# A Parallel Hybrid Framework for Graph Processing

Ashwin Shivapuram, Somay Jain, Chirag Majithia, Virinchi Srinivas
ashwinsl@cs.umd.edu, somay@cs.umd.edu,
chiragm@terpmail.umd.edu, virinchi@cs.umd.edu

**Abstract**

Graphs are fundamental data structures that can capture relationship across different entities. Graphs can be used to model complicated data in different domains like social networks, web graphs, biological networks etc. Currently, the rate at which the data is being produced makes it very common for us to observe graphs with millions of nodes and billions of edges. In order to process these large graphs in an efficient fashion, we need to be able to exploit the inherent parallelism present in the graph traversals to run various graph based algorithms.

In this paper, we explore the possibility to build a hybrid graph processing framework that uses both parallel models - shared memory and message passing interface to efficiently execute various graph traversal algorithms to a larger scale by parallelizing. By using this hybrid approach we were able to achieve consistent and considerable speed-up of up to 9x over the serial implementation of the algorithms.

## 1 Introduction

Graphs are a powerful abstraction for representing underlying relations in large unstructured datasets. The wide applicability of graphs with applications in search engines, social networks, telecommunication networks etc. has given a lot of interest in processing large graphs. In many of the graph algorithms like Breadth First Search(BFS), Depth First Search(DFS) etc. we can simultaneously explore different vertices independently. By exploiting this inherent parallelization we can enhance the performance of these algorithms by simple parallelization strategies.

There exists a plethora of graph processing systems [1, 2, 3, 4, 5, 6]. In all these systems, each compute node performs some computation and send/receives from its adjacent compute nodes. These systems need to deal with race conditions. Ligra [7] presents a framework for graph processing using a shared memory model. Shared memory algorithms tend to be much simpler when compared to message passing based systems. Although, today's servers can support more than a terabyte of memory in a single multi-core machine, but not all hardware present in servers can accommodate terabytes of memory. Even if it can, the shared memory bus becomes a performance bottleneck by using just shared memory parallelization. Hence, taking this situation into consideration, we explore to build a graph processing system which can be partitioned and split across various compute nodes so that the whole graph can be distributed across many compute nodes. Further, within each compute node, we focus to exploit any parallelism using a shared memory model.

In this paper, we are using MPI (Message Passing Interface) to communicate among different compute nodes in the distributed system setting. We are also using OpenMP directives to exploit the parallelization locally within a single compute node. In this paper, we present few of the graph algorithms like Breadth First Search, Triangle Counting, Page Rank and Clustering Coefficient(CC) implemented using the hybrid parallelization strategy involving both the message passing and shared memory parallelization paradigms. Our results suggest that using this strategy provides considerable performance gain of up to 9x over the serial implementation of the graph algorithms.

## 2 Related Work

Pregel [1] introduced by Google and implemented in C/C++, is one of the first Bulk Synchronous Protocol (BSP) implementations that is based on a "think like a vertex" computing paradigm. In each iteration (superstep), each vertex performs some computation and pushes the result of each computation to each of its neighbors along the edges. The system being based on BSP, collects messages which can be seen only in the next iteration. Apache Giraph [5] and GPS [3] are open source implementations of Pregel having some additional features. They provide a marginal improvement over Pregel.

Pegasus [6], uses Hadoop implementation of MapReduce in the distributed computing setting. GraphLab [2] is a framework that relies on GAS (gather, apply, scatter) processing model which is a pull-based model fundamentally different from the push-based models based on BSP. PowerGraph [4] is another system which borrows ideas from both GraphLab and Pregel primarily suited for power-law graphs which have a skewed degree distribution. [8, 9] are survey papers presenting the wide range of graph processing systems that are present today. Nscale [10] is a state-of-the art approach in the context of graph partitioning. They propose to pack subgraphs into minimum number of partitions while keeping the load across different partitions. However, in this project we look to partition the graph into different non-overlapping subgraphs as opposed to Nscale which extracts and loads subgraphs which are specified as Datalog-based pattern queries to be extracted from the larger graph. Further, graph partitioning happens to be a minor step in our project.

Ligra [7] is a graph processing framework which is specific for shared-memory architecture which makes graph traversal algorithms easy to write. The framework relies on two simple primitives EdgeMap and VertexMap for mapping over edges and vertices respectively. These simple routines can be applied to various graph traversal algorithms that operate on a subset of vertices of the graph during each iteration. We are different from the earlier mentioned systems in the sense that we look to attain efficiency within one node using shared memory model and scalability across nodes using message passing by partitioning the graph across multiple computing nodes.

## 3 Approach and Implementation

We propose an hybrid approach using both OpenMP and MPI to implement graph algorithms like Breadth First Search (BFS), Page Rank, Counting Triangles and Clustering Coefficient(CC). While Ligra [7] works well on a single compute node, it cannot be efficiently used in cases where the graph does not fit in memory. In this paper, we propose to partition the graph between multiple compute nodes with necessary replication of boundary nodes across partitions and use both MPI and OpenMP to implement the algorithms. We use MPI for any communication with the neighboring compute nodes and OpenMP to exploit any parallelism within a single compute node. We explain in detail the graph partitioning and each of the algorithms next.

