Scalable High-Performance Community Detection Using Label Propagation in Massive Networks

Sharon Boddu and Maleq Khan

Department of Electrical Engineering and Computer Science
Texas A&M University-Kingsville
sharonbdd25@gmail.com, maleq.khan@tamuk.edu

Abstract. Community detection is the problem of finding naturally forming clusters in networks. It is an important problem in mining and analyzing social and other complex networks. Community detection can be used to analyze complex systems in the real world and has applications in many areas, including network science, data mining, and computational biology. *Label propagation* is a community detection method that is simpler and faster than other methods such as Louvain, InfoMap, and spectral-based approaches. Some real-world networks can be very large and have billions of nodes and edges. Sequential algorithms might not be suitable for dealing with such large networks. This paper presents distributed-memory and hybrid parallel community detection algorithms based on the label propagation method. We incorporated novel optimizations and communication schemes, leading to very efficient and scalable algorithms. We also discuss various load-balancing schemes and present their comparative performances. These algorithms have been implemented and evaluated using large high-performance computing systems. Our hybrid algorithm is scalable to thousands of processors and has the capability to process massive networks. This algorithm was able to detect communities in the Metaclust50 network, a massive network with 282 million nodes and 42 billion edges, in 654 seconds using 4096 processors.

Keywords: Community detection · parallel graph algorithms, · network analysis · graph mining

1 Introduction

Graphs can be used to represent many complex real-world systems and networks. A node in a network can be represented by a vertex in a graph, and a link between nodes by an edge. Therefore, we use the terms graphs and networks, as well as nodes and vertices, interchangeably. We can find many networks in the real world. For example, the world-wide web is a network of HTML pages connected by hyperlinks [3]. Similarly, social networks are made up of individuals represented as vertices and the interpersonal relationships between them as edges [7, 11, 16]. Identifying the organization of vertices and the presence of a modular structure or communities can help to understand these complex systems [16, 17]. Real-world networks often exhibit community structures [16]. A community within a network can be formally said to be a group of nodes with more edges connecting to other vertices within the group than vertices outside the group [17,20]. In many cases, communities tend to emerge organically within complex systems, and these communities can represent meaningful subsystems with a specific purpose or function. In the world-wide web, a community might correspond to a set of web pages with similar content or user interests [3]. In a protein-protein interaction network, a community represents a group of proteins with similar functionalities [15]. A community in a social network might correspond to a group of like-minded individuals with common interests [1].

There are some existing algorithms for community detection. Well-known algorithms for community detection include Louvain [2], Infomap [21], label propagation [20], spectral optimization-based [18], hierarchical [14], minimum-cut-based [6], and betweenness-score-based [17] methods. These methods can find communities with high quality, but often require expensive computational work [24]. Serial implementation of these algorithms can take prohibitively long to find communities in large networks. The parallel computing paradigm enables the processing of massive graphs quickly. Multiple processors can be used to distribute the computational load among them and reduce the total runtime. However, it is challenging to parallelize the community detection algorithm due to irregular communication patterns and computational dependencies. Communications among processors can impede the efficiency and scalability of the algorithm. Hence, it is essential to develop efficient communication strategies.

In this paper, we propose label-propagation-based parallel community detection algorithms for distributed-memory systems. Minimizing communication and achieving good load balancing is crucial to ensuring the scalability of a parallel algorithm. To reduce communication and computation, we have employed several novel optimizations, which resulted in efficient and scalable algorithms. We also studied various load-balancing schemes. The presented algorithm is implemented using Message Passing Interface (MPI) primitives. Additionally, we extended our algorithm to hybrid parallel systems using MPI and OpenMP. The hybrid parallel algorithm combines the advantages of distributed and shared-memory systems, improving speedup and scalability even further. Our main contributions are summarized below.

1. We developed a distributed-memory parallel algorithm based on label propagation. We employed efficient communication strategies to improve performance.
2. We devised novel techniques, which eliminate redundancy in computation and reduce the runtime significantly.
3. We experimented with various communication routines and load-balancing schemes and presented experimental results comparing them.
4. Finally, we present a hybrid algorithm that effectively utilizes the advantages of both shared and distributed memory systems and demonstrates significantly improved performance and scalability.

For the community detection problem, there are a few existing distributed-memory parallel algorithms, which are briefly discussed in Section 2.3. Our algorithms are significantly faster and more scalable than the state-of-the-art algorithms such as DistLouvain [8] and DDOLP [12]. For instance, on the Twitter graph with 61M (million) nodes and 1.9B (billion) edges, our hybrid algorithm runs in around 120 seconds using 4000 processors, whereas the DistLouvain algorithm takes 600 seconds using the same number of processors. Our hybrid algorithm scales to more than 4000 processors, whereas DDOLP scales up to only one hundred processors. Our parallel algorithms are capable of working with very large graphs. Our algorithm was successfully executed on a massive Metaclust graph that comprised 282 million nodes and 42 billion edges in 654 seconds using 4096 processors. In contrast, the largest graphs that were tested with DistLouvain and DDOLP were UK graphs with 100 million nodes and 3.3 billion edges and a synthetic graph with 20 million nodes and 400 million edges, respectively.

The remainder of the paper is organized as follows. Some preliminary concepts and related work are discussed in Section 2. Our parallel algorithms are presented in Section 3, and the experimental results are given in Section 4.

