–102.

[7] G. Cong and I. Tanase. 2016. Composable Locality Optimizations for Accelerating Parallel Forest Computations. In *HPCC*. 190–197.

[8] T. A. Davis and Y. Hu. 2011. The university of Florida sparse matrix collection. *ACM Trans. Math. Softw.* 38, 1, Article 1 (Dec. 2011).

[9] L. Dhulipala, G. E. Blelloch, and J. Shun. 2021. Theoretically Efficient Parallel Graph Algorithms Can Be Fast and Scalable. *ACM Trans. Parallel Comput.* 8, 1, Article 4 (April 2021), 70 pages.

[10] L. Dhulipala, C. Hong, and J. Shun. 2020. ConnectIt: a framework for static and incremental parallel graph connectivity algorithms. *Proc. VLDB Endow.* 14, 4 (Dec. 2020), 653–667.

[11] X. Dong, L. Wang, Y. Gu, and Y. Sun. 2023. Provably Fast and Space-Efficient Parallel Biconnectivity. In *PPoPP*, Maryam Mehri Dehnavi, Milind Kulkarni, and Sriram Krishnamoorthy (Eds.). 52–65.

[12] A. Fallin, A. Gonzalez, J. Seo, and M. Burtscher. 2023. A High-Performance MST Implementation for GPUs. In *Proc. ACM SC*. 1–13.

[13] J. E. Gonzalez, Y. Low, H. Gu, D. Bickson, and C. Guestrin. 2012. PowerGraph: Distributed Graph-Parallel computation on natural graphs. In *Proc. USENIX OSDI*. 17–30.

[14] M. Hardhik, D. S. Banerjee, K. R. Ramamoorthy, K. Kothapalli, and K. Srinathan. 2017. Nearly Balanced Work Partitioning for Heterogeneous Algorithms. In *ICPP*. 50–59.

[15] C. Hong, L. Dhulipala, and J. Shun. 2020. Exploring the Design Space of Static and Incremental Graph Connectivity Algorithms on GPUs. In *PACT*, Vivek Sarkar and Hyesoon Kim (Eds.). ACM, 55–69.

[16] J. Jaiganesh and M. Burtscher. 2018. A high-performance connected components implementation for GPUs. In *HPDC*. 92–104.

[17] A. Kyrola, G. E. Blelloch, and C. Guestrin. 2012. GraphChi: Large-Scale Graph Computation on Just a PC. In *USENIX*. 31–46.

[18] A. Kyrola, J. Shun, and G. E. Blelloch. 2014. Beyond synchronous: new techniques for external-memory graph connectivity and minimum spanning forest. In *Proc. Symp. Exp. Alg.* Springer, 123–137.

[19] J. Leskovec and A. Krevl. 2014. SNAP Datasets: Stanford Large Network Dataset Collection. http://snap.stanford.edu/data.

[20] S. Li, R. Tang, J. Zhu, Z. Zhao, X. Gong, W. Wang, J. Zhang, and P.C. Yew. 2023. Liberator: A Data Reuse Framework for Out-of-Memory Graph Computing on GPUs. *IEEE TPDS* 34, 6 (2023), 1954–1967.

[21] G. Malewicz, M. H. Austern, A. Bik, J. C. Dehnert, I. Horn, N. Leiser, and G. Czajkowski. 2010. Pregel: a system for large-scale graph processing. In *Proc. ACM SIGMOD Int. Conf. on Management of Data*. 135–146.

[22] K. Meng, J. Li, G. Tan, and N. Sun. 2019. A pattern based algorithmic autotuner for graph processing on GPUs. In *PPoPP*. 201–213.

[23] S. Min, V. S. Mailthody, Z. Qureshi, J. Xiong, E. Ebrahimi, and W-M. Hwu. 2020. EMOGI: Efficient Memory-access for Out-of-memory Graph-traversal In GPUs. *Proc. VLDB Endow.* 14, 2 (2020), 114–127.

[24] J. Nesetril, E. Milkova, and H. Nesetrilova. 2001. Otakar Boruvka on minimum spanning tree problem: translation of both the 1926 papers, comments, history. *Discrete Mathematics* 233, 1-3 (2001), 3–36.

[25] D. Nguyen, A. Lenharth, and K. Pingali. 2013. A lightweight infrastructure for graph analytics. In *SOSP*. 456–471.

[26] NVIDIA Corporation. 2024. CUB: DeviceRadixSort API. https://nvidia.github.io/cccl/cub/api/structcub_1_1DeviceRadixSort.html Accessed: 2025-02-28.

