Fast Parallel Conversion of Edge List to Adjacency List for Large-Scale Graphs

Shaikh Arifuzzaman[†‡] **and Maleq Khan**[‡]
[†]Department of Computer Science
[‡]Network Dynamics & Simulation Science Lab, Virginia Bioinformatics Institute
Virginia Tech, Blacksburg, VA 24061, USA
{sm10, maleq}@vbi.vt.edu

ABSTRACT

In the era of Bigdata, we are deluged with massive graph data emerged from numerous social and scientific applications. In most cases, graph data are generated as lists of edges (*edge list*), where an edge denotes a link between a pair of entities. However, most of the graph algorithms work efficiently when information of the adjacent nodes (*adjacency list*) for each node is readily available. Although the conversion from edge list to adjacency list can be trivially done on the fly for small graphs, such conversion becomes challenging for the emerging large-scale graphs consisting billions of nodes and edges. These graphs do not fit into the main memory of a single computing machine and thus require distributed-memory parallel or external-memory algorithms.

In this paper, we present efficient MPI-based distributed memory parallel algorithms for converting edge lists to adjacency lists. To the best of our knowledge, this is the first work on this problem. To address the critical load balancing issue, we present a parallel load balancing scheme which improves both time and space efficiency significantly. Our fast parallel algorithm works on massive graphs, achieves very good speedups, and scales to large number of processors. The algorithm can convert an edge list of a graph with 20 billion edges to the adjacency list in less than 2 minutes using 1024 processors. Denoting the number of nodes, edges and processors by n, m, and P, respectively, the time complexity of our algorithm is $O(\frac{m}{P} + n + P)$ which provides a speedup factor of at least $\Omega(\min\{P, d_{avg}\})$, where d_{avg} is the average degree of the nodes. The algorithm has a space complexity of $O(\frac{m}{P})$, which is optimal.

Author Keywords

Parallel Algorithms; Massive Graphs; Bigdata; Edge List; Adjacency List; Load Balancing.

ACM Classification Keywords

D.1.3 Programming Techniques: Concurrent Programming—*Parallel Programming*; G.2.2 Discrete Mathematics: Graph Theory—*Graph Algorithms*

HPC'15, April 12-15, 2015, Alexandria, VA.

INTRODUCTION

Graph (network) is a powerful abstraction for representing underlying relations in large unstructured datasets. Examples include the web graph [11], various social networks, e.g., Facebook, Twitter [17], collaboration networks [19], infrastructure networks (e.g., transportation networks, telephone networks) and biological networks [16].

We denote a graph by $G(V, E)$, where V and E are the set of vertices (nodes) and edges, respectively, with $m = |E|$ edges and $n = |V|$ vertices. In many cases, a graph is specified by simply listing the edges $(u, v), (v, w), \cdots \in E$, in an arbitrary order, which is called *edge list*. A graph can also be specified by a collection of adjacency lists of the nodes, where the *adjacency list* of a node v is the list of nodes that are adjacent to v. Many important graph algorithms, such as computing shortest path, breadth-first search, and depth-first search are executed by exploring the neighbors (adjacent nodes) of the nodes in the graph. As a result, these algorithms work efficiently when the input graph is given as adjacency lists. Although both edge list and adjacency list have a space requirement of $O(m)$, scanning all neighbors of node v in an edge list can take as much as $O(m)$ time compared to $O(d_v)$ time in adjacency list, where d_v is the degree of node v.

Adjacency matrix is another data structure used for graphs. Much of the earlier work [3, 15] use adjacency matrix $A[.,.]$ of order $n \times n$ for a graph with n nodes. Element $A[i, j]$ denotes whether node j is adjacent to node i. All adjacent nodes of i can be determined by scanning the i-th row, which takes $O(n)$ time compared to $O(d_i)$ time for adjacency list. Further, adjacency matrix has a prohibitive space requirement of $O(n^2)$ compared to $O(m)$ of adjacency list. In a real-world network, m can be much smaller than n^2 as the average degree of a node can be significantly smaller than n. Thus adjacency matrix is not suitable for the analysis of emerging large-scale networks in the age of bigdata.