2 Background

2.1 Preliminaries

Let $G = (V, E)$ be a graph where V is the set of vertices and E is the set of edges. If $(u, v) \in E$, u and v are called neighbors of each other. N_v denotes the set of neighbors of vertex v; that is, $N_v = \{u : (u, v) \in E\}$ and $d_v = |N_v|$ denotes the degree of v. Communities can be defined as groups of vertices having dense connections within each group but relatively sparse connections between the groups. The communities form a partition of the vertex set V: $P = \{C_1, C_2, \ldots, C_k\}$, where the community $C_i \subset V$ and $C_i \cap C_j = \phi$ for $i \neq j$.

We measure and compare the quality of communities using the normalized mutual information (NMI) metric, which is defined as follows. Suppose that we have two partitions, $P = \{P_1, P_2, \ldots, P_k\}$ and $Q = \{Q_1, Q_2, \ldots, Q_k\}$. The NMI between P and Q is defined as:

$$\text{NMI}(P, Q) = \frac{2 \times \text{I}(P, Q)}{H(P) + H(Q)},$$

where $H()$ represents entropy, and $I()$ represents mutual information between the partitions. $\text{I}(P, Q)$ is given by:

$$\text{I}(P, Q) = \sum_{i=1}^{|P|} \sum_{j=1}^{|Q|} \frac{|P_i \cap Q_j|}{n} \log \frac{n|P_i \cap Q_j|}{|P_i||Q_j|},$$

and $H(P)$ is given by:

$$H(P) = -\sum_{i=1}^{|P|} \frac{|P_i|}{n} \log \frac{|P_i|}{n}.$$

NMI is commonly used to evaluate community detection algorithms by comparing detected communities with ground truth communities.

2.2 Sequential Algorithms

Community detection algorithms can be classified as divisive or agglomerative algorithms. Divisive algorithms start with the entire graph as one community and progressively split the communities based on some measures such as internal degrees [5] and statistical inference [11]. Agglomerative algorithms start with singleton communities (communities with only one vertex) and gradually merge communities using similarity measures such as the Jaccard index, cosine similarity, and clustering coefficient [24]. One well known approach to community detection is modularity maximization. Modularity maximization algorithms start with some initial community assignments and repeatedly update the communities in multiple steps. At each step, communities are updated such that the modularity gain is maximized [4, 14]. The Louvain algorithm [2] is a state-of-the-art modularity optimization algorithm for community detection. In contrast to the Louvain method, which optimizes a global modularity score, the label propagation method [20] uses only local information (the labels of the immediate neighbors) for each node and does not require optimizing any global quality score. Algorithm 1 provides the pseudocode of the label propagation method [20]. It starts by assigning a unique community label to each vertex. At this point, each vertex is a separate singleton community. In each subsequent iteration, these labels are updated based on some consensus or selection rule. A common consensus rule is that every vertex is assigned a label that is the majority among the labels of its neighbors, with ties broken arbitrarily. Raghavan et al. [20] reported that for the most large real-world networks, after only five iterations, the labels of 95% of the nodes converged to their final values.

2.3 Related Parallel Algorithms

Most of the previous parallel algorithms for community detection are for shared-memory systems. There are only a few algorithms for distributed-memory systems. Distributed direction optimizing label propagation (DDOLP) [12] is a distributed-memory parallel algorithm based on label propagation. It propagates a selected subset of vertices called seed vertices in two execution modes, *push* operation and *pull* operation. Push operation is a blind propagation step in which the seed vertices propagate their labels without any resistance. The pull

Algorithm 1 Label Propagation Algorithm

```
1: for all $v \in V$ do
2:     $L(v) \leftarrow v$
3: $Converged \leftarrow false$
4: while not $Converged$ do
5:     $Converged \leftarrow true$
6:     for all $v \in V$ do
7:         $L'(v) \leftarrow majority\ label$
8:     for all $v \in V$ do
9:         if $L'(v) \neq L(v)$ then
10:            $Converged \leftarrow false$
11:            $L(v) \leftarrow L'(v)$
12: end while
```

operation propagates labels according to a consensus (similar to the label propagation algorithm). By selectively propagating seed labels, DDOLP artificially initializes core communities, interfering with the detection of natural community cores. These drawbacks make DDOLP unsuitable for applications that need to detect naturally-formed communities. DDOLP was implemented in AM++, and hence we are unable to compare it with our algorithm directly. DDOLP was evaluated on synthetic graphs [12], but not on real-world graphs. Real-world graphs present unique challenges for load-balancing [22]; our algorithm uses a suitable graph partitioning scheme to overcome this problem. The scalability of DDOLP is limited to less than 100 processors [12] on large graphs like uk-2007 (with 105M vertices and 3.3B edges). In contrast, our algorithm scales to thousands of processors, as shown in Section 4.

DistLouvain [8] is a state-of-the-art distributed-memory algorithm, which is a parallelization of the modularity-maximizing Louvain algorithm [2]. DistLouvain scales to thousands of processors. We extensively compare our algorithm with DistLouvein and show that our algorithm outperforms DistLouvain in both runtime and scalability (see Section 4). Distributed Parallel Louvain Algorithm with Load-balancing (DPLAL) [22] is another algorithm for community detection. DPLAL uses efficient METIS graph partitioning [9] to address load imbalances in real-world graphs. DPLAL code is not publicly available; hence we are not able to directly compare with them. DPLAL is shown to scale up to 512 processors on the orkut graph (3M vertices and 117M edges), whereas our algorithm scales up to 4096 processors and is capable of processing much larger graphs like MetaClust50 (282.2M vertices and 42.8B edges).

