Efficient Parallel Processing of All-Pairs Shortest Paths on Multicore and GPU Systems

Mohammed H Alghamdi[*], *Member, IEEE,* Ligang He[*], *Member, IEEE,* Shenyuan Ren[*] *Member, IEEE,* and Mohammed Maray *Member, IEEE,*

***Abstract*—Finding the shortest path between any two nodes in a graph, known as the All-Pairs Shortest Paths (APSP), is a fundamental problem in many data analysis problems, such as supply chains in logistics, routing protocols in IoT networks that involve consumer electronics as well as data analysis for social networking apps and Google Maps apps used by the general public on their smartphones. In this work, we present a novel approach to solve the APSP problem on multicore and GPU systems. In our approach, a graph is first pre-processed by partitioning the graph into sub-graphs. Then, each sub-graph is processed in parallel using any existing shortest path algorithm such as the Floyd-Warshall algorithm or Dijkstra's algorithm. Finally, the distance results in individual sub-graphs are aggregated to obtain the distances of APSP for the entire graph. OpenMP and CUDA are used to implement the parallelization on multicore CPUs and GPUs, respectively. We conduct the extensive experiments with both synthetic and real-world graphs on the JADE (Joint Academic Data Science Endeavour) cluster at the University of Oxford, which is part of the Tier-2 high performance computing facilities in the UK. In the experiments, we compared our methods with three existing APSP algorithms in the literature, including n-Dijkstra, ParAPSP and SuperFW. The results show that our methods outperform the existing algorithms, achieving the speedup of up to 8.3x over Dijkstra.**

***Index Terms*—All-Pairs Shortest Path, Graph partition, parallel processing, supply chain process, sheared memory parallelism, GPU programming.**

I. Introduction

DETERMINING the shortest path between two or more nodes in a graph is a common task in solving various data analysis problems [1]. The algorithms used to determine the shortest paths have a multitude of applications, including supply chains in logistics [2], routing protocols in IoT networks that involve consumer electronics, as well as data analysis for social networking apps and Google Maps apps used by the general public on their smartphones [3], [4]. All-pairs shortest path (APSP) is a type of shortest path algorithm. Given a directed or undirected weighted graph $G(V, E, w)$, where V is the set of nodes in the graph, E is the set of weighted edges connecting the nodes, and w is the weight of that edge, the APSP algorithm returns the shortest path between any two nodes $V \in G$, where the shortest path is defined as the minimum sum of edge weights on the path that connects two nodes in the graph. There are various algorithms for examining the all-pairs shortest paths, such as Johnson's algorithm [5] and Floyd-Warshall algorithm [6]. Another well-known algorithm is Dijkstra's algorithm [7], which is initially used to solve the Single Source Shortest Path (SSSP) problem. However, when it runs from all nodes in the graph, it can solve the APSP problem for graphs that do not contain edges with negative weights.

Recently, there has been a noticeable increase in using graphs to model real-world problems, which consequently demands the development of efficient methods for graph processing [8]. The massive computational load needed to solve the APSP problem makes sequential processing an impractical solution. Consequently, attention to parallelizing the APSP process has been rising lately. Multicore computers and GPUs are two mainstream types of parallel processing architectures.

In this work, we present two approaches, called SM-CNA and HybridCNA, to parallelizing the processing of APSP for a weighted graph. SM-CNA is a parallelization approach on shared-memory architectures such as multicore computers while HybridCNA utilizes both multicore CPUs and GPUs to solve the APSP problem in parallel. In both approaches, the graph is partitioned into subgraphs, and all-pairs shortest paths in each subgraph are computed in parallel. Next, the local results in subgraphs are aggregated to obtain the all-pairs shortest paths in the entire graph. In both approaches, the steps of partitioning the graph and finding APSP of the subgraphs are parallelized on a multicore CPU using OpenMP. The difference between SM-CNA and HybridCNA is that in SM-CNA the step of aggregating the local results in individual subgraphs is performed on the multicore CPU too, while the aggregating step in HybridCNA is performed on GPU (implemented in CUDA). In HybridCNA, two parallelization methods, called Hybrid Thread (H-Thread) and Hybrid Block (H-Block), are designed to aggregate the local results, which aim to achieve two goals: i) minimizing the communication between the CPU and the GPU, and ii) taking full advantage of the GPU's power by launching appropriate kernels. Additionally, we extend the parallelization methods to work in a server with multiple GPUs to further accelerate the processing.

The calculation of APSP involves irregular accesses to the graph data, including both vertices and edges. This irregularity

M. Alghamdi, Department of Information and Technology Systems, College of Computer Science and Engineering, University of Jeddah, Jeddah, Saudi Arabia.
E-mail: mhalghamdi@uj.edu.sa
L. He, Department of Computer Science, University of Warwick, Coventry, UK
E-mail: ligang.he@warwick.ac.uk
S. Ren, Beijing Jiaotong University, Beijing, China, and Visiting Scientist, Clarendon Laboratory, Department of Physics, University of Oxford, UK
E-mail: syren@bjtu.edu.cn
M. Maray, College of Computer Science and Information Systems, King Khalid University, Abha, Saudi Arabia
E-mail: mmarey@kku.edu.sa
[*] Corresponding author

limits the degree of parallelism achievable during APSP calculation, resulting in underutilization of parallel computing resources. To tackle this challenge, this paper introduces a novel graph partitioning strategy that significantly enhances the parallelism of APSP calculations. Moreover, the proposed strategies craftly exploit common vertices between graph partitions to combine local APSP results. This combination approach is especially designed for high parallelism. By incorporating the graph partition strategy, parallel APSP computation, and the parallel combination of local APSP results, we achieve a high degree of parallel processing throughout the APSP computation steps. Consequently, our approaches outperform existing APSP algorithms in the literature. We conducted the extensive experiments to evaluate the performance of our methods with both synthetic graphs of different sizes and real-world graphs such as social networks and road networks. We compared our methods with three existing APSP algorithms in the literature, including n-Dijkstra [9], ParAPSP [10] and SuperFW [11]. The experimental results show that the parallelization methods developed in this paper can outperform the existing APSP algorithms in the literature, achieving the speedup of up to 8.3x over Dijkstra.

The contributions of this paper are summarized below.

- A novel graph partitioning strategy is proposed to enable higher degree of parallelism in APSP calculations.
- A parallelization algorithm called SM-CNA is proposed to compute APSP on share-memory computing architectures, such as multi-core computers.
- The HybridCNA algorithm is further proposed to accelerate the SM-CNA algorithm by parallelizing the combination of local APSP results on GPU.
- We implement SM-CNA and HybridCNA using OpenMP and CUDA, and evaluate their performance on the JADE platform at Oxford, which is part of the Tier-2 high performance computing facilities in the UK. The experimental results show that the parallelization approaches developed in this paper can outperform the existing APSP algorithms, achieving the speedup of up to 8.3x over Dijkstra.

The rest of this paper is organized as follows. Related work is discussed in Section II. The SM-CNA algorithm for shared-memory architectures is presented in Section III. The HybridCNA algorithm for GPU is presented in Section IV. The proposed parallelization approaches are evaluated in Section V. Finally, this paper is concluded in Section VI.

II. Related Work

In graph theory, determining shortest paths is termed as the basic operation. The primary challenge in solving the APSP problem is that it needs a considerable amount of computation to determine APSP [12]. Therefore, with GPU devices being improved over time, the mechanisms for parallelizing the APSP on GPU have been developed [13]. Okuyama et al. [14] show that GPU can be used to accelerate the solving of APSP. They presented an algorithm with the capability of solving the APSP problem on the CUDA-enabled GPU. The scheme, which was based on the SSSP-based algorithm, was able to solve multiple SSSP problems in parallel because of efficient usage of the on-chip shared memory. The algorithm allowed stream processors (SPs) to concurrently access similar data since every SP participates in solving one of the tasks. Notably, this kind of access, which is common, reduces the access of data to the off-chip memory.

Another work that used multi-SSSP to solve the APSP on GPU is proposed in [15]. It presents a multi-search abstraction as a method for expressing the algorithms that execute the BFS algorithm simultaneously. This research involves an efficient implementation of the abstraction, which is demonstrated to outperform the existing GPU methods implicitly for large graphs of various diameters by more than a factor of two. The authors further demonstrated that a single GPU can be used to solve the APSP problem on sparse graphs with millions of nodes. Their BFS algorithm works only on unweighted sparse graphs, and solves the APSP problem with the complexity of $O(mn)$ (where m is the number of edges and n is the number of nodes).

A work that used the Floyd-Warshall algorithm is described in [16], which presented a blocked algorithm to solve the APSP problem on a hybrid CPU-GPU system. They proposed a united algorithm that can solve the APSP problem for a graph whose size is greater than the capacity of the GPU memory. The total number of operations for this algorithm is $2n^3$ (n is the number of nodes). Their algorithm achieved the peak performance when the number of vertices in the graph is larger than a few thousand.