In most cases, graphs are generated as list of edges since it is easier to capture pairwise interactions among entities in a system in arbitrary order than to capture all interactions of a single entity at the same time. Examples include captur-

ing person-person connection in social networks and protein-protein links in protein interaction networks. This is true even for generating massive random graphs [2, 14] which is useful for modeling very large system. As discussed by Leskovec et. al [18], some patterns only exist in massive datasets and they are fundamentally different from those in smaller datasets. While generating such massive random graphs, algorithms usually output edges one by one. Edges incident on a node v are not necessarily generated consecutively. Thus a conversion of edge list to adjacency list is necessary for analyzing these graphs efficiently.

Why do we need parallel algorithms? With unprecedented advancement of computing and data technology, we are deluged with massive data from diverse areas such as business and finance [5], computational biology [12] and social science [7]. The web has over 1 trillion webpages. Most of the social networks, such as, Twitter, Facebook, and MSN, have millions to billions of users [13]. These networks hardly fit in the memory of a single machine and thus require external memory or distributed memory parallel algorithms. Now external memory algorithms can be very I/O intensive leading to a large runtime. Efficient distributed memory parallel algorithms can solve both problems (runtime and space) by distributing computing tasks and data to multiple processors.

In a sequential settings with the graphs being small enough to be stored in the main memory, the problem of converting an edge list to adjacency list is trivial as described in the next section. However, the problem in a distributed-memory setting with massive graphs poses many non-trivial challenges. The neighbors of a particular node v might reside in multiple processors which need to be combined efficiently. Further, computation loads must be well-balanced among the processors to achieve a good performance of the parallel algorithm. Like many others, this problem demonstrates how a simple trivial problem can turn into a challenging problem when we are dealing with bigdata.

Contributions. In this paper, we study the problem of converting *edge list* to *adjacency list* for large-scale graphs. We present MPI-based distributed-memory parallel algorithms which work for both directed and undirected graphs. We devise a parallel load balancing scheme which balances the computation load very well and improves the efficiency of the algorithms significantly, both in terms of runtime and space requirement. Furthermore, we present two efficient merging schemes for combining neighbors of a node from different processors– *message-based* and *external-memory* merging– which offer a convenient trade-off between space and runtime. Our algorithms work on massive graphs, demonstrate very good speedups on both real and artificial graphs, and scale to a large number of processors. The edge list of a graph with $20B$ edges can be converted to adjacency list in two minutes using 1024 processors. We also provide rigorous theoretical analysis of the time and space complexity of our algorithms. The time and space complexity of our algorithms are $O(\frac{m}{P} + n + P)$ and $O(\frac{m}{P})$, respectively, where n, m, and P are the number of the nodes, edges, and processors, respectively. The speedup factor is at least $\Omega(\min\{P, d_{avg}\})$, where d_{avg} is the average degree of the nodes.

Organization. The rest of the paper is organized as follows. The preliminary concepts and background of the work are briefly described in *preliminaries and background* section. Our parallel algorithm along with the load balancing scheme is presented in *parallel algorithm* section. In section *dataset and experimental setup*, we present our dataset and specification of computing resources used for experiments. We discuss our experimental results in section *performance analysis*, and we conclude thereafter.

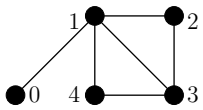
PRELIMINARIES AND BACKGROUND

In this section, we describe the basic definitions used in this paper and then present a sequential algorithm for converting edge list to adjacency list.

Basic definitions

We assume n vertices of the graph $G(V, E)$ are labeled as $0, 1, 2, \ldots, n-1$. We use the words *node* and *vertex* interchangeably. If $(u, v) \in E$, we say u and v are neighbors of each other. The set of all adjacent nodes (neighbors) of $v \in V$ is denoted by N_v, i.e., $N_v = \{u \in V | (u, v) \in E\}$. The degree of v is $d_v = |N_v|$.