3 Our Parallel Algorithms

In this section, we present our parallel algorithms for detecting communities using the label propagation method. First, we present our algorithm for distributed-memory systems. We then discuss how this algorithm can be extended for hybrid systems. We present a few novel optimization techniques to reduce the number of messages and local computation (computation done locally by each processor). These techniques significantly improve the efficiency and scalability of the algorithm. The hybrid extension of our algorithm, together with the optimization techniques, makes it highly efficient and scalable to thousands of processors.

We introduce some notation and definitions that are used to describe our algorithm. The graph $G = (V, E)$ is partitioned and distributed among the processors so that the processor P_i stores the vertices V_i and their corresponding set of edges E_i. Each $v \in V_i$ is said to be local to the processor P_i, and P_i is the owner of v. For any edge $(u, v) \in E$, if the same processor owns both u and v, u is called a *local neighbor* of v; otherwise, u is a *ghost neighbor* of v. Let N_v be the set of all neighbors of v and $L(v)$ the label of v.

For each vertex $v \in V_i$, we use a list of labels of the neighbors of v. Let NL_v be the multiset containing the labels of the vertices in N_v. A label can appear multiple times in NL_v since different neighbors of v can have the same label.

Multiset NL_v can be implemented using an array of size d_v. Thus, the total space required for the NL_v multisets for all $v \in V_i$ is $O(|V_i| + |E_i|)$.

Initially, each vertex is assigned its id as its label. The processor P_i initializes $L(v) = v$ and $NL_v = N_v$ for each $v \in V_i$. Then, the algorithm proceeds in synchronized iterations; that is, a processor continues to the next iteration only after all processors complete their current iteration. In each iteration, a processor executes three major steps: a) update the labels of the vertices, b) check for convergence of the labels, and c) exchange the labels with the other processors. The details of each of these steps are described below.

a) Computing the majority label: Array NL_v contains the labels of the neighbors of v. For each $v \in V_i$, processor P_i finds the most frequent label, called the majority label, in NL_v and assigns it to $L(v)$. The majority label in NL_v can be computed by sorting the elements in NL_v and then performing a linear scan and keeping track of the most frequent item. For each v, it will take $O(d_v \log d_v)$ time, and thus the processor P_i takes $O\left(\sum_{v \in V_i} d_v \log d_v\right)$ time for this step in each iteration.

b) Checking convergence: The labels are converged when none of the vertices changes its label in an iteration. Each processor can locally check whether the label of any of its vertices has been changed. The processors can then exchange this information with each other to find whether any vertex in the entire set V has changed its label. If a processor exchanges individual messages with the other processors, it will take $O(p)$ time. However, it can be done more efficiently using a parallel collective reduction operation in $O(\log n)$ time.

c) Exchanging labels and updating NL_v: As described above, in each iteration for each v, a new label is computed using the labels in NL_v, the labels of the neighbors of v from the preceding iteration. Thus, at the end of the current iteration, and if it is not the last iteration, the old labels are removed from NL_v. The newly updated labels are added to NL_v to prepare it for the next iteration. Since other processors can own some of the neighbors of a vertex, the processors need to exchange the labels of the vertices with each other. A naive way to do this is: each processor P_i sends the labels of all $v \in V_i$ to all other processors and receives the labels of the other vertices from them. However, it is not necessary to send the labels of all $v \in V_i$ to all other processors. The label $L(v)$ should be sent only to the owners of the neighbors of v. For each neighbor $u \in N_v$, if u is a local neighbor, $L(v)$ is added to NL_u; otherwise, P_i sends $L(v)$ to the owner of u. The owner of u adds $L(v)$ to NL_u after receiving the message from P_i. Note that if a processor P_i needs to send multiple labels to another processor P_j, sending a single message containing all of these labels is more efficient than sending multiple messages, each containing a single label. In the actual implementation of our algorithm, the labels destined for the same processor are combined in a buffer, and the contents of the buffer are sent in a single message. Thus, in each iteration, a processor sends at most one message to any other processor. The pseudocode of our algorithm is given in Algorithm 2. In the next section, we describe some more techniques to further optimize communication and local computation.

Algorithm 2 Parallel Label Propagation Algorithm

```
1:  for all $v \in V$ do in parallel
2:      $L(v) \leftarrow v$
3:      $NL_v \leftarrow N_v$
4:  $Converged \leftarrow false$
5:  while not $Converged$ do
6:      $Converged \leftarrow true$
7:      for all $v \in V$ do in parallel
8:          $L'(v) \leftarrow$ majority label in $NL_v$
9:          $NL_v \leftarrow \phi$
10:         if $L'(v) \neq L(v)$ then
11:             $Converged \leftarrow false$
12:     Parallel_Reduction($Converged$)
13:     $Converged \leftarrow \bigwedge_{i=0}^{p-1} Converged_i$
14:     if not $Converged$ then
15:         for all $v \in V$ do in parallel
16:             $L(v) \leftarrow L'(v)$
17:             for all $u \in N_v$ do
18:                 if $u$ is a ghost neighbor
19:                     Send $(L(v), u)$ to the owner of $u$
20:                 else
21:                     $L(v)$ to $NL_u$
22:         Receive the labels sent by other processors
23:         for all received labels $(L(v), u)$
24:             Add $L(v)$ to $NL_u$
25: end while
```

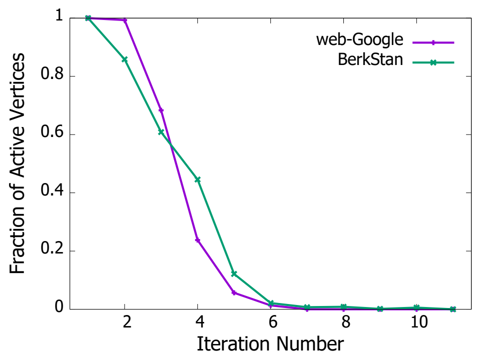


Fig. 1: The figure on the left shows the number of active vertices corresponding to the number of iterations for two graphs: Web-Google and BerkStan, each with a few million edges. The number of active vertices is significantly reduced with the progress of the algorithm

3.1 Techniques to Further Optimize Communication and Computation

A major challenge in making the distributed-memory parallel algorithm scalable to a large number of processors is the large number of messages exchanged by the processors. To reduce communication among processors, we present some techniques that also reduce local computation in addition to reducing communications.