[27] R. Panja and S. S. Vadhiyar. 2019. HyPar: A divide-and-conquer model for hybrid CPU-GPU graph processing. *JPDC*. 132 (2019), 8–20.

[28] P.Gera, H. Kim, P. Sao, H. Kim, and D. Bader. 2020. Traversing large graphs on GPUs with unified memory. *Proc. VLDB Endow.* 13, 7 (2020), 1119–1133.

[29] A. Polak, A. Siwiec, and M. Stobierski. 2021. Euler meets GPU: practical graph algorithms with theoretical guarantees. In *IPDPS*. 233–244.

[30] R. A. Rossi and N. K. Ahmed. 2015. The Network Data Repository with Interactive Graph Analytics and Visualization. In *AAAI*. https://networkrepository.com

[31] A. Sabet, Z. Zhao, and R. Gupta. 2020. Subway: minimizing data transfer during out-of-GPU-memory graph processing. In *EuroSys*. 1–16.

[32] S. Sahu, A. Mhedhbi, S. Salihoglu, J. Lin, and M. Tamer Özsu. 2020. The ubiquity of large graphs and surprising challenges of graph processing: extended survey. *VLDB J.* 29, 2-3 (2020), 595–618.

[33] S. Sakr, A. Bonifati, and H. Voigt et. al. 2021. The future is big graphs: a community view on graph processing systems. *Commun. ACM* 64, 9 (2021), 62–71.

[34] D. Sengupta, S. L. Song, K. Agarwal, and K. Schwan. 2015. Graphreduce: processing large-scale graphs on accelerator-based systems. In *Proc. ACM SC*. 1–12.

[35] Y. Shiloach and U. Vishkin. 1982. An O(log n) parallel connectivityalgorithm. *J. algorithms* 3 (1982), 57–67.

[36] J. Shun and G. E. Blelloch. 2013. Ligra: a lightweight graph processing framework for shared memory. In *PPoPP*. 135–146.

[37] J. Soman, K. Kothapalli, and PJ. Narayanan. 2010. A fast GPU algorithm for graph connectivity. In *Proc. IEEE IPDPS Workshops*. 1–8.

[38] V. Vineet, P. Harish, S. Patidar, and P. J. Narayanan. 2009. Fast minimum spanning tree for large graphs on the gpu. In *Proc. HPG*. 167–171.

[39] M. Wadwekar and K. Kothapalli. 2017. A fast GPU algorithm for biconnected components. In *IC3*. 1–6.

[40] P. Wang, L. Zhang, C. Li, and M. Guo. 2019. Excavating the Potential of GPU for Accelerating Graph Traversal. In *Proc. IEEE IPDPS*. 221–230.

[41] Y. Wang, A. Davidson, Y. Pan, Y. Wu, A. Riffel, and J. D. Owens. 2016. Gunrock: A high-performance graph processing library on the GPU. In *PPoPP*. 1–12.

[42] C. Yu, S. Royuela, and E. Quiñones. 2024. Enhancing Heterogeneous Computing Through OpenMP and GPU Graph. In *ICPP*. 534–543.

[43] Y. Zhang, A. Azad, and A. Buluç. 2020. Parallel algorithms for finding connected components using linear algebra. *JPDC* 144 (05 2020).

[44] W. Zhou. 2017. *A practical scalable shared-memory parallel algorithm for computing minimum spanning trees.* Ph. D. Dissertation. Karlsruher Institut für Technologie (KIT).

*source code not available for direct comparison

8 Appendix

8.1 Correctness of PHEM-BCC Algorithm

We now proceed to prove the correctness of Algorithm 2 in the PHEM model.

Fact 8.1. *Each edge in a graph belongs to exactly one bcc.*

Lemma 8.2. *Let G_1 and G_2 be subgraphs of G, formed by partitioning the edges of G. Suppose F_1 and F_2 are spanning forests of G_1 and G_2, respectively. Then, a spanning forest of $F_1 \cup F_2$ is a spanning forest of G.*

Lemma 8.3. *Each connected component of F_c is in one biconnected component of $T \cup F_c \cup E_b$.*

Proof. Let C be a connected component of F_c. Note that C is a tree, and each edge in C is a cross edge in $G-T$. We apply induction on the number of edges in C and show that all edges in C belong to the same BCC in $T \cup C \cup E_b$. Base condition is when there is only a single edge in C. By Fact 8.1, that edge is in a biconnected component of $T \cup C \cup E_b$.