In edge-list representation, edges $(u, v) \in E$ are listed one after another without any particular order. Edges incident to a particular node v are not necessarily listed together. On the other hand, in adjacency-list representation, for all v, adjacent nodes of v, N_v, are listed together. An example of these representations is shown in Figure 1.



a) Example Graph

$(0,1)$	$N_0 = \{1\}$
$(1,2)$	$N_1 = \{0,2,3,4\}$
$(1,3)$	$N_2 = \{1,3\}$
$(1,4)$	$N_3 = \{1,2,4\}$
$(2,3)$	$N_4 = \{1,3\}$
$(3,4)$	

b) Edege List c) Adjacency List

Figure 1: The edge list and adjacency list representations of an example graph with 5 nodes and 6 edges.

A Sequential Algorithm

The sequential algorithm for converting edge list to adjacency list works as follows. Create an empty list N_v for each node v, and then, for each edge $(u, v) \in E$, include u in N_v and v in N_u. The pseudocode of the sequential algorithm is given in Figure 2. For a directed graph, line 5 of the algorithm should be omitted since a directed edge (u, v) doesn't imply that there is also an edge (v, u). In our subsequent discussion, we assume that the graph is undirected. However, the algorithm also works for the directed graph with the mentioned modification.

This sequential algorithm is optimal since it takes $O(m)$ time to process $O(m)$ edges and thus can not be further improved. The algorithm has a space complexity of $O(m)$.

```
1: for each v ∈ V do
2:     N_v ← ∅
3: for each (u, v) ∈ E do
4:     N_u ← N_u ∪ {v}
5:     N_v ← N_v ∪ {u}
```

Figure 2: Sequential algorithm for converting edge list to adjacency list.

For small graphs that can be stored wholly in the main memory, the conversion in a sequential setting is trivial. However, emerging massive graphs pose many non-trivial challenges in terms of memory and execution efficiency. Such graphs might not fit in the local memory of a single computing node. Even if some of them fit in the main memory, the runtime might be prohibitively large. Efficient parallel algorithms can solve this problem by distributing computation and data among computing nodes. We present our parallel algorithm in the next section.

THE PARALLEL ALGORITHM

First we present the computational model and an overview of our parallel algorithm. A detailed description follows thereafter.

Computation Model

We develop parallel algorithms for message passing interface (MPI) based distributed-memory parallel systems, where each processor has its own local memory. The processors do not have any shared memory, one processor cannot directly access the local memory of another processor, and the processors communicate via exchanging messages using MPI constructs.

Overview of the Algorithm

Let P be the number of processor used in the computation and E be the list of edges given as input. Our algorithm has two phases of computation. In Phase 1, the edges E are partitioned into P *initial partitions* E_i, and each processor is assigned one such partition. Each processor then constructs neighbor lists from the edges of its own partition. However, edges incident to a particular node might reside in multiple processors, which creates multiple partial adjacency lists for the same node. In Phase 2 of our algorithms, such adjacency lists are merged together. Now, performing Phase 2 of the algorithm in a cost-effective way is very challenging. Further, computing loads among processors in both phases need to be balanced to achieve a significant runtime efficiency. The load balancing scheme should also make sure that space requirement among processors are also balanced so that large graphs can be processed. We describe the phases of our parallel algorithm in detail as follows.

(Phase 1) Local Processing

The algorithm partitions the set of edges E into P partitions E_i such that $E_i \subseteq E$, $\bigcup_k E_k = E$ for $0 \leq k \leq P-1$. Each partition E_i has almost $\frac{m}{P}$ edges– to be exact, $\lceil \frac{m}{P} \rceil$ edges, except for the last partition which has slightly fewer ($m - (p-1)\lceil \frac{m}{P} \rceil$). Processor i is assigned partition E_i. Processor i then constructs adjacency lists N_v^i for all nodes v such that $(.,v) \in E_i$ or $(v,.) \in E_i$. Note that adjacency list N_v^i is only a partial adjacency list since other partitions E_j might have edges incident on v. We call N_v^i *local adjacency list* of v in partition i. The pseudocode for Phase 1 computation is presented in Figure 3.

```
1: Each processor i, in parallel, executes the following.
2: for (u, v) ∈ E_i do
3:     N_v^i ← N_v^i ∪ {u}
4:     N_u^i ← N_u^i ∪ {v}
```

Figure 3: Algorithm for performing Phase 1 computation.