### 3.1 Graph Partitioning and Subgraph Distribution

The first step in our approach would be to partition the graph. We present our graph partitioning approach in Algorithm 1.

**Algorithm 1** Graph Partitioning Approach

```
1: procedure PARTITION($G, p$)                        ▷ Partitioning G=(V,E) into p partitions
2:     Download and load the graph dataset G                         ▷ We are using SNAP.
3:     Preprocess the graph dataset to renumber the graph vertices from 0.
4:     Partition the preprocessed graph dataset using METIS.
5:     return node to partition map
```

Once we load the graph dataset, we preprocess the graph to renumber the vertex IDs to start from 0. Once the preprocessing is done we use METIS [11] to partition the graph into required number of partitions. METIS uses a single computing node to partition the graph assigning each vertex to a partition. METIS partitions the graph by minimizing the number of the edges that are cut across various partitions. This is a better partitioning scheme than a naive assignment of $n/p$ continuous vertices to each partition, where $n$ is the number of vertices and $p$ is the number of partitions. Hence, we resort to using METIS for graph partitioning which ensures proper balancing of nodes across different partitions. Further, although ParMetis is much more efficient than Metis, we propose to use ParMetis for future work; setting up using ParMetis is much more complex.

Once every node has been assigned to a unique partition, we logically construct the subgraph of each partition by connecting all the edges among the vertices that are local to the given partition(compute node). Further, we also add edges between vertices that are local to given processor and ghost(remote) vertices. This constructs the subgraph that can be loaded on every partition either in parallel or the root node performs the subgraph construction and distributes the subgraph pertaining to each processor. Further, by adding edges among the ghost vertices in any constructed subgraph helps in significantly reducing message passing overhead between different compute nodes. Only in the case of computing Clustering Coefficient(CC), we load the subgraphs pertaining to each partition in parallel by different compute nodes and we also connect the edges among the ghost(remote) vertices in this case. We explain in detail the approach and implementation of each algorithm next.

### 3.2 Breadth First Search

In this paper, we implement a parallel version of BFS algorithm from scratch using MPI and OpenMP as described in Algorithm 2. BFS can be run in parallel by processing the vertices at each level in parallel. After we partition the graph using Algorithm 1, we read the partitioned graph from the MPI compute node 0 (root node) and distribute the subgraphs to the other compute nodes.

**Algorithm 2** Breadth First Search

```
 1: procedure BFS($G, root, p$)                                   ▷ Graph G=(V,E); $root$ : root vertex
 2:     Load the p-partitioned Graph into p MPI nodes.
 3:     $frontier \leftarrow$ neighbours of $root$
 4:     $level \leftarrow 1$
 5:     Initialize $visited$ array of booleans as $false$
 6:     $visited[root] \leftarrow true$
 7:     $distance[root] \leftarrow 0$
 8:     while $frontier \neq null$ do
 9:         $local \leftarrow$ vertices in $frontier$ owned by local node.
10:         $remote \leftarrow$ vertices in $frontier$ owned by remote nodes.
11:         $frontier \leftarrow null$
12:         for all $r \in remote$ do
13:             send $r$ using MPI to its owner node.
14:         end for
15:         #pragma omp parallel for                                  ▷ OpenMP directive
16:         for all $l \in local$ do
17:             if $visited[l] == false$ then
18:                 $frontier \leftarrow$ neighbours of $l$
19:                 $distance[l] \leftarrow level$
20:                 $visited[l] \leftarrow true$
21:             end if
22:         end for
23:         for each partitions $p$ do
24:             $remote[p] \leftarrow$ neighbours received from $p$-th partition     ▷ Sent in line 13
25:             #pragma omp parallel for                              ▷ OpenMP directive
26:             for all $r \in remote[p]$ do
27:                 if $visited[r] == false$ then
28:                     $frontier \leftarrow$ neighbours of $r$
29:                     $distance[r] \leftarrow level$
30:                     $visited[r] \leftarrow true$
31:                 end if
32:             end for
33:         end for
34:         $level \leftarrow level + 1$
35:     end while
36:     return $distance$
```

While traversing the vertices at one level, the vertices for the next level are added to a queue - *frontier*. The vertices in the *frontier* which belong to a different compute node are sent to the corresponding compute nodes using asyncronous MPI send. In order to decrease time spent on waiting for the receives from the remote compute nodes we process the locally owned vertices in the meantime. The processing of locally owned vertices is furthur parallelized by using OpenMP directives. In the end, when a compute node receives the list of vertices locally owned vertices from remote nodes, it adds them to it's own queue and continues the traversal. The compute nodes are synchronized at the end of each level using MPI Barrier. This is continued on till all the vertices which can be reached by the *root* vertex are visited and the distance to all the reachable vertices is returned.