For most graphs, the labels change only in the first few iterations. In later iterations, the cores of many communities have already formed and do not change their labels. Only the vertices on the periphery of the community cores change labels and eventually converge to a final solution [12, 13]. If $L'(v) \neq L(v)$ in an iteration, the vertex v is called *active* in that iteration; otherwise, v is *inactive*. Figure 1 shows the number of active vertices with the number of iterations for two graphs. The number of active vertices is quickly reduced with the progress of the algorithms. Therefore, exchanging only the updated labels can significantly reduce the communication cost, especially in later iterations. In the next optimized version of our algorithm, called the *active version*, the processors exchange the labels of only the active vertices. Processor $P(v)$ sends $L'(v)$ to $P(u)$ in an iteration if and only if $L'(v) \neq L(v)$ in that iteration. $P(v)$ also sends the old label $L(v)$ along with the new label $L'(v)$ so that $P(u)$ can update NL_u by replacing $L(v)$ with $L'(v)$.

In the implementation of NL_u, it is more efficient to store the counts of the distinct labels instead of storing all occurrences of the labels. For each distinct label, we have only one entry where each entry is a pair: a label and the count or frequency of the label. To store these (label, count) pairs, we use two alternative data structures: HashMap and MaxHeap. For each u, processor $P(u)$ maintains one such data structure: HashMap(u) or MaxHeap(u). When $P(u)$ receives $L(v)$ and $L'(v)$, the previous and current labels of v, the count for $L(v)$ is decremented by 1, and the count for $L'(v)$ is increased by 1. If the count of a label becomes 0 after decrementing, its entry is removed from the HashMap (or MaxHeap). Some additional details and analyses of these two data structures are described below. Let ℓ_u be the number of distinct labels in HashMap(u) (or MaxHeap(u)) and t_u the number of update operations (increments and decrements) performed on HashMap(u) (or MaxHeap(u)) in an iteration. Note that in any iteration, both ℓ_u and t_u are, at most, d_u, the degree of u.

HashMap: We use the HashMap data structure implemented in the Standard Template Library (STL). Each update operation takes $O(\log \ell_u)$ time, and t_u updates take $O(t_u \log \ell_u)$ time. To find a label with the maximum count from HashMap(u), we iterate through the entries in HashMap(u), which takes $O(\ell_u)$ time. Thus, in one iteration, the total time to update HashMap(u) and find the majority label is $O(\ell_u + t_u \log \ell_u)$, whereas the array-based implementation of NL_u in the base algorithm takes $O(d_u \log d_u)$ time. Since both ℓ_u and t_u can be significantly smaller than d_u, especially in the later iterations, the use of HashMap with the active version of our algorithm not only reduces communications, it also reduces local computation in finding the majority labels for the local vertices.

MaxHeap: We use our own custom implementation of the MaxHeap data structure. We need to store (label, count) pairs in a MaxHeap. While the items in the MaxHeap is ordered by the counts, we need to search the items using the labels. To facilitate such search operations efficiently, we use a mapping from the labels to the indices of the MaxHeap using an STL Map data structure. Like HashMap, each update operation in MaxHeap(u) takes $O(\log \ell_u)$ time, and t_u update operations take $O(t_u \log \ell_u)$ time. Finding a label with the maximum count is done in $O(1)$ time since a MaxHeap always keeps the largest element at the root of the MaxHeap. As a result, the runtime performance of the MaxHeap version is a little better than that of the HashMap version.

3.2 Comparison of Alternative Communication Options in MPI

We implement our algorithm using Message Passing Interface (MPI) primitives. As required in our algorithm, the communication of the labels can be done using MPI in a few different ways, as explained below. **Ordered-Recv and Dynamic-Recv schemes:** Each processor sends and receives the labels to and from the other $p-1$ processors. To exchange these messages, we utilize MPI_Send() and MPI_Recv(), which are point-to-point communication routines. The processor P_i sends a message with updated labels to another processor P_j by calling MPI_Send(). The processor P_j receives the message by calling a matching MPI_Recv() operation. We present two schemes, *Ordered-Recv* and *Dynamic-Recv*, to receive the messages by using the MPI_Recv() function. In the *Ordered-Recv* scheme, we receive messages in a queue ordered by processor IDs (or MPI ranks). The processor P_i first receives the message from P_0, then $P_1 \dots P_{i-1}, P_{i+1} \dots P_{p-1}$ in that order.

In the *Dynamic-Recv* scheme, we receive the arriving messages on a first-come-first-serve basis. MPI_PROBE() function is used to find the source and size of the next message, and then the message is received by calling MPI_Recv() without following any order of the source processors.

A2Av scheme: In the *A2Av* scheme, we exchange labels by using collective communication routines like MPI_Alltoall() and MPI_Alltoallv(). Every processor participates in this operation simultaneously and exchanges messages with every other processor. In our algorithm, the messages sent are not of uniform size, and hence we use the irregular collective method MPI_Alltoallv(). We first perform an MPI_Alltoall() operation to exchange send and receive counts (the size of the messages to be exchanged), which is needed for the collective communication function MPI_Alltoallv(). Then, all processors execute MPI_Alltoallv() simultaneously in a synchronous fashion, which causes the exchange of the label data between all pairs of processors.