The work in [17] implemented a GPU version of the adjacency matrix (ADJ-APSP) and the breadth-first search (BFS-APSP). In addition, they developed two versions of their method. The first version is implemented by OpenMP, while the second one is a hybrid implementation by OpenMP and MPI. They conducted the experiments on an unweighted graph only. The results showed that parallelizing ADJ-APSP on a single GPU improved the performance by up to 16.53 times over the single-CPU implementation. On the other hand, parallelization over multiple GPUs achieved even more performance improvement, with the recorded speedup of up to 101.10 times over the single-GPU implementation.

On the investigation concerning a parallel implementation of Johnson's algorithm, reference [18] developed the approach which has the capability of solving the APSP problem based on the current or recent GPU architecture. The objective of the new approach was formulated to increase the speed of APSP computation for large graphs in relation to the CPU. Since GPU can provide high computational cost at a minimal cost, it has been utilized as a substantial alternative. Additionally, to enhance the execution of operations on GPU, the operations are programmed using the framework such as CUDA. The study experimented on three parallel implementations of Johnson's APSP algorithm on the GPU. It further compared the three implementations based on their execution times to determine their advantages and drawbacks.

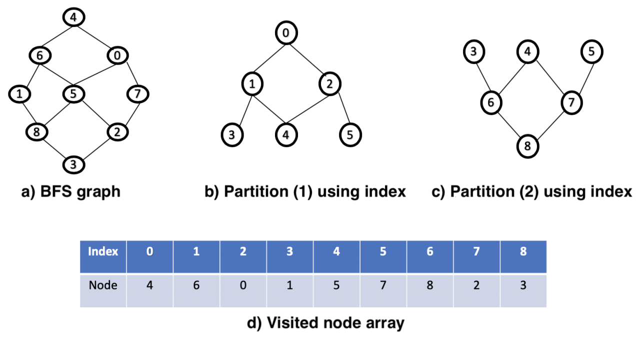
SuperFW is a recently proposed parallel APSP algorithm [11]. Its main objective is to enhance the well-established Floyd-Warshall algorithm by exploiting the algebraic relationship between Floyd-Warshall and Gaussian elimination. SuperFW incorporates several optimization techniques to reduce computaiton, improve locality and enhance parallelism.

This paper is part of the comprehensive research presented in Alghamdi's PhD thesis [19], where we aim to solve the APSP problem in parallel on different types of parallel architectures.

III. SM-CNA

Given a directed weighted graph $G(V, E, W)$, where V is the set of nodes in the graph, E is the set of weighted edges connecting the nodes, and W is the set of weights of the edges. In the case of an unweighted graph, we assume that all edges have equal weights. The shortest path between two nodes, v_i and v_j, is denoted by $S(v_i, v_j)$, while $T(v_i, v_j)$ denotes the temporary (intermediate) value of the shortest path between v_i and v_j. The weights of the shortest path between v_i and v_j is represented by $W(v_i, v_j)$. Our method can also work on undirected graphs. When dealing with undirected graphs, we can treat them as directed graphs where there are two directed edges between any pair of connected nodes with the same weight.

In this work, we solve the All-Pairs Shortest-Path (APSP) problem for G by partitioning G and processing each partition in parallel. P denotes the number of partitions (subgraphs) after the partition of the graph G. In this section, we present the parallelization approach on a multicore CPU, which is called SM-CNA (Shared-Memory-based parallelization through Common Nodes Algorithm). There are three stages in SM-CNA: i) partitioning the graph, ii) finding the APSP in each subgraph, iii) aggregating the results of the shortest paths in individual subgraphs, which are presented next.



Index	0	1	2	3	4	5	6	7	8
Node	4	6	0	1	5	7	8	2	3

Fig. 1: Rearranging the vertices in each subgraph using the index

A. Partitioning the Graph

We partition the graph in this stage. We first run the BFS (Breath-First Search) algorithm on the graph and transform the graph into a multi-leveled graph. The root node of the graph (the node from which BFS starts the search) has level 0. When BFS visits any of the successor nodes of the root, it labels it with level 1. When BFS is completed, every graph node is labelled with a level, and the graph is effectively transformed into a multi-levelled graph. Then, the graph is partitioned across the nodes at some particular levels. If the graph is partitioned at a level, the level is called the boundary level of the two neighbouring partitions. The nodes on the boundary level are duplicated in both neighbouring partitions. In general, after graph partitioning, each partition P_i contains a number of levels and assigned to a CPU core. Since we duplicate the boundary level, the last level in P_i is the first level in P_{i+1}. We call the nodes in the duplicated level the Common Nodes (CN). The set of common nodes duplicated in the neighbouring partitions P_i and P_{i+1} is denoted by C_i.

After transforming the graph into a multi-levelled graph, a node in level i only has the direct connection to a node in level i+1. For the nodes beyond level i+1 (e.g., level j, j >i+1), the node in level i needs to pass through at least one node in levels i+1, i+2, ...,j-1. For example, a node in level 0 needs to pass through at least one node in level 1 and at least one in level 2 to reach a node in level 3. This indicates that the shortest path between two nodes in different levels is the path with the minimum cost crossing the levels. The relation can be modelled in the following equation 1, where P is the number of partitions.

$$(W(v_{L_i}, v_{L_{i+2}}) = Min\{W(v_{L_i}, v_{L_{i+1}}) + W(v_{L_{i+1}}, v_{L_{i+2}})) \\ |1 \le i \le P-1\} \tag{1}$$

We use this relation to choose boundary levels and duplicate the common nodes between two neighbouring partitions. The common nodes can be regarded as the interface between two neighbouring partitions. Take the graph in Figure 1 as an example. In the graph, the nodes in level 2 are duplicated. If node 1 in partition P_1 wants to reach node 8 in partition P_2, then the path contains at least one node from level 2.

Graph's name	The number of nodes in a level	The number of nodes that need to be duplicated
Western US Power Grid	258	183
out.ego-facebook	471	5
road-minnesota	56	40
SW-Trial	77	42
rt_israel.	732	200

TABLE I: The number of nodes actually duplicated on exemplar real-world networks

Duplicating a node does not mean that we have to process it twice because the edges of the node are not doubled. The only edges that are duplicated are those connecting the common nodes themselves. Suppose that s is a node in the duplicated level i. It has five edges with two connecting to the nodes in level $i-1$ and three to level $i+1$. When we examine s, two edges will be processed in first subgraph and three in the second subgraph. The total number is five, same as that when we process the entire graph without partitioning. In general, after graph partitioning, the total number of operations for processing a graph is the same, but the nodes and edges in each subgraph are processed in parallel. If a node on the boundary level is not connected to the next level, this node is not be duplicated. When a graph is relatively sparse, there are usually a limited number of nodes that are connected to the next level. Table I list the examples in real_world networks.

The graph partitioning stage also involves reordering the nodes in the partitions by renaming the nodes with the order in which BFS visits the nodes, which we call the indices of the nodes. The idea is to assign the nodes to the partitions in the ascending order of index. Taking the graph in figure 1 as the example, we have 9 nodes named 0, 1, 2, ..., 8. If the graph is partitioned without reordering the nodes, nodes (4,6,0,1,5,7) are assigned to P_1 while nodes (1,5,7,8,2,3) assigned to P_2. After reordering, P_1 contains nodes (0,1,2,3,4 and 5), while P_2 contains nodes (3,4,5,6,7 and 8). There is no additional cost incurred by reordering since it is conducted while BFS runs. Besides, the information about the partitions is stored in an array. The size of this array is equal to the total number of partitions plus one, i.e., $P + 1$. Element i of the array holds two values regarding partition i: the first value is the reordered index of the first node in partition i while the second value is the total number of common nodes between partitions i and $i + 1$. For the last element of the array, the first value is effectively the size of the entire graph while the second value is set to zero. Algorithm 1 shows the reordering process.

The reordering method can improve the performance significantly. To understand the benefits of reordering, let us take a look at how to find the shortest paths between the nodes in two neighbouring partitions (to be performed in the aggregating stage). To calculate these paths, three pieces of information are needed in our method: the node in one partition, the node in the other partition and the common nodes between the two partitions. The straightforward way is to store the nodes of each partition in a separate array, and store the common nodes between each pair of neighbouring partitions in another array. This means that $P + P - 1$ arrays are needed to store the information, which makes it inefficient to loop through the information. For example, in order to pick a source node from a partition, we need to loop through the common nodes to make sure the source node is not one of the common nodes. It is similar for picking the destination node from another partition. After reordering, the node indices are now in the ascending order. We can use only one array (denoted by Ar[*]) to store and derive the information. For example, if we need to find a node in partition P_i, then its index is in the range of $(Ar[P_i].firstnode, Ar[P_{i+1}].firstnode)$. The indices of the common nodes between two neighbouring partitions P_i and P_{i+1} are in range $(Ar[P_{i+1}].firstnode, Ar[P_{i+1}].firstnode + Ar[P_i].commonnode))$.