## 3.3 Triangle Counting

A triangle in an undirected graph is a clique of size 3. The triangle count of a graph is an important metric in graph analytics because it gives an intuitive measure of how clustered the graph is, and also finds applications in a variety of areas such as social networks, spam detection, and link classification.

Similar to the BFS implementation, we are proposing an hybrid implementation of Triangle Counting in a graph that uses both MPI and OpenMP to exploit the inherent parallelism present in the Triangle Counting algorithm. The procedure is as described in Algorithm 3.

**Algorithm 3** Triangle Counting

```
 1: procedure TRIANGLE COUNTING($G, p$)                              ▷ Graph G=(V,E)
 2:     Load the p-partitioned Graph into p MPI nodes.
 3:     $G \leftarrow$ Convert To Directed Graph($G$)                   ▷ using Algorithm 4
 4:     $triangles \leftarrow 0$
 5:     for all $v \in V[p]$ do              ▷ V[p] - Vertices belonging to $p$-th partition
 6:         $local[v] \leftarrow$ neighbours owned by local node.
 7:         $remote[v] \leftarrow$ neighbours owned by remote nodes.
 8:         for all $r \in remote[v]$ do
 9:             Request adjacency list of $r$ using MPI
10:         end for
11:         for all $l \in local[v]$ do
12:             $triangles \leftarrow triangles +$ Count Intersections($v, l$)    ▷ using Algorithm 5
13:         end for
14:         for all $r \in remote[v]$ do
15:             Receive adjacency list of $r$ using MPI
16:             $triangles \leftarrow triangles +$ Count Intersections($v, r$)    ▷ using Algorithm 5
17:         end for
18:     end for
19:     $triangles \leftarrow$ MPI_reduceAll($triangles$,MPI_SUM).     ▷ Adding all computed triangles
20:     return $triangles$
```

As we can count the same triangle thrice, once from each of the three vertices which constitutes it, the duplicate triangle counting is a repetitive and resource/time consuming process. In our approach as we are splitting the graph into partitions, maintaining a visited map and synchronizing it across all parallel nodes is a complex and time consuming task. Hence it becomes a bottleneck to the parallel implementation of the algorithm and degrades the performance. So in order to alleviate these problems we convert the graph into a desired directed graph using Algorithm 4.

In Algorithm 4 we are converting undirected edges into directed edges by retaining only one direction of each edge. The direction retained is from the vertex with lower *globalId* to the vertex with higher *globalId* and the vice-versa is removed. This implicitly makes the triangle visible only to the vertex with the lowest *globalId* among the three vertices forming the triangle and hence resolves the duplicate triangle counting issue.

Once we have generated the directed graph using the Algorithm 4, we can count the triangles which share the edge $(v, u)$ using Algorithm 5. This utilizes the OpenMP directives to parallelize the local execution.

In Algorithm 3, for each vertex in a parition, we group its neighbors into locally and remotly owned vertices. In order to maximize the performance, we request the adjacency list of the remotely owned vertices using asynchronous send (MPI) before any computation begins. While we wait to receive the adjacency lists from the remote nodes, we compute the triangles formed by the locally owned vertices using Algorithm 5 which utilizes OpenMP. On receiving the adjacency list of the remotely owned vertices, we use Algorithm 5 again to compute the triangles that share the remote vertex. Once all the vertices

**Algorithm 4** Conversion to Directed Graph

```
1: procedure CONVERT TO DIRECTED GRAPH($G$)                ▷ Graph G=(V,E)
2:     for all $v \in V$ do
3:         for all $u \in AdjList[v]$ do
4:             $AdjList[v] \leftarrow$ adjacency list of vertex $v$
5:             if $u <= v$ then
6:                 Remove $u$ from $AdjList[v]$
7:             end if
8:         end for
9:     end for
10:    return directed graph
```

**Algorithm 5** Counting Intersections

```
1: procedure COUNT INTERSECTIONS($v, u$)    ▷ $v$ - Base Vertex; $u$ - Connected Vertex
2:     AdjList[$v$] $\leftarrow$ Adjacency list of vertex $v$
3:     AdjList[$u$] $\leftarrow$ Adjacency list of vertex $u$
4:     count $\leftarrow 0$
5:     #pragma omp parallel for                               ▷ OpenMP directive
6:     for all $x \in AdjList[v]$ do
7:         if $x \in AdjList[u]$ then
8:             Increment count
9:         end if
10:    end for
11:    return count
```

in the partition are processed, we use MPI reduction to sum all the triangles counted by each compute node and returns the total number of triangles in the graph.

## 3.4 Page Rank

Page Rank of a node in a Graph G=(V,E) signifies its importance. It is an important measure used in various applicants like search engines, social media analysis etc. We use an iterative algorithm to compute the Pank Rank of all the nodes. The sketch of our implementation is as shown in Algorithm 6 which uses both MPI and OpenMP parallelization paradigms.