3.3 Hybrid Algorithm

Hybrid parallel algorithms utilize shared-memory parallelism by using concurrent access to a common memory and distributed-memory parallelism by using communication. To improve the scalability of our algorithm, we implement a hybrid parallel version using both MPI (for distributed-memory) and OpenMP primitives (for shared-memory). Consider a hybrid system with C compute nodes and t cores in each node; we have a total of $p = tC$ cores or processors. The input graph is divided into C partitions and partition $G_i(V_i, E_i)$ is assigned to the ith compute node. The algorithm uses C MPI tasks, one MPI task in each node. Then within each MPI task, t threads are created. The threads in the ith compute node share the data in partition $G_i(V_i, E_i)$ and other data structures in the ith compute node. The local computation of each node is further parallelized by dividing the computation tasks among the t threads. The initialization and computing of the majority of labels are divided among the threads. Communication among the compute nodes is done by the master thread in each compute node. This hybrid algorithm significantly improves the runtime and scalability of our algorithm.

3.4 Load Balancing

We study various ways of partitioning the vertex set V among processors.

Contiguous Vertex Partitioning Scheme (CONT): We assign consecutive vertices in an equal number to each partition such that each partition has approximately $\frac{|V|}{p}$ vertices. The first processor P_0 contains vertices $V_0 = \{v_0, v_1, \ldots, v_{\frac{|V|}{p}}\}$, the second processor P_1 contains $V_1 = \{v_{\frac{|V|}{p}+1}, v_{\frac{|V|}{p}+2}, \ldots, v_{\frac{2|V|}{p}}\}$ and so on. In the general case, processor P_i contains vertices $V_i = \{v_x, v_{x+1}, v_{x+2}, \ldots, v_{x+\frac{|V|}{p}-1}\}$, where $x = (i-1)\frac{|V|}{p}$.

Round Robin Vertex Partitioning Scheme (RR): Similar to CONT scheme, each processor holds approximately $\frac{|V|}{p}$ vertices. The first processor P_0 contains vertices $V_0 = \{v_0, v_{\frac{|V|}{p}}, v_{2\frac{|V|}{p}}, \ldots\}$, the second processor P_1 contains $V_1 = \{v_1, v_{\frac{|V|}{p}+1}, v_{2\frac{|V|}{p}+1}, \ldots\}$ and so on. In the general case, processor P_i contains vertices $V_i = \{v_i, v_{i+1\frac{|V|}{p}}, v_{i+2\frac{|V|}{p}}, \ldots\}$.

Alternating Round Robin Scheme (ARR): We also study a modification of RR, which we call an alternating round-robin scheme. In alternative steps, the vertices are assigned to the partitions in normal and reverse round-robin orders as follows. The first p vertices are assigned in normal order: vertex i, $0 \leq i < p$, is assigned to partition V_i. The next p vertices are assigned in reverse order: vertex i, $p \leq i < 2p$, is assigned to partition V_{2p-i-1}. Then, the next p vertices are assigned in normal order, and so on.

Equal Edge Count Scheme (EqEdg): In this scheme, like CONT, a consecutive set of vertices is assigned to each partition, but the number of vertices in the partitions may not be equal. The number of vertices is chosen so that all partitions have a nearly equal number of edges.

4 Experimental Results

In this section, we evaluate the performance of our parallel algorithm using a wide range of real-world graphs of various sizes and types. We conducted extensive experiments to compare the runtime and scalability of our algorithm with the state-of-the-art DistLouvain algorithm [8]. To be fair, both DistLouvain and our algorithm were run on the same parallel computing machines with exactly the same configuration. We also present experimental results showing the improvements made by the optimizations described in Section 3. A discussion comparing our algorithm with two other recent parallel algorithms DDOLP [12] and DPLAL [22] is given in Section 2.3.

4.1 Experimental Setup and Datasets

We used the Expanse Computing Cluster at the San Diego Supercomputing Center available to us through the ACCESS system [19]. The cluster houses AMD EPYC 7742 (Rome) computing nodes (with 128 CPU cores, 2.25 GHz). We also used the Bridges-II supercomputer at the Pittsburg Supercomputing Center (PSC), which shares a processor architecture similar to that of the Expanse system. We implemented our algorithms using the Message Passing Interface (MPI) library for distributed-memory systems and further used OpenMP and MPI for the hybrid version. We used the OpenMPI implementation of the MPI standard and compiled our codes using GCC version 9.2.0 with the -fopenmp -O3 flags. The data set is chosen to represent a wide variety of networks. Most of these

Table 1: List of graphs used in experiments.

Graph name	Nodes	Edges	Avg-Deg
BerkStan	0.7M	7.6M	21.7
web-Google	0.9M	5.1M	11.3
facebook	4.0M	24.0M	12.0
LiveJournal	3.9M	34.0M	17.4
Orkut	3.0M	117.0M	78.0
FriendSter	65.6M	1.8B	55.3
twitter	61.6M	1.9B	61.7
uk2007	105M	3.3B	62.8
Metaclust50	282.2M	42.8B	303.3

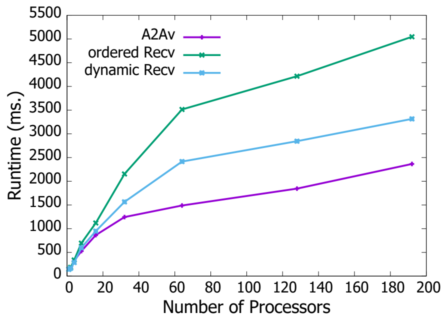
networks are taken from the Stanford Large Network Data Set Collection [10]. Table 1 lists the graphs used in our experiments.