This phase of computation has both the runtime and space complexity of $O(\frac{m}{P})$ as shown in Lemma 1 .

LEMMA 1. *Phase 1 of our parallel algorithm has both the runtime and space complexity of* $O(\frac{m}{P})$.

Proof: Each initial partition i has $|E_i| = O(\frac{m}{P})$ edges. Executing Line 3-4 in Figure 3 for $O(\frac{m}{P})$ edges requires $O(\frac{m}{P})$ time. Now the total space required for storing local adjacency lists N_v^i in partition i is $2|E_i| = O(\frac{m}{P})$. □

Thus the computing loads and space requirements in Phase 1 are well-balanced. The second phase of our algorithm constructs the final adjacency list N_v from local adjacency lists N_v^i from all processors i. Note that balancing load for Phase 1 doesn't make load well balanced for Phase 2 which requires a more involved load balancing scheme as described later in the following sections.

(Phase 2) Merging Local Adjacency Lists

Once all processors complete constructing local adjacency lists N_v^i, final adjacency lists N_v are created by merging N_v^i from all processors i as follows.

$$N_v = \bigcup_{i=0}^{P-1} N_v^i \tag{1}$$

The scheme used for merging local adjacency lists has significant impact on the performance of the algorithm. One might think of using a dedicated *merger* processor. For each node $v \in V_i$, the merger collects N_v^i from all other processors and merges them into N_v. This requires $O(d_v)$ time for node v. Thus the runtime complexity for merging adjacency lists of all $v \in V$ is $O(\sum_{v \in V} d_v) = O(m)$, which is at most as good as the sequential algorithm.

Next we present our efficient parallel merging scheme which employs P parallel mergers.

An Efficient Parallel Merging Scheme

To parallelize Phase 2 efficiently, our algorithm distributes the corresponding computation disjointly among processors. Each processor i is responsible for merging adjacency lists N_v for nodes v in $V_i \subset V$ such that for any i and j, $V_i \cap V_j = \emptyset$ and $\bigcup_i V_i = V$. Note that this partitioning of

nodes is different from the initial partitioning of edges. How the nodes in V is distributed among processors crucially affects the load balancing and performance of the algorithm. Further, this partitioning and load balancing scheme should be parallel to ensure the efficiency of the algorithm. Later in this section, we discuss a parallel algorithm to partition set of nodes V which makes both space requirement and runtime well-balanced. Once the partitions V_i are given, the scheme for parallel merging works as follows.

- *Step 1:* Let S_i be the set of all local adjacency lists in partition i. Processor i divides S_i into P disjoint subsets S_i^j, $0 \leq j \leq P-1$, as defined below.

$$S_i^j = \{N_v^i : v \in V_j\}. \tag{2}$$

- *Step 2:* Processor i sends S_i^j to all other processors j. This step introduces non-trivial efficiency issues which we shall discuss shortly.

- *Step 3:* Once processor i gets S_j^i from all processors j, it constructs N_v for all $v \in V_i$ by the following equation.

$$N_v = \bigcup_{k:N_v^k \in S_k^i} N_v^k \tag{3}$$

We present two methods for performing Step 2 of the above scheme. The first method explicitly exchanges messages among processors to send and receive S_i^j by using message buffers (main memory). The other method uses disk space (external memory) to exchange S_i^j. We call the first method *message-based merging* and the second *external-memory merging*.