For any node v, the Page Rank is computed as follows:

$$pagerank(v) = \frac{(1-d)}{N} + d \sum_{n \in neighbor(v)} \frac{pagerank(n)}{degree(n)} \tag{1}$$

where, d is a damping factor set to 0.85. We can split Equation 1 as follows:

$$pagerank(v) = constant + local(v) + remote(v) \tag{2}$$

where,

$$constant = \frac{(1-d)}{N} \tag{3}$$

$$local(v) = d \sum_{n \in local-neighbor(v)} \frac{pagerank(n)}{degree(n)} \tag{4}$$

$$remote(v) = d \sum_{n \in remote-neighbor(v)} \frac{pagerank(n)}{degree(n)} \tag{5}$$

Observe from Equation 2 that the page rank computation for any node has three parts. The first part, *constant* (Equation 3) is a constant for all the nodes. The second part, $local(v)$ (Equation 4) which consists of local neighbors can be computed in the local compute node as all the values required are available locally. The third part, $remote(v)$ (Equation 5) consists of information that is to be fetched from neighboring compute nodes where the remote vertices reside. In order to exploit parallelism, we first request $\frac{pagerank(n)}{degree(n)}$ from each non-local neighbor $n$ using MPI's asynchronous send. While we wait to receive this information from other compute nodes, we can in parallel compute the *constant* (Equation 3) and the $local(v)$ (Equation 4) locally in every node using OpenMP. Finally, on receiving the information from all the other compute nodes, we compute $remote(v)$ (Equation 5) to complete one iteration of page rank. We perform 10 iterations of page rank to achieve convergence. The details of the implementation is as shown in Algorithm 6.

**Algorithm 6** Page Rank

```
1:  procedure PAGE RANK($G, p$)                              ▷ Graph G=(V,E)
2:      Load the p-partitioned Graph into p MPI nodes.
3:      for all $v \in V[p]$ do             ▷ V[p] - Vertices belonging to $p$-th partition
4:          $local[v] \leftarrow$ neighbours owned by local node.
5:          $remote[v] \leftarrow$ neighbours owned by remote nodes.
6:          for all $l \in local[v]$ do
7:              $p[l] \leftarrow 0$                   ▷ Assign page rank of each local vertex to 0
8:      $iterations \leftarrow 10$                   ▷ Number of iterations of Page Rank
9:      $step \leftarrow 0$                                    ▷ Current Iteration
10:     while $step < iterations$ do
11:         for all $r \in remote[v]$ do
12:             Request $\frac{p[r]}{degree(r)}$ using MPI
13:         #pragma omp parallel for                       ▷ OpenMP directive
14:         for all $l \in local[v]$ do                         ▷ Constant value
15:             $constant[l] \leftarrow \frac{0.15}{|V|}$
16:             $localsum[l] \leftarrow 0$
17:         #pragma omp parallel for                       ▷ OpenMP directive
18:         for all $l \in local[v]$ do                      ▷ Local Computation
19:             for all $n \in neighbor(l) \wedge n$ is local do
20:                 $localsum[l] \leftarrow localsum[l] + \frac{p[n]}{degree(n)}$
21:         for all $l \in local[v]$ do                     ▷ Remote Computation
22:             for all $n \in neighbor(l) \wedge n$ is remote do
23:                 Receive $\frac{p[n]}{degree(n)}$ from remote $n$.
24:                 $localsum[l] \leftarrow localsum[l] + \frac{p[n]}{degree(n)}$
25:         for all $l \in local[v]$ do
26:             $p[l] \leftarrow p[l] + constant[l] + 0.85 \times localsum[l]$
27:         $step \leftarrow step + 1$
28:     $page\_rank \leftarrow MPI\_Gather(p, root)$        ▷ Gather all page ranks at root
29:     return $page\_rank$
```

## 3.5 Clustering Coefficient (CC)

Clustering coefficient (CC) is a measure of each node which represents how clustered its neighborhood is. This measure is used extensively in social networks, biological networks to find the degree of connectivity between neighbors of a given node.

For a given node v, we compute the CC as follows :

$$CC(v) = \frac{\text{Number of edges between neighbors of v}}{\binom{N}{2}} \tag{6}$$

where $N$ is the number of neighbors of $v$ and the value of CC ranges between 0 and 1. We can observe that CC indicates how densely or sparsely a neighborhood of a node is. Higher the value of CC, higher is the neighborhood density.

As explained in subgraph distribution, we connect edges between remote vertices for implementing this algorithm. This enables to perform computing CC of all local nodes within the local compute node without any dependency on neighboring compute nodes. We exploit the parallelism of this local CC computation in parallel using OpenMP. On completing the local computation, each compute node takes a sum of the CC of all its local nodes. Using MPI ReduceAll, these values are sent to the MPI's root compute node. The fraction of sum of all acquired values from each compute node (sum of clustering coefficient of every node in the graph) to the number of nodes in the graph results in the global clustering coefficient. Detailed implementation of our Clustering Coefficient(CC) algorithm is as shown in Algorithm 7 below.