The time complexity of graph partitioning is $O(|V| + |E|)$. Graph partitioning can be parallelized by applying the existing methods of running the BFS algorithm in parallel [20] [21] [22] [23]. When applying an existing parallel BFS algorithm, we can assign level labels to the nodes in the same way as in sequential BFS algorithm. For example, suppose the root node is connected to two nodes, v_i and v_j, both labelled as level 1. If the BFS algorithm is executed in parallel, with node v_i being processed by one thread and node v_j by another, all the nodes connected to v_i and v_j will be labelled as 2 (the levels of v_i and v_j plus one). This outcome is equivalent to running the BFS algorithm sequentially. As for duplication of the common nodes, it only occurs after the BFS traversal is complete. Therefore, if the BFS is run in parallel, the duplication step remains unaffected and does not present any issues.

In order to achieve the node balance among partitions, we set a counter when running the BFS algorithm. When BFS visits a node and the counter is equal to N/P, where N is the number of nodes in graph G and P is the number of subgraphs we decide to partition the graph G into, the level that contains that node is selected as the boundary level.

It is also worth noting that we conducted several experiments to determine the optimal choice for the root node when transforming the graph into a multi-level form. These experiments involved selecting the nodes with the highest degree, average degree, lowest degree, or even a leaf node as the root. Surprisingly, we found that the choice of the root node has minimal impact on performance. Therefore, the root node can be chosen randomly. The reason behind this phenomenon is that even if many nodes appear in one level, which becomes the boundary level, any node on the boundary level that is not connected to the next level will not be duplicated. This means that when the graph is relatively sparse, there tends to be only a limited number of nodes that are connected to the next level. Furthermore, our experiments focused on real-world networks, and we found that in relatively sparse graphs, there tends to be only a limited number of nodes that have connections to the next level (as shown in Table 1). This further supports the conclusion that the choice of the root node has minimal impact on the overall performance.

Algorithm 1 Partitioning the Graph

```
Input: $BFS(G, s)$, where $G$ is the weighted graph, $s$ is the
       source node to start the BFS from.
Output: Indexing Array
Let $q$ be a Queue;
 Index[$V_i$] array with size equal to the total number of vertexes
 in $G$
 count = 0;
 Add label to $s$ = count;
 Index.insert(s,count);
 q.enqueue($s$)
 while $q$ is not empty do
     n = q.dequeue;
     for all neighbours $w$ of $n$ in Graph G do
         count = n.label;
         if $w$ is not labled then
             lable $w$ with count+1;
              Index.insert(w,count);
              q.enqueue($w$)
```

B. Calculating the APSP in Each Subgraph

In this stage, the APSP in each subgraph is calculated in parallel. Before this stage starts, a 2-Dimensional (2D) matrix, which is called the *distance matrix* is created to store the distance of the shortest path between the nodes. An advantage of using a matrix is that searching and updating the matrix are efficient with the time complexity of $O(1)$. The matrix size is $N * N$, where N is the total number of nodes in the graph. A row or a column corresponds to a node, while an element $[i, j]$ in the matrix holds the weight of the edge connecting node i and j. When the stage starts, the elements in the diagonal of the matrix are set to 0 (representing the distance from a node to itself), while other elements are set to infinity (∞) (representing the temporary distance between a node and another node). Since SM-CNA is for the shared memory architecture, each CPU core can access the matrix and update the values. At the end of our method, the matrix will be completed with the distances of the shortest paths between any two nodes in the graph.

OpenMP is used to implement the processing of a sub-graph on a CPU core (by a thread) in parallel and to synchronize the running of multiple threads. First, every thread calculates the shortest path between the common nodes in its subgraph and update the distance matrix if the calculated value is smaller than that calculated and stored by the other thread (the thread that processes the neighbouring subgraph). After all threads complete the work, the stored values are the weights of the edges connecting the common nodes themselves. Next, we apply the Floyd-Warshall algorithm [6] (or any existing all-pairs shortest path algorithm) and find the APSP in each partition in parallel. After this stage, we can obtain the shortest paths between two nodes in the same partition.

The bigger the graph, the more memory space is needed to store 2D *distance matrix*) proposed above, whose size is n^2. In the aggregating stage, the information required to find the shortest path between the source node s and the destination node t is the distance of the shortest path between s and a common node and that from the common node to t. Other data are not needed. This motivates us to design a more memory-efficient results storing scheme than the 2D *distance matrix*. The new storing scheme requires a 3D matrix, which is called 3D *distance matrix*. In the 3D *distance matrix*, only the weights needed to calculate the shortest distances of other paths are stored. More specifically, the size of this 3D *distance matrix* is $c * n * 2$, where c (rows) is the total number of the common nodes between all partitions, n (columns) is the total number of nodes in the graph, and the third dimension holds two values: i) the distance of the shortest path from the common node c_i to the node n_j (denoted by $W(c_i, n_j)$), and ii) the distance from the node n_j to the common node c_i (i.e., $W(n_j, c_i)$). In the case of undirected graphs, we do not need the third dimension since the two values are the same. Since c (the total number of common nodes after the graph partitioning) is much less than n (the total number of nodes in the graph). The 3D *distance matrix* is much more memory-efficient than the 2D counterpart. Consequently, larger-scale graphs can be processed by our method.

C. Aggregating the Results in Individual Subgraphs

In this stage, we aggregate the local results obtained in two neighbouring partitions and obtain the shortest distance of a path that crosses these two partitions (i.e., the source node is in a partition and the destination node is in the neighbouring partition). The shortest path between $v_i \in P_i$ and $v_j \in P_j$ must passes at least one common node between the two partitions. We make use of this insight to aggregate the local results in two neighbouring partitions. The calculation is presented as follows.

Assume $v_{k_1}, v_{k_2}, ...v_{k_r}$ is the set of common nodes that are connected to both v_i and v_j. Then the distance of the shortest path between v_i and v_j, $W(v_i, v_j)$, can be calculated by Eq. 2.

$$W(v_i, v_j) = Min\{W(v_i, v_{k_m}) + W(v_{k_m}, v_j) | 1 \leq m \leq r\} \tag{2}$$

After Eq. 2 is applied to two neighbouring partitions, the shortest paths between any two nodes in these two partitions are obtained. The two partitions can be merged as one bigger partition. Eq. 2 is then applied to calculate the APSP between this newly merged partition and its neighbouring partition. This process iterates until the result between all graph partitions are aggregated. The number of iterations in the results aggregation is $\sum_{i=2}^{P_i-1} P_i - i$ (where P_i is the total number of partitions). After this stage is completed, the shortest path between any two nodes in the graph is obtained and the APSP problem is solved. The algorithm for aggregating the results in subgraphs is outlined in Algorithm 2.

Note that the aggregation step of our method is executed in parallel to enhance efficiency. The parallelization is achieved by assigning each subgraph to a separate thread. Each thread is responsible for finding the shortest path to all other nodes in the remaining subgraphs. This task is accomplished by performing simple mathematical operations outlined in Equation 2.

In case where the number of subgraphs exceeds the number of available threads, we divide the total number of subgraphs by the number of threads to determine the workload allocation. Consider an example where there are three subgraphs and only two threads. In this case, one of the threads will handle two subgraphs, while the other will be assigned to process a single subgraph. The objective of this approach is to evenly distribute the computational load among the available threads. By dividing the subgraphs evenly, or as evenly as possible, across the threads, we can maximize parallel processing efficiency. This allocation strategy ensures that each thread has a fair share of the workload, even if the number of subgraphs is greater than the number of threads available.

IV. HybridCNA

The HybridCNA approach is proposed to further accelerate the SM-CNA approach presented in section III. The acceleration works by processing the stage of aggregating the subgrahs' results on GPU. In this section, we present the details of HybridCNA. In particular, we describe the steps executed on the CPU and those on the GPU. Moreover, we will present two parallelization strategies for the GPU threads to aggregate subgraphs' results.