(1) Message-based Merging: Each processor i sends S_i^j directly to processor j via messages. Specifically, processor i sends $|N_v^i|$ (with a message $< v, N_v^i >$) to processor j where $v \in V_j$. A processor might send multiple lists to another processor. In such cases, messages to a particular processor are bundled together to reduce communication overhead. Once a processor i receives messages $< v, N_v^j >$ from other processors, for $v \in V_i$, it computes $N_v = \cup_{j=0}^{P-1} N_v^j$. The pseudocode of this algorithm is given in Figure 4.

```
1: for each v s.t. (., v) ∈ E_i ∨ (v, .) ∈ E_i do
2:     Send < v, N_v^i > to proc. j where v ∈ V_j
3: for each v ∈ V_i do
4:     N_v ← ∅
5: for each < v, N_v^j > received from any proc. j do
6:     N_v ← N_v ∪ N_v^j
```

Figure 4: Parallel algorithm for merging local adjacency lists to construct final adjacency lists N_v. A message, denoted by $< v, N_v^i >$, refers to local adjacency lists of v in processor i.

(2) External-memory Merging: Each processor i writes S_i^j in intermediate disk files F_i^j, one for each processor j. Processor i reads all files F_j^i for partial adjacency lists N_v^j for each $v \in V_i$ and merges them to final adjacency lists using step 3 of the above scheme. However, processor i doesn't read in the whole file into its main memory. It only stores local adjacency lists N_v^j of a node v at a time, merges it to N_v, releases memory and then proceeds to merge the next node $v+1$. This works correctly since while writing S_i^j in F_i^j, local adjacency lists N_v^i are listed in the sorted order of v. External-memory merging thus has a space requirement of $O(\max_v d_v)$. However, the I/O operation leads to a higher runtime with this method than message-based merging, although the asymptotic runtime complexity remains the same. We demonstrate this space-runtime tradeoff between these two methods in our *performance analysis* section.

The runtime and space complexity of parallel merging depends on the partitioning of V. Next, we discuss the partitioning and load balancing scheme followed by the complexity analyses.

Partitioning and Load Balancing

The performance of the algorithm depends on how loads are distributed. In Phase 1, distributing the edges of the input graph evenly among processors provides an even load balancing both in terms of runtime and space leading to both space and runtime complexity of $O(\frac{m}{P})$. However, Phase 2 is computationally different than Phase 1 and requires different partitioning and load balancing scheme.

In Phase 2 of our algorithm, set of nodes V is divided into P subsets V_i where processor i merges adjacency lists N_v for all $v \in V_i$. The time for merging N_v of a node v (referred to as *merging cost* henceforth) is proportional to the degree $d_v = |N_v|$ of node v. Total cost for merging incurred on a processor i is $\Theta(\sum_{v \in V_i} d_v)$. Distributing equal number of nodes among processors may not make the computing load well-balanced in many cases. Some nodes may have large degrees and some very small. As shown in Figure 5, distribution of merging cost $(\sum_{v \in V_i} d_v)$ across processors is very uneven with an equal number of nodes assigned to each processor. Thus the set V should be partitioned in such a way that the cost of merging is almost equal in all processors.

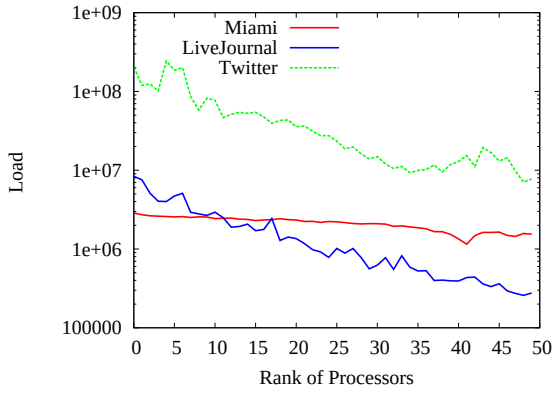


Figure 5: Load distribution among processors for LiveJournal, Miami and Twitter before applying the load balancing scheme. Summary of our dataset is provided in Table 2.