**Algorithm 7** Clustering Coefficient

```
1:  procedure CLUSTERING COEFFICIENT (CC)($G, p$)                    ▷ Graph G=(V,E)
2:      Load the p-partitioned Graph into p MPI nodes.               ▷ Perform in Parallel
3:      for all $v \in V[p]$ do                                      ▷ V[p] - Vertices belonging to $p$-th partition
4:          $local \leftarrow$ nodes local to p
5:          for all $l \in local[v]$ do
6:              $cc[l] \leftarrow 0$                                 ▷ Assign cc of each local vertex to 0
                                     ▷ Adding edges between remote vertices ensures no message passing
7:      #pragma omp parallel for                                     ▷ OpenMP directive
8:      for all $l \in local$ do                                     ▷ Local Computation
9:          $num \leftarrow 0$
10:         for all $n1 \in neighbor(l)$ do
11:             for all $n2 \in neighbor(l)$ do
12:                 if edge(n1,n2) then
13:                     $num \leftarrow num + 1$
14:         $cc[l] \leftarrow \frac{2 \times num}{degree(l) \times (degree(l)-1)}$
15:     $cc \leftarrow MPI\_reduceAll(sum(cc), MPI\_SUM)$              ▷ Gather all local CC at root
16:     $globalCC \leftarrow \frac{cc}{|V|}$
17:     return $globalCC$
```

## 4 Datasets

A lot of real world graph datasets are freely available online [1], having thousands to millions of nodes and edges from various domains such as social media, communication networks, road networks, etc. Specifically, we have extracted four undirected networks whose statistics are shown in Table 1. In Table 1, note that $|V|$, $|E|$ and CC refer to the number of vertices, edges and Average Clustering Coefficient (CC) of the graph respectively. The diameter of the graph is defined as the longest shortest path. The graphs show variance in their size, diameter and also the average clustering coefficient of the graph which corresponds to the density of the graph. We observe that DBLP is the smallest graph ranging from around 300,000 nodes and California road network is the largest graph having close to 2 million nodes. Further, based on the CC value, we can observe that road-network graph is the most sparse while DBLP is the most dense.

| Dataset | $\|V\|$ | $\|E\|$ | CC | diameter |
|---|---|---|---|---|
| DBLP | 317080 | 1049866 | 0.6324 | 21 |
| Amazon | 334863 | 925872 | 0.3967 | 44 |
| YouTube | 1134890 | 2987624 | 0.0808 | 20 |
| California Road-network | 1965206 | 2766607 | 0.0464 | 849 |

Table 1: Description of the graph databases used

[1]https://snap.stanford.edu/data/

## 5 Results

### 5.1 Experimental Results

We setup the experiments on UMD cluster `deepthought2`. As part of these experiments, we ran all the four different graph algorithms we have explored in this paper in two different settings. One with just the MPI and other by using OpenMP along with the MPI(hybrid) and we compare them against the serial implementation. We ran the both MPI and hybrid parallel algorithms by varying the number of computing resources provided as 2, 4, 8 and 16 nodes with one MPI process per compute node and 16 OpenMP threads per node.

In general we observe a execution time speed-up of up-to 9x across all the four algorithms. We can also observe that the Road Network and YouTube datasets take longer running time when compared to both Amazon and DBLP datasets, which is due to their graph size. Further, as we increase the number of parallel nodes, the execution time is decreasing steadily indicating that the algorithm is able to gain performance as more parallelism is introduced. This trend is visible across all datasets and consequently using 16 MPI nodes provide the best performance gain among all. Further, depending on the graph structure, we also observed a trend of decreasing performance or very slight increase in performance when using OpenMP along with the MPI. We describe the reason as to why this happened in the Challenges section below.

We present the running time of BFS algorithm in Table 2. The speedup of both parallel implementations (MPI and Hybrid) BFS algorithms over different datasets is shown in Figure 1. The speed-up of parallel process is approximately 4.5 times with just 2 parallel MPI compute nodes (one process each in a node) over all the datasets. We observe a good speedup attaining up to 7x over the larger YouTube and Road Network datasets and up to 9x speed-ups over the relatively smaller Amazon and DBLP datasets. As stated above, when using the OpenMP along with MPI, we see a trend of performance decrease. We believe this is more observable in BFS because the relative work done in each iteration of the inner loop of the BFS algorithm is very minimal and hence the overhead to merge the private lists maintained for OpenMP threads was greater than the gain achieved.

| Number of Processors | Youtube | | Road Network | | Amazon | | DBLP | |
|---|---|---|---|---|---|---|---|---|
| | MPI | Hybrid | MPI | Hybrid | MPI | Hybrid | MPI | Hybrid |
| Serial (1) | 735 | | 927 | | 183 | | 194 | |
| Parallel (2) | 640 | 1278 | 885 | 1601 | 161 | 596 | 163 | 586 |
| Parallel (4) | 357 | 839 | 568 | 1309 | 87 | 380 | 95 | 399 |
| Parallel (8) | 210 | 651 | 273 | 1025 | 48 | 298 | 54 | 295 |
| Parallel (16) | 116 | 571 | 188 | 717 | 27 | 327 | 30 | 305 |