Algorithm 2 aggregating the results in subgraphs

```
Input: Ar (the partitioning array withe the size of (P+1)), M
       (the matrix holding the distances of the shortest paths
       between nodes in graph $G$)
parts = total number of partition
Rank = partition rank(0 to parts-1);
for $x$ = Rank +1 to parts do
    for $i = Ar_{rank,0} to Ar_{rank+1,0}$ do
        for $j = Ar_{(x,0)+(x-1,1)} to Ar_{(x+1,0)+(x,1)}$ do
            for $c = Ar_{(x,0)} to Ar_{(x,0)+(x-1,1)}$ do
                if $M_{i,j} > M_{i,c} + M_{c,j}$ then
                    $M_{i,j} = M_{i,c} + M_{c,j}$;
                end
            end
        end
    end
end
```

A. Processing on CPU

The steps that are run on the CPU (host) start with reading the graph and partitioning it into a certain number of subgraphs. Then the 3D distance matrix is created, which is updated with the distances of the shortest paths when subgraphs' results are aggregated. OpenMP is used to generate threads and each thread runs on a CPU core to process a subgraph. All CPU threads run simultaneously to find the shortest path between the nodes in the same subgraph. Each thread updates the distance matrix with the distances of newly found shortest paths. After the stage of finding the APSP in each subgraph is completed, CPU allocates the device memory for the 3D distance matrix on GPU and copy the 3D distance matrix from CPU memory to GPU memory. Unlike the existing GPU-based methods for solving the shortest path problem, our method do not need to copy the graph itself to the GPU, but only copy the distance matrix once, which reduces the communication cost and also allows us to solve the APSP problem for even bigger graphs that cannot be fit in the GPU memory. When the graph is so big that the corresponding 3D matrix is bigger than the GPU memory, we can partition the matrix along the rows and send a partition of the distance matrix to the GPU at a time.

After the 3D distance matrix has been copied to GPU, the execution is shifted to GPU by lauching the kernel function, which aggregates the subgraphs' results and find the shortest paths between the nodes in different subgraphs. When the aggregating stage is completed on GPU, GPU transfers the final distance matrix, which holds the distances of the shortest paths between any two nodes in the entire graph, to the host.

B. Processing on GPU

The threads on the GPU run simultaneously and apply the following equation to aggregate the local results in each subgraph and find the shortest paths between the nodes in different sub-graphs.

$$W(v_i, v_j) = Min\{W(v_i, v_{C_m}) + W(v_{C_m}, v_j) | 1 \leq m \leq r\} \quad (3)$$

The threads find the values $W(v_i, v_{C_m})$ and $W(v_{C_m}, v_j)$ in the distance matrix, which is received from the CPU. Then the threads update the distance matrix by adding the newly calculated distance values ($W(v_i, v_j)$).

We refer to the step of aggregating the results of two subgraphs as a *Subgraph-Combination* (SC) operation. Namely, every two subgraphs takes one SC operation to calculate the shortest paths between their nodes. Assuming a graph is partitioned into P subgraphs, the total number of SC operations can be calculated by the following equation:

$$\sum_{i=1}^{P-1} \textit{(P-i)} \qquad \textit{P is the total number of partitions} \quad (4)$$

Now we use an example to illustrate the SC operations. Suppose that a graph is partitioned into five subgraphs (i.e., partitions), which are indexed from 0 to 4. The SC operations needed to aggregate the results are illustrated in Table II, where "0 to 1" represents aggregating the results of partitions 0 and 1, namely finding the shortest paths between the nodes in partitions 0 and 1.

We have identified two types of parallelism, namely subgraph-level parallelism and node-level parallelism, for aggregating results from multiple subgraphs. In the subgraph-level parallelism, multiple *independent* SC operations are run simultaneously. In Table II, the SC operations in different columns are independent and can be run in parallel. For example, SC operation (0 to 1) and SC operation (1 to 2) in Table II) can be run in parallel. The SC operations in the same column in the table have the dependency and must be run in sequence. For example, the operation (0 to 2) can only start after (0 to 1) has been completed since partition 1 is partition 0's neighbour but partition 2 is not.

In node-level parallelism, the shortest paths starting from different nodes in a subgraph are computed in parallel. For example, assuming s_1 and s_2 are two nodes in partition 0, the process of finding the shortest paths from s_1 to the nodes in partition 1 can be performed in parallel with finding the shortest paths from s_2 to the nodes in partition 1.

This work proposes two parallelization strategies, H-Thread and H-Block, to achieve subgraph-level and node-level parallelism, respectively, in order to fully leverage the potential of GPUs. H-Block assigns a thread block to aggregate the results for a certain number of subgraphs (i.e., aggregate the results between the subgraphs assigned to the block and other subgraphs). Multiple blocks aggregate the results for different subgraphs in parallel, which achieves subgraph-level parallelism. H-Thread assigns a thread in a thread block to aggregate the results for a certain number of nodes in the subgraph. Different threads in a block can then aggregate the results for different nodes in the subgraph in parallel, which achieves node-level parallelism. The rest of subsection presents H-Thread and H-Block strategies in detail.

1) H-Thread and node-level parallelism: n_i denotes the total number of nodes in subgraph i. The H-Thread strategy works by assigning a certain number of nodes in a subgraph (e.g., subgraph i) to a GPU thread. Assume k nodes are assigned to be processed by a thread. When the thread completes

TABLE II: Case study of aggregating the result of five subgraphs

0 to 1 0 to 2 0 to 3 0 to 4	1 to 2 1 to 3 1 to 4	2 to 3 2 to 4	3 to 4

the calculation of the shortest paths from these k nodes and all nodes in subgraph $i+1$, the thread continues to calculate the shortest paths from these k nodes to all nodes in subgraph $i+2$ until the shortest paths from these k nodes to the nodes in all other subgraphs have been calculated.

Consider the example in table II. Assume subgraph 0 is assigned to a thread block. Each thread in the block is assigned to calculate the shortest paths between a number of nodes (e.g., k nodes) in subgraph 0 and all nodes in subgraph 1. After the first SC operation (0 to 1) is completed for the k nodes, the thread moves on run the second SC operation (0 to 2) for the k nodes. The procedure goes on until the last SC operation, (3 to 4), has been completed for the k nodes. Algorithm 3 outlines the steps in the H-thread strategy.

Algorithm 3 The H-Thread Strategy

```
Let $M_r$ be the distance matrix;
Let $P$ be the total number of partitions;
Let $C_{i,j}$ be the list of common nodes between subgraphs $i$ and $j$;
index = blockIdx.x * blockDim.x + threadIdx.x;
x= index;
for $i$ in range($0$ to $P-1$) do
    x= x+ rank of first node in i;
    for $j$ in range($i+1$ to $P-1$) do
        for $y$ in $j$ nodes do
            for $c$ in $C_{i,j}$ do
                if $M_r$[x][y] > $M_r$[x][c] + $M_r$[c][y] then
                    $M_r$[x][y] = $M_r$[x][c] + $M_r$[c][y];
```

The number of nodes assigned to a thread is ($\frac{n_i}{T_i}$,$T_i \leq n_i$), where n_i is the total number of nodes in subgraph i and T_i is the number of threads in the thread block assigned to process subgraph i. The maximum number of threads that can be launched equals to the total number of nodes in the biggest subgraph. The time that a thread takes to complete a SC operation for the nodes assigned to the thread (i.e., aggregating the shortest distances from the thread's $\frac{n_i}{T_i}$ nodes to all the nodes in another subgraph), denoted by $t_{sc}(i)$, can be modelled as follows, where $c_{i,i+1}$ is the number of common nodes that connect subgraphs i and $i+1$.

$$t_{sc}(i) = \frac{n_i}{T_i} \times c_{i,i+1} \times (n_{i+1} - c_{i,i+1}) \tag{5}$$

Thus the total time that a thread takes to complete all the SC operations for the $\frac{n_i}{T_i}$ nodes (denoted by $t_{allsc}(m)$) can be modelled by Eq. 6, where P is the total number of subgraphs and m is the index of the calling thread.

$$t_{allsc}(m) = \sum_{i=0}^{P-1}(\frac{n_i}{T_i} \times \sum_{j=i+1}^{P-1} c_{i,j} \times (n_j - c_{i,j})) \tag{6}$$

Since all threads in a block run in parallel, we can use Eq. 7 to model the computation time that the thread block (assuming it is block k) takes to finish the aggregation stage with the H-Thread strategy, denoted by $t_b(k)$.

$$t_b(k) = \max\{t_{allsc}(m)\} \tag{7}$$

2) H-Block and subgraph-level parallelism: In the H-Block strategy, a number of thread blocks are generated with each containing a number of threads. We schedule different thread blocks to run the SC operations in parallel. For instance, if we schedule four thread blocks to run the SC operations in table II, the SC operations 0 to 1, 1 to 2, 2 to 3 and 3 to 4 will be run by the four thread blocks in parallel, which is subgraph-level parallelism. Moreover, the threads in each block use the H-Thread strategy to perform a SC operation with different nodes in parallel. Therefore, both subgraph- and node-level parallelism are achieved. Algorithm 4 outlines the H-Block strategy.