An edge $e(u, w)$ exists in C, such that e is adjacent to a leaf node, and all edges in $C - e$ remain connected since C is a tree. Due to the inductive hypothesis, we know that all edges in $C - e$ belong to the same bcc in $T \cup C \cup E_b$. We can find an edge in $C-e$ which is adjacent to e, say $e' = (w, v)$. We know that any edge in F_c is a cross edge, and there is no ancestral relationship between the endpoints of that edge in T. Hence, there are no ancestral relationships between u and w, and v and w in tree T. If there is no ancestral relationship between u and v, then the paths in T from u to $lca(u, v)$ and from v to $lca(u, v)$ plus the edges (u, w) and (v, w) in C form a simple cycle. If there is some ancestral relationship between u and v, then the paths in T between v and u, the edge (v, w) and the edge (u, w) form a simple cycle. In both cases, e' and e are in a cycle. This follows that e' and e belong to a BCC in $T \cup C \cup E_b$. Consequently, all edges in C appear in the same BCC in $T \cup C \cup E_b$.

Due to the observation that adding edges to a graph only merges BCCs and by Fact 8.1, we conclude that all edges of C are in one BCC of $T \cup F_c \cup E_b$. □

Theorem 8.4. *Two vertices are biconnected in G if and only if they are biconnected in $T \cup F_c \cup E_b$.*

Proof. Consider a connected component C formed with edges in E_c. Let T' be a tree in F_c such that T' is a spanning tree of C. Let us consider an edge (x, y) in $C - T'$. By Lemma 8.3, there exists a biconnected component B in $T \cup F_c \cup E_b$ such that $V(T') \subseteq V(B)$. Consequently, x and y belong to $V(B)$ and thus part of a cycle in $T \cup F_c \cup E_b$. Adding the edge $e(x, y)$ to the corresponding biconnected component containing x and y does not alter the graph's biconnectivity. Therefore, the biconnectivity of G (that is $T \cup E_c \cup E_b$), is equivalent to that of $T \cup F_c \cup E_b$. □

Fact 8.5. *Let E_1, E_2 and E_3 be subsets of $E(G) - E(T)$ such that $E_1 = E_2 \cup E_3$. Let $u \in V(G)$ and $f(x, y)$ denote the minimum/maximum of x and y. Suppose:*

$$arr1[u] = f\big\{\{first[u]\} \cup \{first[u'] \mid (u, u') \in E1\}\big\},$$

$$arr2[u] = f\big\{\{first[u]\} \cup \{first[u'] \mid (u, u') \in E2\}\big\},$$

$$arr3[u] = f\big\{\{first[u]\} \cup \{first[u'] \mid (u, u') \in E3\}\big\},$$

then $arr1[u] = f(arr2[u], arr3[u])$.

Despite $w1$ and $w2$ arrays are processed for certain edges independently in CPU and GPU, followed by merging in Algorithm 3, Fact 8.5 ensures that $w1$ and $w2$ tags are computed correctly. Similarly, Lemma 8.2 supports that T and F_c are obtained correctly, even if they are computed in a heterogeneous way. By Theorem 8.4, we know that applying a biconnected components algorithm on $T \cup F_c \cup E_b$ is sufficient to find the biconnected components of G. The steps described in PHEM-BCC algorithm (Algorithm 2) are responsible for extracting $T \cup F_c \cup E_b$ from G, and running FAST-BCC algorithm [11] on $T \cup F_c \cup E_b$, in a heterogeneous way. All of these observations prove that our PHEM-BC algorithm is correct, so we have the following theorem.

Theorem 8.6. *Algorithm 2 correctly identifies the BCCs of G.*

8.2 Cut Vertices, Cut Edges and Edge Labels.

Given a graph G, PHEM-BCC algorithm (Algorithm 2) produces labels ($l[\cdot]$) for non-root vertices and component heads ($componentHead[\cdot]$) of all labels except the label of the root vertex. We now present two characterizations that help to find all cut vertices and all cut edges in the parallel setting in constant time using labels and component head arrays. Let w be an arbitrary vertex in G and r be the root vertex of the spanning tree used in Algorithm 2. w is a cut vertex if and only if one of the following holds true. i) $w \neq r$ and w appears atleast once in $componentHead[\cdot]$ ii) $w \neq r$ and w appears atleast twice in $componentHead[\cdot]$. An edge $(par[u], u)$ is a cut edge iff frequency of label of u in $l[\cdot]$ is exactly one. From vertex labels and component heads, we can obtain a BCC label for an edge (u, v) in G in constant time as follows: If $l[u] = l[v]$, then edge label is $l[u]$; If v is component head of $l[u]$, then edge label is $l[u]$; Otherwise, edge label is $l[v]$.