Table 2: Runtime (in milliseconds) for BFS (100 iterations) on Different Datasets

The running time for Triangle Counting (Table 3), Clustering Coefficient (Table 4) and Page Rank (Table 5) also show performance gain when compared to their serial implementation. The speed-up achieved by Triangle Counting, Clustering Coefficient and Page Rank are as shown in Figure 2, Figure 3 and Figure 4 respectively and we see a performance gain of up to 9x using just MPI(16 nodes) and we see comparable performance when using OpenMP along with the MPI.

| Number of Processors | YouTube | | Road Network | | Amazon | | DBLP | |
|---|---|---|---|---|---|---|---|---|
| | MPI | Hybrid | MPI | Hybrid | MPI | Hybrid | MPI | Hybrid |
| Serial (1) | 312.18 | | 2.11 | | 2.66 | | 5.59 | |
| Parallel (2) | 367.56 | 363.58 | 1.65 | 1.79 | 2.08 | 1.89 | 3.87 | 3.88 |
| Parallel (4) | 361.92 | 359.26 | 0.93 | 1.05 | 1.02 | 1.1 | 2.5 | 2.35 |
| Parallel (8) | 224.2 | 222.62 | 0.68 | 0.99 | 0.63 | 0.66 | 1.39 | 1.35 |
| Parallel (16) | 223.37 | 215.3 | 0.53 | 0.38 | 0.37 | 0.36 | 0.91 | 0.74 |