Algorithm 4 The H-Block Strategy

```
1.20 Let $M_r$ be the result matrix;
Let $P$ be the total number of partitions;
Let $C_{i,j}$ be the list of common nodes between subgraphs $i$ and $j$;
index = threadIdx.x;
$i$ = blockIdx.x;
x= index+ rank of first node in $P_i$;
for $j$ in range($i+1$ to $P-1$) do
    for $y$ in $j$ nodes do
        for $c$ in $C_{i,j}$ do
            if $M_r$[x][y] > $M_r$[x][c] + $M_r$[c][y] then
                $M_r$[x][y] = $M_r$[x][c] + $M_r$[c][y];
```

Based on Eq. 7, the total computation time for a GPU to complete the aggregation stage with the H-Block strategy, denoted by t_{GPU} can be modelled as Eq. 8. We will compare the H-Thread and H-Block approaches in Section V.

$$t_{GPU} = \max\{t_b(k)\} \tag{8}$$

The HybridCNA method can also be deployed on a server with a multicore CPU and multiple GPUs. In a system with multiple GPUs, the CPU schedules the SC operations to run across multiple GPUs. The CPU sends the required data to each GPU and launch the kernels, and then all GPUs process the SC operations in parallel. The distance matrix is copied to all GPUs. The distance matrix will hold the distances of the shortest paths between any nodes in the entire graph after all GPUs complete their work. When launching the kernels, we still use the H-Block and H-Thread strategies to manage the threads in each GPU. Consider the example in Table III. In GPU1 and GPU2, we can use the H-Thread strategy only since

GPU 1	GPU 2	GPU 3
0 to 1		
0 to 2	1 to 2	2 to 3
0 to 3	1 to 3	3 to 4
0 to 4	1 to 4	

TABLE III: An example of processing a graph on three GPUs; the graph is partitioned into five subgraphs

the SC operations depend on each other. On GPU3, however, we can use the H-Block strategy since the two SC operations are independent of each other.

V. Evaluation

In this section, we evaluate the efficiency of the SM-CNA and HybridCNA methods. First, we compare the performance between the serial implementation and parallel implementation. Then, we conduct ablation studies regarding the performance of two strategies, H-Thread and H-Block, with various graph sizes. Next, we compare the HybridCNA method with the SM-CNA. Finally, we evaluate the effectiveness of the HybridCNA method with a real-world graph.

The experiments were run on the Joint Academic Data Science Endeavour (JADE) server at the Oxford University [24]. It consists of eight NVIDIA Tesla V100 GPUs interconnected by NVIDIA's NV link interconnect technology. The specification of CPU is Dual 20-Core Intel Xeon E5-2698 v4 2.2 GHz. The main Memory is 512 GB 2,133 MHz DDR4 RDIMM. Cuda 10.1 and OpenMP 3.0.0 are installed to implement the HybridCNA method.

We conduct the experiments with different graph sizes from 2k to 16k. The graph generator [25] is used to generate the graphs following the power-law distribution, which has been shown to be the property of real-world graphs. To evaluate the impact of the common nodes, we first ran the experiment with a different number of common nodes from 2 to 1000. After that, we set the total number of common nodes to be 200 for the rest of the experiments, which is the highest number of common nodes in the real-world graph used in our experiments.

A. *Speedup of Parallelization*

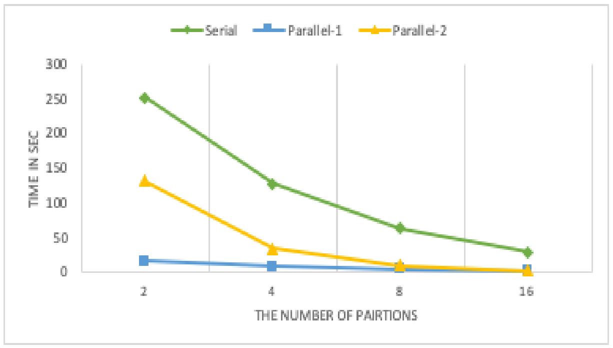


Fig. 2: The performance of our two parallel implementations comparing to the serial implementation; The graph size is 16K

We first run the serial implementation of our SM-CNA method, in which after the graph is partitioned, the shortest paths between the nodes in all the subgraphs are found in sequence and local results in individual subgraphs are also aggregated in sequence. We implement SM-CNA in two different ways as follows.

i) Parallel-1: the graph is partitioned into a fixed number of subgraphs, and then the subgraphs are assigned to a different number of cores for processing;

ii) Parallel-2: the number of subgraphs that the graph is partitioned into is the same as the number of cores in the multicore computer with each subgraph being assigned to a core.

Figure 2 shows the comparison between the serial implementation and the two parallel implementations of SM-CNA. The graph size used in this experiment is 16K. The graph is partitioned into 2, 4, 8 and 16 subgraphs. When the graph is partitioned, the cost of finding the APSP is reduced significantly. This is because the time spent in solving APSP grows exponentially as the graph size increases. Therefore, partitioning the graph into smaller subgraphs reduces the time of each subgraph exponentially. However, the benefit of reducing the time of finding the APSP in each subgraph is partly cancelled by the increased time spent in aggregating the results between subgraphs as the graph is partitioned into more subgraphs. However, the increase in aggregation time is much lower than the decrease in the time of calculating APSP. As the result, we can observe that the more partitions, the faster the algorithm is for solving the APSP problem. Figure 5 shows the speed up of serial SM-CNA as the number of partitions increases.

Parallel-2 in figure 2 is the parallel implementation of SM-CNA with each subgraph being assigned to a CPU core, i.e., all subgraphs are processed simultaneously. It can be seen from figure 5(b) that our SM-CNA delivers the outstanding speedup over the serial SM-CNA.

In parallel-1 implementation of SM-CNA, the number of partitions is fixed to 16, which are then processed on a different number of cores (2, 4, 8 and 16). We can see from figure 2 that parallel-1 is faster than parallel-2 when the number of cores is small. Parallel-2 catches up as the number of partitions increases.

B. *Evaluating H-Thread and H-Block*

In this section, we compare the two strategies, H-Thread and H-Block, proposed to assign the workload to the GPU threads. We ran the experiment on a graph with the size of 16k and partitioned the graph into 2, 4, 8 and 16 subgraphs. The experimental results are shown in Figure 6, where the y-axis is the time taken to aggregate the results between the subgraphs (GPU time) with each approach. We does not count the communication time between the CPU and the GPU (copying the data to the GPU memory) because it is the same in both approaches.

As shown in figure 6, when the number of the partitions is small (two and four), the H-Thread approach achieves better performance than H-Block. However, as the number of partitions increases, H-Block overtakes H-Thread. This is because when a graph is partitioned into a small number of

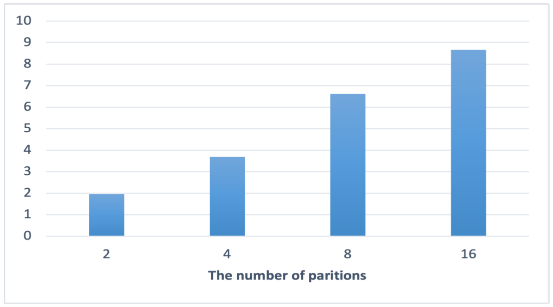


Fig. 3

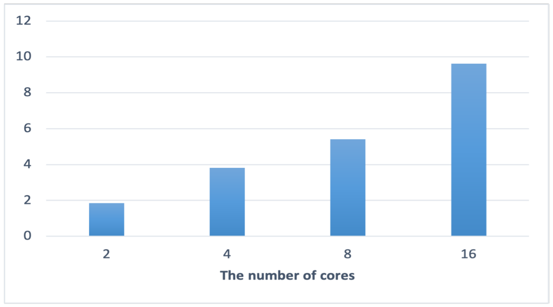


Fig. 4

Fig. 5: Speedup of serial (a) and parallel (b) CNA, Graph size is 16K

	SM-CNA		HybridCNA	
Number of Common node	aggregating Time	Total time	GPU Time	Total time
10	0.296	7.008	0.477	7.136
100	2.196	8.88	0.629	7.346
500	11.109	17.809	1.259	7.92
1000	24.284	30.967	1.814	8.475

TABLE IV: Case study of finding APSP of a graph with the size of 4k and partitioned into two subgraphs.

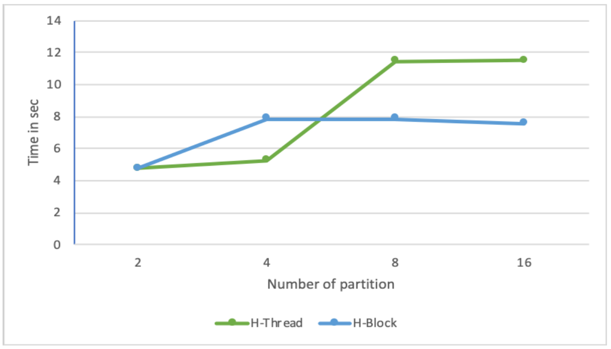


Fig. 6: Comparing the H-Thread strategy with the H-Block strategy. The graph size is 16k

subgraphs, the number of nodes in each subgraph is big. Since in H-Thread the nodes in an operation are parallelized and the operations are handled one at a time, there will be more threads engaged in the computation simultaneously than when the graph is partitioned into a smaller number of subgraphs. Additionally, in H-Block, at least one block completes its task and terminates after each iteration, which means a fewer number of threads run simultaneously after each iteration (i.e, the decrease in the degree of parallelism). The effect is noticeable if the number of threads in each block is large (such as the four partitions in figure 6). However, when the number of partitions increase to 8 and 16, a fewer number of GPU threads are terminated and more threads work simultaneously for longer time (i.e., the increase in the degree of parallelism). This is when H-Block starts performing better than H-Thread. Also, we can notice that unlike H-Thread, the number of partitions does not affect much the performance of the H-Block approach. Since the operations are run in parallel in H-Block, all nodes in the subgraphs are assigned to the GPU threads and run at the same time.