4.2 Performance of Various MPI Communication Schemes

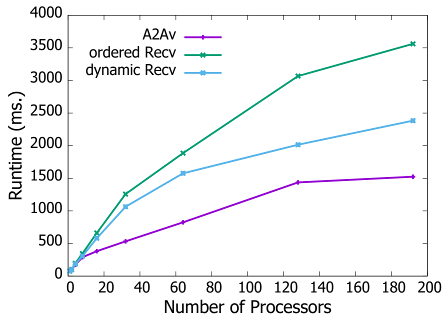
Fig. 2a and 2b compare the performances of the communication schemes, *Ordered-Recv*, *Dynamic-Recv*, and *A2Av*, disucssed in Section 3.2. The figures show the runtime for only the communication step of our algorithm. *A2Av* performs significantly better than the other two schemes. Between the other two schemes, *Ordered-Recv* is simpler, but *Dynamic-Recv* performs better. *Ordered-Recv* imposes an order in which messages can be received. This ordering requirement causes delay in receiving some of the messages even if they are available in the buffer. In the rest of the experiments, we use the *A2Av* scheme for MPI communication. Notice that the runtime of the communication step of the algorithm increases with the number of processors since more processors require more communications among them.

4.3 Effect of the Load Balancing Schemes

Fig. 3a and 3b show the rumtime of the algorithm with different load schemes: *CONT*, *RR*, *ARR*, *EqEdg* (described in Section 3.4). The results are shown for

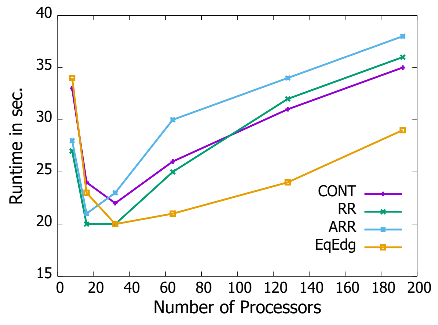


(a) Facebook network

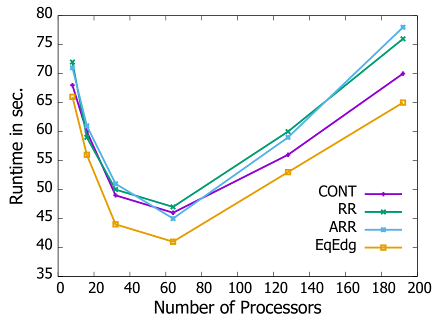


(b) BerkStan network

Fig. 2: Runtime of communication step for various communication schemes



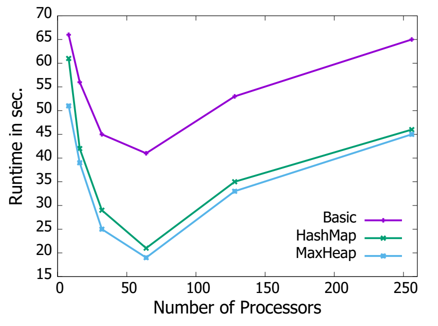
(a) Facebook network



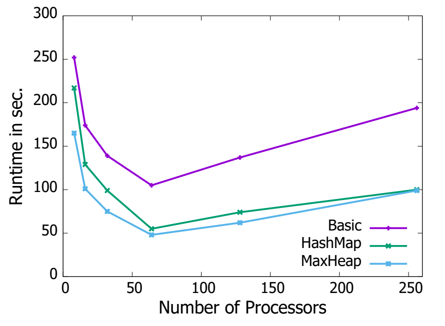
(b) LiveJournal network

Fig. 3: Runtime of communication step for various load balancing schemes

two networks: Facebook and LiveJournal. We observe that the *EqEdg* scheme outperforms the other schemes. The primary communication step of our algorithm exchanges the labels of the neighbors of the vertices. Further, the main computation also involves iterating through the neighbors of the vertices. Therefore, partitioning the graph with equal number of edges in the partitions ensures that the communication and computation load in the processors is almost equal.



(a) LiveJournal

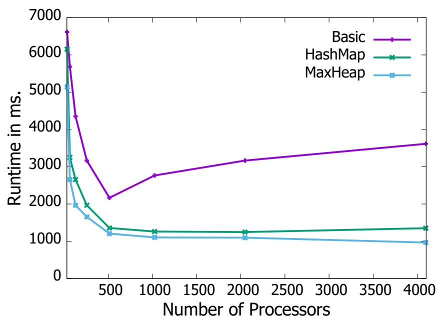


(b) Orkut

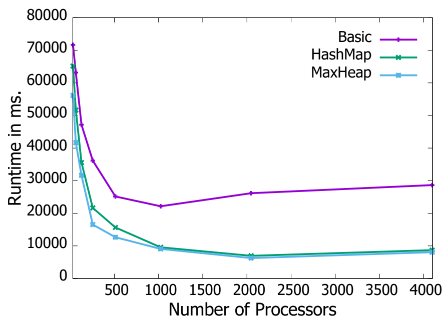
Fig. 4: Performance improvement due to various optimizations for the pure MPI algorithm

4.4 Optimizations

We also experimentally evaluate the additional optimizations discussed in Section 3.1 by comparing the following variants: *i*) *Base*: the baseline version of our algorithm, *ii*) *HashMap*: the active version of our algorithm with the HashMap



(a) LiveJournal



(b) Orkut

Fig. 5: Performance improvement due to various optimizations for the hybrid algorithm

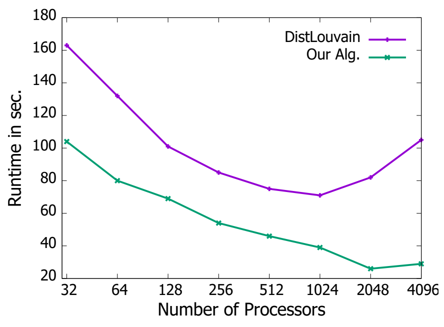
data structure, and *iii*) *MaxHeap*: the active version of our algorithm with the MaxHeap data structure.