Table 3: Runtime (in seconds) for Triangle Counting on Different Datasets

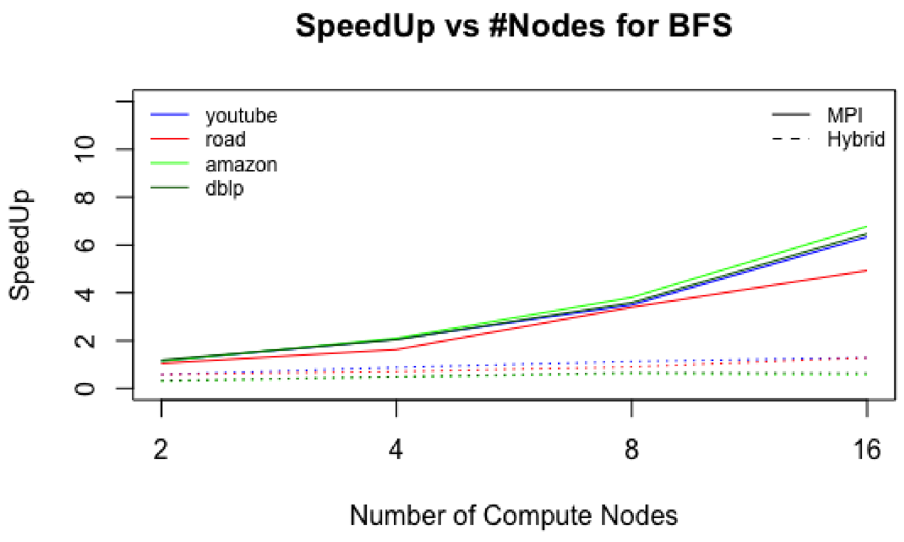

Figure 1: Speed up for BFS using MPI and Hybrid

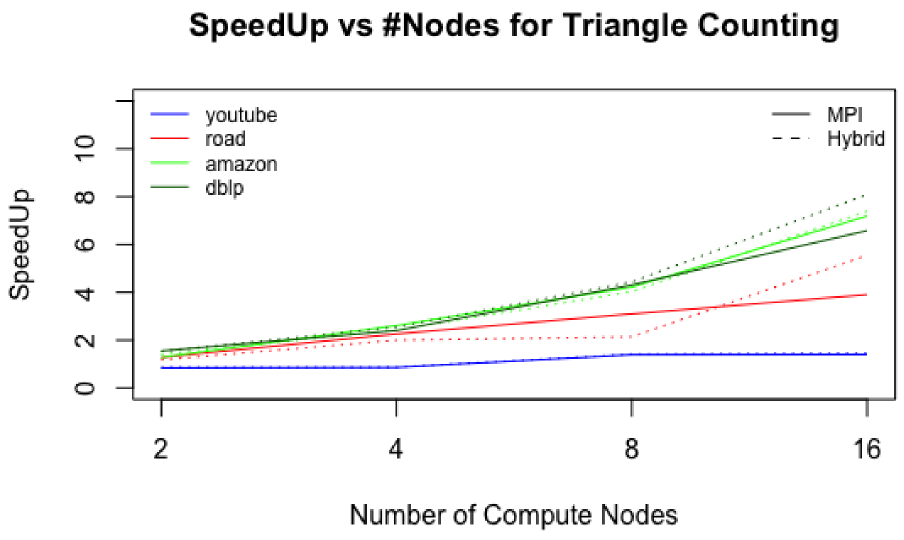

Figure 2: Speed up for Triangle Counting using MPI and Hybrid

| Number of Processors | Youtube | | Road Network | | Amazon | | DBLP | |
|---|---|---|---|---|---|---|---|---|
| | MPI | Hybrid | MPI | Hybrid | MPI | Hybrid | MPI | Hybrid |
| Serial (1) | 307.04 | | 2.31 | | 2.86 | | 5.47 | |
| Parallel (2) | 437.51 | 523.08 | 2.66 | 2.35 | 2.29 | 2.23 | 4.49 | 4.43 |
| Parallel (4) | 419.16 | 415.9 | 1.35 | 1.26 | 1.38 | 1.34 | 2.86 | 2.64 |
| Parallel (8) | 266.93 | 324.55 | 0.69 | 0.66 | 0.88 | 0.74 | 1.68 | 1.36 |
| Parallel (16) | 263.39 | 288 | 0.45 | 0.49 | 0.44 | 0.46 | 1.1 | 0.78 |

Table 4: Runtime (in seconds) for Clustering Coefficient on Different Datasets

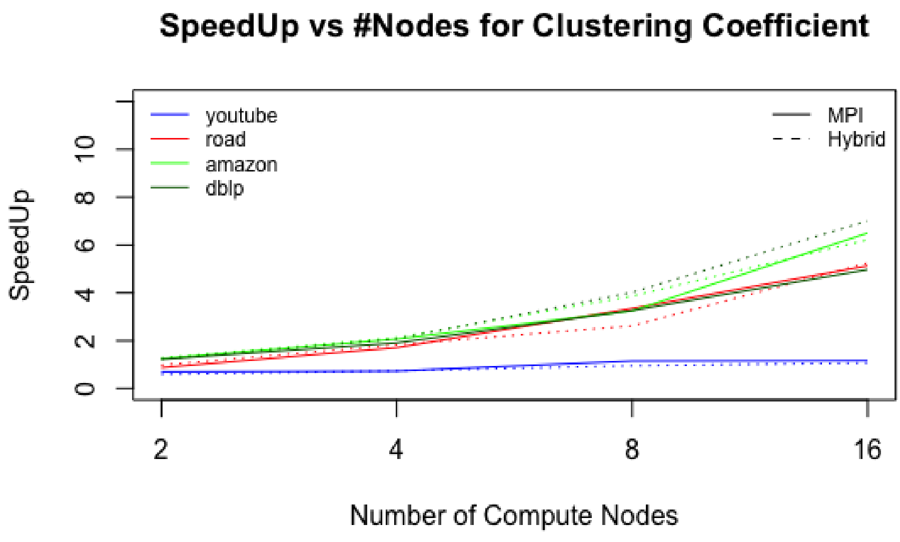

Figure 3: Speed up for Clustering Coefficient using MPI and Hybrid

| Number of Processors | Youtube | Road Network | Amazon | DBLP |
|---|---|---|---|---|
| Serial (1) | 10.4 | 5.95 | 2.89 | 4.7 |
| Parallel (2) | 8.3 | 3.47 | 1.4 | 2.14 |
| Parallel (4) | 5.06 | 1.85 | 1.43 | 1.51 |
| Parallel (8) | 4.42 | 1.04 | 1.05 | 1.26 |
| Parallel (16) | 4.42 | 0.63 | 1.09 | 1.2 |

Table 5: Runtime (in seconds) for Page Rank (10 iterations) using MPI on Different Datasets

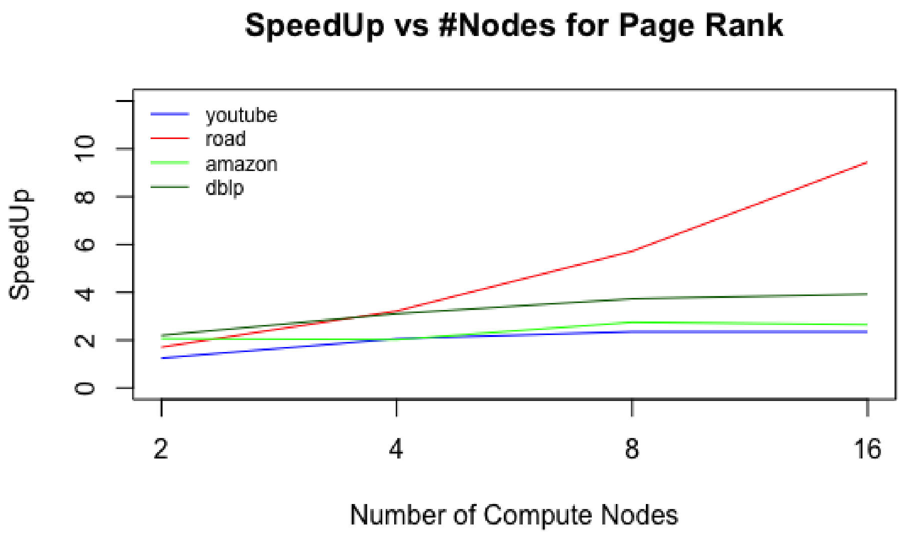

Figure 4: Speed up for Page Rank (10 iterations) using MPI

## 5.2 Challenges

Though the hybrid strategy of using both message passing (MPI) and shared memory (OpenMP) parallelization results in enhancing efficiency and scalability of algorithms, we faced few scenarios where this hybrid parallel strategy did not perform as expected when compared to the corresponding serial implementation on few algorithms. Some scenarios are as follows:

- Load balancing: Due to graph splitting strategies, sometimes only few of the compute nodes handle bulk of the work. In these cases the message passing between the nodes become a bigger overhead than the overall performance gain, which affects both MPI and hybrid implementations. This can be seen in the case of YouTube graph where serial implementation performs better than the parallel implementation on 2 and 4 threads per node in the case of Triangle Counting and CC (Table 3 and Table 4).
- OpenMP critical sections: In some scenarios, like BFS, as we can see from Table 2 and Figure 1 the hybrid strategy was slower than just using MPI to parallelize the code. We believe the reason for this negative trend is because the relative work done in each iteration of the loop was minimal and adding OpenMP directives require critical sections such as merging lists, or adding to a list.
- OpenMP nested loops: In cases like triangle counting and clustering coefficient, the computation for each vertex needs a nested loop to iterate over it's neighbours and it's neighbour's neighbours. Though these iterations can be done in parallel by adding OpenMP directives, these are nested loops which require fork-join of OpenMP threads at every outer loop iteration.

## 6 Conclusion and Future Work

We are able to implement fast parallel graph algorithms like BFS, Triangle Counting, Page Rank and Clustering Coefficient which have up to 9x speed-up over their serial counterparts using the hybrid approach of both message passing (MPI) and shared memory (OpenMP) parallelization paradigms. In future we should be able to extend the same hybrid parallelization approach to other graph algorithms like Shortest Path, Spanning Tree etc. which also have inherent parallelization in their design.

As mentioned in the Challenges, the implementation approach used in this paper have considerable performance dependency on the graph partitioning strategy as it does not take into account if every compute node is equally sharing the work. So in future we can improve these graph algorithms by enabling the load balancing among the compute nodes themselves so all the nodes are doing equal amount of work. Also in this paper we are using METIS[11] library to partition the graph, while METIS is an efficient tool to partition the graph, ParMetis(Parallel METIS) provides a performance gain over METIS. In future we should be able to support the graph partitioning using ParMetis which can help us in reducing the graph loading time.

## References

[1] G. Malewicz, M. H. Austern, A. J. Bik, J. C. Dehnert, I. Horn, N. Leiser, and G. Czajkowski, “Pregel: a system for large-scale graph processing,” in *Proceedings of the 2010 ACM SIGMOD International Conference on Management of data*. ACM, 2010, pp. 135–146.

[2] Y. Low, J. E. Gonzalez, A. Kyrola, D. Bickson, C. E. Guestrin, and J. Hellerstein, “Graphlab: A new framework for parallel machine learning,” *arXiv preprint arXiv:1408.2041*, 2014.

[3] S. Salihoglu and J. Widom, “Gps: A graph processing system,” in *Proceedings of the 25th International Conference on Scientific and Statistical Database Management*. ACM, 2013, p. 22.

[4] J. E. Gonzalez, Y. Low, H. Gu, D. Bickson, and C. Guestrin, “Powergraph: Distributed graph-parallel computation on natural graphs.” in *OSDI*, vol. 12, no. 1, 2012, p. 2.

[5] “Giraph : "http://giraph.apache.org",” 2012.

[6] E. Deelman, G. Singh, M.-H. Su, J. Blythe, Y. Gil, C. Kesselman, G. Mehta, K. Vahi, G. B. Berriman, J. Good *et al.*, “Pegasus: A framework for mapping complex scientific workflows onto distributed systems,” *Scientific Programming*, vol. 13, no. 3, pp. 219–237, 2005.

[7] J. Shun and G. E. Blelloch, “Ligra: a lightweight graph processing framework for shared memory,” in *ACM Sigplan Notices*, vol. 48, no. 8. ACM, 2013, pp. 135–146.

[8] R. R. McCune, T. Weninger, and G. Madey, “Thinking like a vertex: a survey of vertex-centric frameworks for large-scale distributed graph processing,” *ACM Computing Surveys (CSUR)*, vol. 48, no. 2, p. 25, 2015.

[9] O. Batarfi, R. El Shawi, A. G. Fayoumi, R. Nouri, A. Barnawi, S. Sakr *et al.*, “Large scale graph processing systems: survey and an experimental evaluation,” *Cluster Computing*, vol. 18, no. 3, pp. 1189–1213, 2015.

[10] A. Quamar, A. Deshpande, and J. Lin, “Nscale: neighborhood-centric large-scale graph analytics in the cloud,” *The VLDB Journal*, vol. 25, no. 2, pp. 125–150, 2016.

[11] G. Karypis and V. Kumar, “A fast and high quality multilevel scheme for partitioning irregular graphs,” *SIAM Journal on scientific Computing*, vol. 20, no. 1, pp. 359–392, 1998.