To summarize, we can conclude that both H-Theard and H-Block can achieve excellent performance in different conditions. When we have a large subgraph, it is better to use the H-Thread approach. When we have many small subgraphs, using H-Block is more beneficial.

C. Comparing HybridCNA with SM-CNA

In this subsection, we compare HybridCNA with SM-CNA against three parameters: the number of common nodes, the graph size, and the number of partitions.

As previously discussed, the number of common nodes has a significant impact on the performance of our method, as it affects the number of SC operations required to aggregate local results. To investigate the effect of this parameter, we conducted an experiment on a graph of size 4k partitioned into two subgraphs. We increased the number of common nodes from 10 to 1000 and recorded the time taken for the aggregating step, as well as the total time. The results are shown in Table IV.

The results indicate that for both methods, as the number of common nodes increased, the time taken to aggregate the local results also increased. However, the rate of increase in time is much higher for SM-CNA than for HybridCNA. SM-CNA took 0.2 seconds to aggregate results when there were 10 common nodes and 24.2 seconds when the number of common nodes increased to 1000, which is about 82 times slower. On the other hand, HybridCNA took 0.4 seconds when the number of common nodes was 10 and 1.8 seconds when it increased to 1000, which is only about 3.8 times slower. The reason for this is the significantly higher power of the GPU, which can support the simultaneous execution of many more threads than the number of CPU cores that can be generated in SM-CNA.

Another observation from Table IV is that when the number of common nodes is small (e.g., 10), SM-CNA is faster than HybridCNA. This is because the communication overhead between the host and the device in HybridCNA is higher, as we will discuss in the next subsection.

In the next experiment, we evaluated the performance of HybridCNA and SM-CNA with different graph sizes, with each graph partitioned into up to 16 subgraphs. We recorded the time taken by each method for the aggregating step. The aggregation time of HybridCNA includes the kernel time as well as the time taken to copy data between the CPU and the GPU. For this evaluation, we set the number of common nodes to be 200.

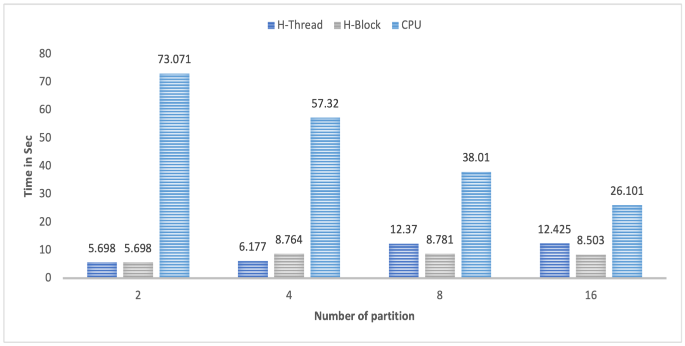


Fig. 7: Comparing HybridCNA with SM-CNA. The graph size is 16k

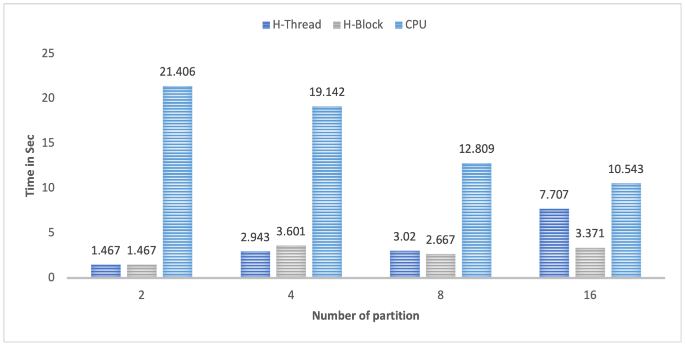


Fig. 8: Comparing HybridCNA with SM-CNA. Graph size is 8k

As shown in Figures 7, 8, 9, and 10, the aggregation time in HybridCNA is consistently lower than that in SM-CNA across all graph sizes. Furthermore, as the graph size increases, the performance gap between the two methods also increases. For example, as shown in Figure 10 for the graph with a size of 2k and partitioned into two subgraphs, HybridCNA is about 3.2 times faster than SM-CNA. For the graph with a size of 16k, HybridCNA is 12.8 times faster, as shown in Figure 7. This is because as the graph size increases, more SC operations need to be performed and the GPU can execute these operations in parallel, which gives it an advantage over SM-CNA.

When we investigated the effect of the number of partitions on the performance of the two methods, we observed that as

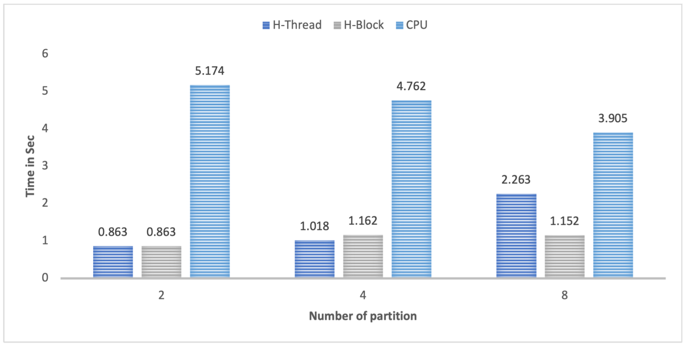


Fig. 9: Comparing HybridCNA with SM-CNA. Graph size is 4k

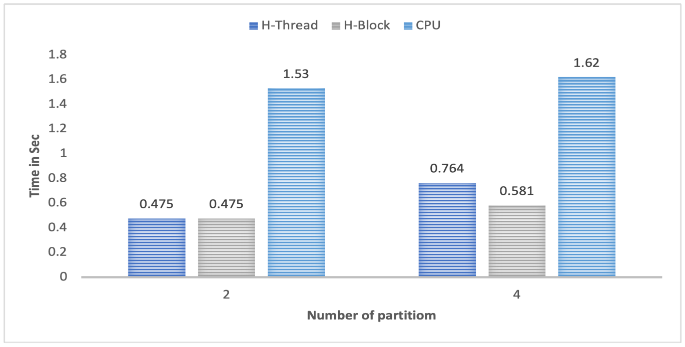


Fig. 10: Comparing HybridCNA with SM-CNA. Graph size is 2k

the number of partitions increases, the time required for SM-CNA decreases while the time for HybridCNA either remains the same or slightly increases. This is because the more subgraphs we partition the graph into, the more CPU cores are needed by SM-CNA to find the shortest paths between the subgraphs, whereas the number of threads used in HybridCNA is not affected by the number of partitions. However, even with the increased time required by SM-CNA, HybridCNA still achieves better performance, with at least a 3x speedup over SM-CNA.

Figure 11 shows the total time achieved by HybridCNA and SM-CNA when running graphs of different sizes. As expected, HybridCNA outperforms SM-CNA in all cases. This is because the aggregation time significantly impacts the total time of the methods, and HybridCNA's faster aggregation time translates into overall faster execution times.

D. *Evaluating HybridCNA on a system with multiple GPUs*

In this section, we evaluate the performance of HybridCNA on a computer equipped with multiple GPUs. We conducted experiments using up to four GPUs and evaluated HybridCNA on three different graph sizes: 4k, 8k, and 16k.

To start, we needed to copy the distance matrix from the CPU to the GPU. The JADE architecture [24] has a memory layer where multiple GPUs can share memory, giving us the option to either copy the matrix once to the shared GPU memory or copy it multiple times to the local memory inside each GPU. Table V shows the difference between

2*Copying way	1GPU			2GPUs			3GPUs		
	copy time	kernel time	Total GPU time	copy time	kernel time	Total GPU time	copy time	kernel time	Total GPU time
one copy	536	2131	2667	536	4887	5423	536	10644	11180
Multi copying	536	2131	2667	631	1639	2270	740	1386	2126

TABLE V: A case study comparing multi-copying with single copying strategy for a graph size of 8K partitioned into 8 subgraphs. The time is measured in milliseconds.

these two copying strategies. As we can see, copying the data once reduces the communication cost between the CPU and the GPU and requires less time than copying the data multiple times. However, we noticed that the kernel time increased significantly when copying the matrix only once. This is because thousands of GPU threads are trying to read and write to the same memory address, causing a racing condition and a dramatic increase in kernel time. On the other hand, when using the multi-copying strategy, the more GPUs engaged in the computation, the more time needed for the copying process. Nonetheless, copying the data multiple times achieved better performance in terms of the total kernel time. Therefore, we will use the multi-copying strategy in the following experiments.