We show the improvements due to these optimizations in Fig. 4a and 4b for the LiveJournal and Orkut networks, respectively. We observe a huge improvement in the *HashMap* and *MaxHeap* versions. These optimizations reduce communication and computation costs significantly. We further demonstrate the improvements by these optimizations for the hybrid algorithm in Fig. 5a and 5b.

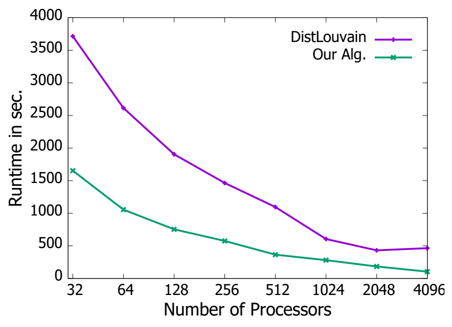
4.5 Hybrid Algorithm

In the hybrid algorithm, we used 4 MPI tasks per node and 32 OpenMP threads per MPI task for our experiments. In order to assess the performance of our algorithm, we compare our algorithm with the state-of-the-art DistLouvain algorithm, which is also a hybrid algorithm. Our experiments used an identical hybrid configuration for both DistLouvain and our algorithm.

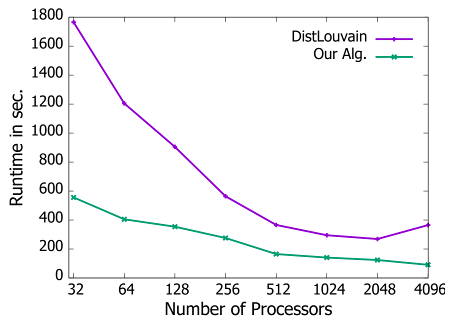
We compared these two algorithms using five large networks, including Metaclust50, a network with 42B edges. For all of these networks, our algorithm consistently outperforms DistLouvain and is significantly faster and more scalable than DistLouvain, as shown in Fig. 6a–6d and Fig. 7. Fig. 7 shows the runtime for a very large network, Metaclust50, with 282M vertices and 42.8B edges using 2048 and 4096 processors. For this network, the algorithms could not be run using a smaller number of processors. With a smaller number of processors, each partition is too large to fit in the memory of a compute node. While DistLouvain scales up to 1,000 or 2,000 processors for most of these graphs, our algorithms continue to scale to more than 4,000 processors. We could not experiment with more than 4,096 processors since we did not have access to any larger number of processors. However, from the trend, it is clear that our algorithm would scale far beyond 4,000 processors.



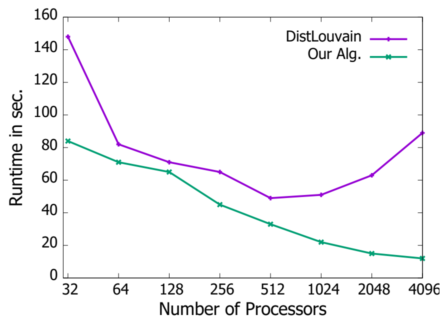
(a) sk-2005



(b) FriendSter



(c) twitter



(d) uk-2007

Fig. 6: Runtime of DistLouvain and our hybrid algorithm on various networks

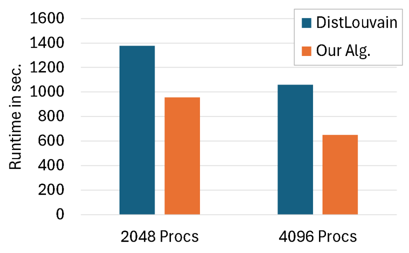


Fig. 7: Runtime of DistLouvain and our hybrid algorithm on a very large network, Metaclust50, with 42.8B edges using 2048 and 4096 processors

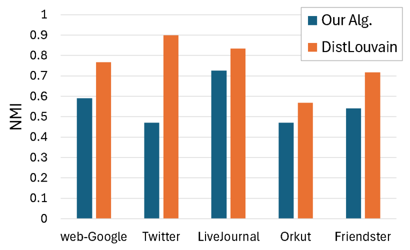


Fig. 8: Normalized mutual information (NMI) between the detected communities and ground truth (GT) communities

4.6 Quality of Communities

To compare detected communities with ground truth (GT) communities, we use normalized mutual information (NMI), which is described in Section 2.1. The NMI scores of DistLouvain and our algorithm for 5 networks are shown in Fig.

8. We obtained the ground truth communities for these networks from the data provided by Yang and Leskovec [23]. Note that the quality of the detected communities is not an artifact of these parallel algorithms. Instead, the quality is determined by the underlying serial algorithm, which is the Louvain algorithm for DistLouvain and the LPA for our parallel algorithm. Nevertheless, we are showing the quality of the detected communities to demonstrate the trade-off between quality and runtime. It is expected and known that the Louvain algorithm produces communities with better quality than LPA. However, for most of these networks, the quality of communities detected by our algorithm is comparable to that of DistLouvain.