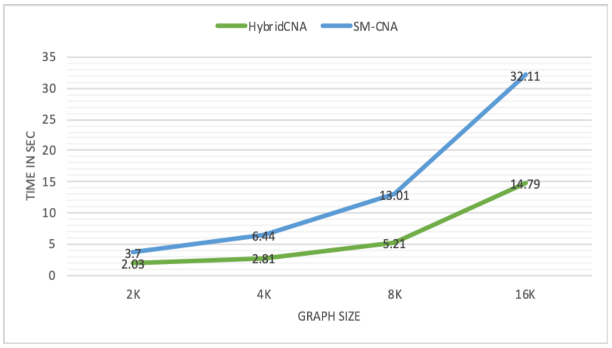


Fig. 11: Total time of HybridCNA and SM-CNA methods in different graph size

Figure 12 and 13 show the total GPU time (data copying time plus the kernel time) of running HybridCNA on graphs with sizes of 8K and 16K. From these figures, we observe that HybridCNA achieves better performance as the number of GPUs increases. Running with two GPUs achieved an excellent speedup (up to 1.6). Similarly, running with 4 GPUs achieved up to 2.5 speedup. This performance is affected by the graph size and the number of partitions. For example, as shown in 12, when the graph is partitioned into 16 subgraphs, running with 4 GPUs took longer than with 3 GPUs when the graph size is 8K. However, when processing the graph with the size of 16K, running with 4 GPUs achieved better performance than with 3 GPUs, as shown in 13.

Figure 14 displays the total time required for processing graphs of varying sizes using multiple GPUs in HybridCNA. The figure reveals that for the 4K graph, processing it with 2 GPUs results in a slightly better performance. Increasing the number of GPUs beyond 2 does not lead to further performance improvement. However, for the 16K graph, using 2 GPUs achieves excellent speedup. Further increasing the number of GPUs to 3 and 4 results in even better performance.

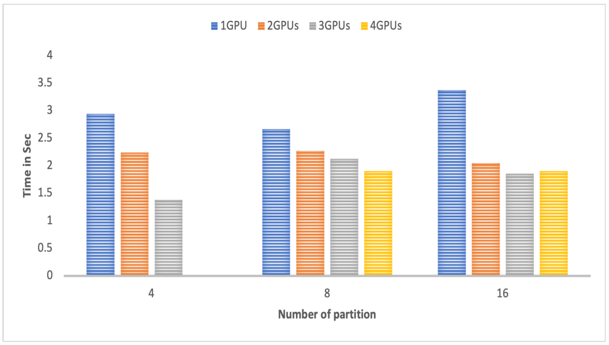


Fig. 12: Comparing the kernel time of running with 1 GPU and running with Multiple GPUs. The graph size is 8k

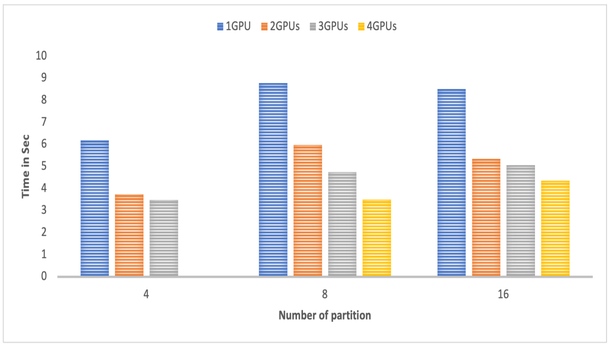


Fig. 13: Comparing the kernel time of running with 1 GPU and running with Multiple GPUs. The graph size is 16k

Therefore, we can conclude that using multiple GPUs is beneficial when the graph size is large.

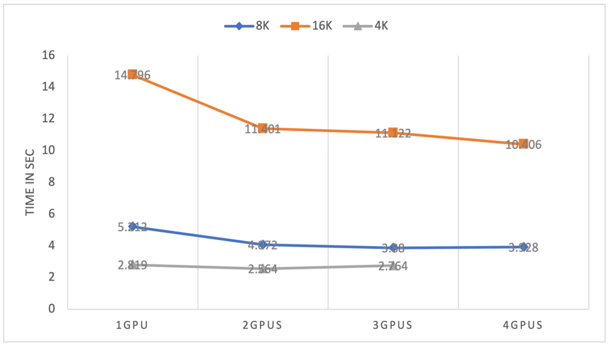


Fig. 14: Comparing the total time between running with 1 GPU and running with Multiple GPUs on different graph sizes

E. Evaluating HybridCNA with Real-world Graphs

To demonstrate the effectiveness of our SM-CNA and HybridCNA, we evaluated them with real-world graphs and

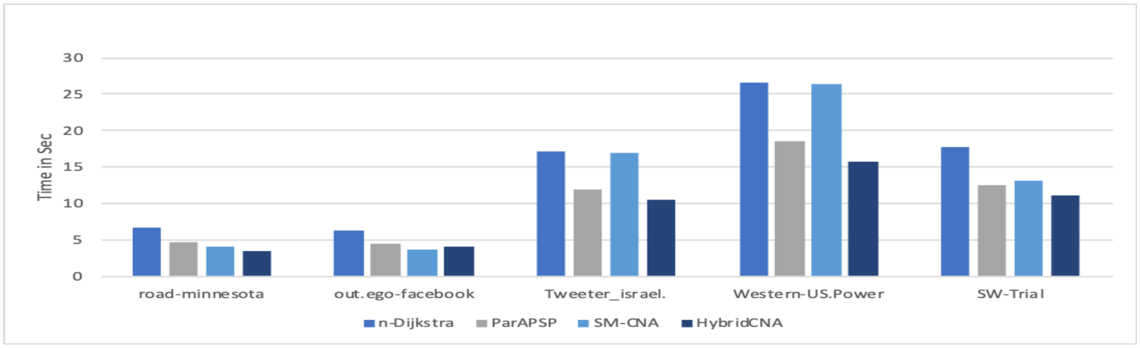


Fig. 15: Testing SM-CNA and HybridCNA methods on real-world networks

Network's Name	Size	Number of partition	Total number of common nodes	Preparing (time in milliseconds)
road-minnesota	2.6K	2	40	103
Western-US.Power	4.9K	3	183	360
Tweeter_israel.	3.6K	2	200	180
out.ego-facebook	2.8K	2	5	141
SW-Trial	10k	7	43	660

TABLE VI: The real networks used in the experiments

compared them with the n-Dijkstra algorithm [9] and the ParAPSP algorithm [10]. The n-Dijkstra algorithm parallelizes the traditional Dijkstra algorithm by running the algorithm from multiple nodes in parallel. The nodes of the graph are divided among the cores and are processed simultaneously until the shortest paths between all nodes are found. The ParAPSP algorithm is a modified Dijkstra's algorithm, which utilizes the information obtained in previous iterations during the processing and uses an ordering approach based on node degree to reduce the parallel overhead. We used a diverse set of graphs, including social media networks, road networks, and power networks. Table VI lists the datasets of these networks, showing the size of the graph, the number of subgraphs we partitioned it into, the total number of common nodes, and finally, the time needed to pre-process the graphs. All the real graphs used in this paper were obtained from SNAP [26], KONECT [27], and Network Repository [28].

Figure 15 shows the comparison between SM-CNA, HybridCNA, n-Dijkstra, and ParAPSP. Observing the performance of SM-CNA and HybridCNA, we can see that when the number of common nodes is small (such as the out.ego-facebook network), SM-CNA achieves better performance. However, if the number of common nodes is large (such as Western-US.Power and Tweeter-israel), HybridCNA delivers much better performance. We can draw the same conclusion when comparing SM-CNA with ParAPSP. Furthermore, from the performance of n-Dijkstra and SM-CNA in Figure 15, we can see that SM-CNA is faster than the n-Dijkstra algorithm in all cases. However, the gap between them decreases when the number of common nodes increases. This outcome matches the results of the experiments conducted on the simulated graph. It motivated us to develop HybridCNA, which achieves higher performance than all other compared algorithms.