5 Conclusion

We have developed an efficient and scalable algorithm to detect communities in large-scale graphs. This algorithm includes several non-trivial optimization techniques and data structures that have significantly improved its performance, surpassing the performance of the state-of-the-art algorithm DistLouvain. Our algorithm can work with massive graphs with billions of edges. Experiments on a wide range of real-world networks have shown that our algorithm can scale to more than 4000 processors consistently. With this capability and performance, our parallel algorithm can help analysts and researchers utilize modern commodity computing clusters, which have a large number of processors, to analyze large-scale networks quickly.

Acknowledgements

The work was partially supported by NSF BIGDATA grant IIS-1633028 and used PSC Bridges-2 RM at Pittsburg Supercomuting Center (PSC) through allocation CIS230052 from the Advanced Cyberinfrastructure Coordination Ecosystem: Services & Support (ACCESS) program [19], which is supported by NSF grants #2138259, #2138286, #2138307, #2137603, and #2138296.

References

1. Bedi, P., Sharma, C.: Community detection in social networks. Wiley interdisciplinary reviews: Data mining and knowledge discovery **6**(3), 115–135 (2016)
2. Blondel, V., Guillaume, J., Lambiotte, R., Lefebvre, E.: Fast unfolding of communities in large networks. Journal of Statistical Mechanics: Theory and Experiment **2008**(10), P10008 (2008)
3. Claude, F., Navarro, G.: Fast and compact web graph representations. ACM Trans. on the Web (TWEB) **4**(4), 1–31 (2010)
4. Clauset, A., Newman, M., Moore, C.: Finding community structure in very large networks. Physical Review E **70**(6), 066111 (2004)
5. Fortunato, S.: Community detection in graphs. Physics Reports **486**(3-5), 75–174 (2010)

6. Fortunato, S., Barthelemy, M.: Resolution limit in community detection. Proc. of the Nat. Academy of Sciences **104**(1), 36–41 (2007)
7. Fortunato, S., Hric, D.: Community detection in networks: A user guide. Physics Reports **659**, 1–44 (2016)
8. Ghosh, S., Halappanavar, M., Tumeo, A., Kalyanaraman, A., Lu, H., et al.: Distributed louvain algorithm for graph community detection. In: 2018 IEEE Int. Parallel and Dist. Proc. Symposium (IPDPS). pp. 885–895. IEEE (2018)
9. Karypis, G., Kumar, V.: Multilevel k-way partitioning scheme for irregular graphs. J. of Parallel and Dist. Comp. **48**(1), 96–129 (1998)
10. Leskovec, J., Krevl, A.: SNAP Datasets: Stanford large network dataset collection. http://snap.stanford.edu/data (Jun 2014)
11. Leskovec, J., Lang, K., Mahoney, M.: Empirical comparison of algorithms for network community detection. In: Proc. of the 19th Int. Conference on World Wide Web. pp. 631–640 (2010)
12. Liu, X., Firoz, J., Zalewski, M., Halappanavar, M., Barker, K., Lumsdaine, A., Gebremedhin, A.: Distributed direction-optimizing label propagation for community detection. In: 2019 IEEE High-Performance Extreme Comp. Conf. (HPEC). pp. 1–6. IEEE (2019)
13. Liu, X., Halappanavar, M., Barker, K., Lumsdaine, A., Gebremedhin, A.: Direction-optimizing label propagation and its application to community detection. In: Proc. of the 17th ACM Int. Conf. on Comp. Frontiers. pp. 192–201 (2020)
14. Malliaros, F., Vazirgiannis, M.: Clustering and community detection in directed networks: A survey. Physics Reports **533**(4), 95–142 (2013)
15. Manipur, I., Giordano, M., Piccirillo, M., Parashuraman, S., Maddalena, L.: Community detection in protein-protein interaction networks and applications. IEEE/ACM Trans. on Comp. Biology and Bioinformatics **20**(1), 217–237 (2021)
16. Newman, M.: Detecting community structure in networks. The European Physical Journal B **38**(2), 321–330 (2004)
17. Newman, M.: Modularity and community structure in networks. Proc. of the National Academy of Sciences **103**(23), 8577–8582 (2006)
18. Newman, M.: Spectral methods for community detection and graph partitioning. Physical Review E **88**(4), 042822 (2013)
19. Parashar, M., Friedlander, A., Gianchandani, E., Martonosi, M.: Transforming science through cyberinfrastructure. Communications of the ACM **65**(8), 30–32 (2022)
20. Raghavan, U., Albert, R., Kumara, S.: Near linear time algorithm to detect community structures in large-scale networks. Physical Review E **76**(3), 036106 (2007)
21. Rosvall, M., Axelsson, D., Bergstrom, C.: The map equation. The European Physical Journal Special Topics **178**(1), 13–23 (2009)
22. Sattar, N., Arifuzzaman, A.: Scalable distributed louvain algorithm for community detection in large graphs. The J. of Supercomputing **78**(7), 10275–10309 (2022)
23. Yang, J., Leskovec, J.: Defining and evaluating network communities based on ground-truth. In: Proc. of the ACM SIGKDD Workshop on Mining Data Semantics. pp. 1–8 (2012)
24. Yang, Z., Algesheimer, R., Tessone, C.: A comparative analysis of community detection algorithms on artificial networks. Scientific Reports **6**, 30750 (2016)