Graphs	SuperFW	SM-CNA	HybridCNA
Western-US.Power	3.8	3.3	5.07
Synthetic graph (16k)	6.1	4.09	8.3

TABLE VII: Speedups achieved by SuperFW, SM-CAN, and HybridCNA over Dijkstra

F. *Comparing with SuperFW*

SuperFW is a recently published parallel APSP algorithm [11]. Its main objective is to enhance the well-established Floyd-Warshall algorithm by exploiting the relationship between Floyd-Warshall and Gaussian elimination. We compared the speedups (over Dijkstra) achieved by our methods and SuperFW using both synthetic and real-world graphs. The synthetic graph used has the size of 16k, which is the largest size used in the experiments in previous subsections. We used the real-world graph named *Western-US.power*, which is also the graph used in [11]. The results are listed in Table VII.

As indicated in the table, HybridCNA achieved a speedup of 5.07x on Western-US.Power, surpassing SuperFW's 3.8x speedup. When using the synthetic graph of size 16k, HybridCNA achieved a speedup of 8.3x, outperforming SuperFW's 6.1x speedup. It is worth noting that SM-CNA performs less favorably compared to SuperFW. This is because SM-CNA soly harnesses the multi-core computing capabilities. The design considerations behind SM-CNA are geared towards optimizing the integration between CPU and GPU, with the aim of minimizing the communication between them and maximizing parallelism on the GPU.

VI. Conclusion

In this work, we proposed two novel parallelization methods, SM-CNA and HybridCNA, for solving the all-pairs shortest paths (APSP) problem on multicore and GPU systems. SM-CNA parallelizes the processing of APSP on a multicore computer while HybridCNA utilizes both CPU and GPU to further accelerate the processing. We developed two strategies, H-Thread and H-Block, to efficiently schedule GPU threads. Both strategies showed excellent performance on specific types of graphs. Additionally, HybridCNA can be easily deployed to a system with multiple GPU, achieving up to 2.5x speedup compared to a single GPU. To evaluate our proposed method, we conducted experiments on both synthetic and real-world graphs. The results demonstrate that HybridCNA achieved up to 8.3x speedup over the Dijkstra algorithm, outperforming the existing methods, i.e., n-Dijkstra, ParAPSP, and SuperFW, compared in this paper.

Acknowledgments

The authors extend their appreciation to the Deputyship for Research & Innovation, Ministry of Education in Saudi Arabia for funding this research work through the project number MoE-IF-UJ-22-04330351-1.

References

[1] K. Deb, *Multi-objective optimization using evolutionary algorithms*, vol. 16. John Wiley & Sons, 2001.

[2] S. Rasmussen, M. Talla, and R. Valverde, "Case study on geocoding based scheduling optimization in supply chain operations management," *WSEAS Transactions on Computer Research*, vol. 7, pp. 29–35, 2019.

[3] T. H. Cormen, *Introduction to algorithms*. MIT press, 2009.

[4] E. Cantú-Paz, "A summary of research on parallel genetic algorithms," 1995.

[5] D. B. Johnson, "Efficient algorithms for shortest paths in sparse networks," *Journal of the ACM (JACM)*, vol. 24, no. 1, pp. 1–13, 1977.

[6] R. W. Floyd, "Algorithm 97: shortest path," *Communications of the ACM*, vol. 5, no. 6, p. 345, 1962.

[7] E. W. Dijkstra *et al.*, "A note on two problems in connexion with graphs," *Numerische mathematik*, vol. 1, no. 1, pp. 269–271, 1959.

[8] Y. Lu, J. Cheng, D. Yan, and H. Wu, "Large-scale distributed graph computing systems: An experimental evaluation," *Proceedings of the VLDB Endowment*, vol. 8, no. 3, pp. 281–292, 2014.

[9] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein, *Introduction to algorithms*. MIT press, 2009.

[10] J. W. Kim, H. Choi, and S.-H. Bae, "Efficient parallel all-pairs shortest paths algorithm for complex graph analysis," in *Proceedings of the 47th International Conference on Parallel Processing Companion*, p. 5, ACM, 2018.

[11] P. Sao, R. Kannan, P. Gera, and R. Vuduc, "A supernodal all-pairs shortest path algorithm," in *Proceedings of the 25th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pp. 250–261, 2020.

[12] M. Alghamdi, L. He, Y. Zhou, and J. Li, "Developing the parallelization methods for finding the all-pairs shortest paths in distributed memory architecture," in *2019 IEEE 38th International Performance Computing and Communications Conference (IPCCC)*, pp. 1–8, IEEE, 2019.

[13] Q. Xu, H. Jeon, and M. Annavaram, "Graph processing on gpus: Where are the bottlenecks?," in *2014 IEEE International Symposium on Workload Characterization (IISWC)*, pp. 140–149, IEEE, 2014.

[14] T. Okuyama, F. Ino, and K. Hagihara, "A task parallel algorithm for computing the costs of all-pairs shortest paths on the cuda-compatible gpu," in *2008 IEEE International Symposium on Parallel and Distributed Processing with Applications*, pp. 284–291, IEEE, 2008.

[15] A. McLaughlin and D. A. Bader, "Fast execution of simultaneous breadth-first searches on sparse graphs," in *Parallel and Distributed Systems (ICPADS), 2015 IEEE 21st International Conference on*, pp. 9–18, IEEE, 2015.

[16] K. Matsumoto, N. Nakasato, and S. G. Sedukhin, "Blocked united algorithm for the all-pairs shortest paths problem on hybrid cpu-gpu systems," *IEICE TRANSACTIONS on Information and Systems*, vol. 95, no. 12, pp. 2759–2768, 2012.

[17] M. Nakao, H. Murai, and M. Sato, "Parallelization of all-pairs-shortest-path algorithms in unweighted graph," in *Proceedings of the International Conference on High Performance Computing in Asia-Pacific Region*, pp. 63–72, 2020.

[18] O. Taştan, O. C. Eryüksel, and A. Temizel, "Accelerating johnson's all-pairs shortest paths algorithm on gpu,"

[19] M. Alghamdi, *Developing the parallelization techniques for finding the all-pairs shortest paths in graphs*. PhD thesis, University of Warwick, 2020.

[20] R. E. Korf and P. Schultze, "Large-scale parallel breadth-first search," in *AAAI*, vol. 5, pp. 1380–1385, 2005.

[21] A. Buluç and K. Madduri, "Parallel breadth-first search on distributed memory systems," in *Proceedings of 2011 International Conference for High Performance Computing, Networking, Storage and Analysis*, pp. 1–12, 2011.

[22] J. Barnat, L. Brim, and J. Chaloupka, "Parallel breadth-first search ltl model-checking," in *18th IEEE International Conference on Automated Software Engineering, 2003. Proceedings.*, pp. 106–115, IEEE, 2003.

[23] H. Guo, L. Huang, Y. Lü, J. Ma, C. Qian, S. Ma, and Z. Wang, "Accelerating bfs via data structure-aware prefetching on gpu," *IEEE Access*, vol. 6, pp. 60234–60248, 2018.

[24] JADE: Joint Academic Data Science Endeavour, "https://www.arc.ox.ac.uk/jade." Accessed: 15/08/2023.

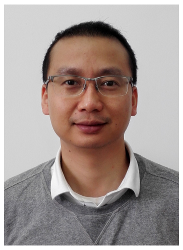
[25] P. Holme and B. J. Kim, "P. holme and bj kim, phys. rev. e 65, 026107 (2002).," *Phys. Rev. E*, vol. 65, p. 026107, 2002.

[26] J. Leskovec and A. Krevl, "SNAP Datasets: Stanford large network dataset collection." http://snap.stanford.edu/data, June 2014.

[27] J. Kunegis, "Konect - the koblenz network collection," 2013.

[28] R. A. Rossi and N. K. Ahmed, "The network data repository with interactive graph analytics and visualization," 2015.

Mohammed H Alghamdi has a Ph.D. in Computer Science from the University of Warwick, UK. His research interest in Distributed Systems, HPC, Parallel Computing, Cloud Computing, and IoT. Currently, he is working as an Assistant Professor at the College of Computer Science and Engineering, University of Jeddah, KSA.



Ligang He is a Reader in the Department of Computer Science at the University of Warwick, UK. His primary research interest lies in the area of parallel and distributed computing. He has published more than 190 papers in the area.

Shenyuan Ren received her Bachelor's degree in Computer Science from Sichuan University, China, in 2014, and her PhD degree in Computer science from the University of Warwick, UK, in 2018. Following that, she was a postdoctoral researcher at the University of Oxford, UK, from 2018 to 2021. She is currently a lecturer in the School of Computer and Information Technology at Beijing Jiaotong University, China, and also a Visiting Scientist in the Clarendon Laboratory, Department of Physics, University of Oxford, UK. Her research interest focuses on high performance Computing, and the intersection of high performance computing and Physics.

Mohammed Maray received the Ph.D. degree from Warwick University, U.K. He is currently an Assistant Professor at King Khalid University, Saudi Arabia, and working as an Assistant Researcher at Petras Project 2020. His current research interests include cloud, the IoT, fog and edge networks, computational offloading, MEC, and MCC. He is a member of ACM, CISSP, and Saudi Academy of Engineering.