Let, $f(v)$ be the cost associated with constructing N_v by receiving and merging local adjacency lists for a node $v \in V$.

We need to compute P disjoint partitions of node set V such that for each partition V_i,

$$\sum_{v \in V_i} f(v) \approx \frac{1}{P} \sum_{v \in V} f(v).$$

Now, note that, for each node v, total size of the local adjacency lists received (in Line 5 in Figure 4) equals the number of adjacent nodes of v, i.e., $|N_v| = d_v$. Merging local adjacency lists N_v^j via set union operation (Line 6) also requires d_v time. Thus, $f(v) = d_v$. Now, since the adjacent nodes of a node v can reside in multiple processors, computing $f(v) = |N_v| = d_v$ requires communication among multiple processors. For all v, computing $f(v)$ sequentially requires $O(m + n)$ time which diminishes the advantages gained by the parallel algorithm. Thus, we compute $f(v) = d_v$ for all v in parallel, in $O(\frac{n+m}{P} + c)$ time, where c is the communication cost. We will discuss the complexity shortly. This algorithm works as follows: for determining d_v for $v \in V$ in parallel, each processor i computes d_v for $\frac{n}{P}$ nodes v, where v starts from $\frac{in}{P}$ to $\frac{(i+1)n}{P} - 1$. Such nodes v satisfy the equation, $\left\lfloor \frac{v}{n/P} \right\rfloor = i$. Now, for each local adjacency list N_v^i constructed in Phase 1, processor i sends $d_v^i = |N_v^i|$ to processor $j = \frac{v}{n/P}$ with a message $< v, d_v^i >$. Once processor i receives messages $< v, d_v^j >$ from other processors, it computes $f(v) = d_v = \sum_{j=0}^{P-1} d_v^j$ for all nodes v such that $\left\lfloor \frac{v}{n/P} \right\rfloor = i$. The pseudocode of the parallel algorithm for computing $f(v) = d_v$ is given in Figure 6.

```
1: for each v s.t. (., v) ∈ E_i ∨ (v, .) ∈ E_i do
2:     d_v^i ← |N_v^i|
3:     j ← v/(n/P)
4:     Send < v, d_v^i > to processor j
5: for each v s.t. ⌊v/(n/P)⌋ = i do
6:     d_v ← 0
7: for each < v, d_v^j > received from any proc. j do
8:     d_v ← d_v + d_v^j
```

Figure 6: Parallel algorithm executed by each processor i for computing $f(v) = d_v$.

Once $f(v)$ is computed for all $v \in V$, we compute cumulative sum $F(t) = \sum_{v=0}^{t} f(v)$ in parallel by using a parallel prefix sum algorithm [4]. Each processor i computes and stores $F(t)$ for nodes t, where t starts from $\frac{in}{P}$ to $\frac{(i+1)n}{P} - 1$. This computation takes $O(\frac{n}{P}+P)$ time. Then, we need to compute V_i such that computation loads are well-balanced among processors. Partitions V_i are disjoint subset of consecutive nodes, i.e., $V_i = \{n_i, n_i + 1 \ldots, n_{(i+1)} - 1\}$ for some node n_i. We call n_i *start* node or *boundary* node of partition i. Now, V_i is computed in such a way that the sum $\sum_{v \in V_i} f(v)$ becomes almost equal ($\frac{1}{P} \sum_{v \in V} f(v)$) for all partitions i. At the end of this execution, each processor i knows n_i and $n_{(i+1)}$. Algorithm presented in [6] compute V_i for the problem of triangle counting. The algorithm can also be applied for our problem to compute V_i using cost function $f(v) = d_v$. In summary, computing load balancing for Phase 2 has the following main steps.

- *Step 1:* Compute cost $f(v) = d_v$ for all v in parallel by the algorithm shown in Figure 6.
- *Step 2:* Compute cumulative sum $F(v)$ by a parallel prefix sum algorithm [4].
- *Step 3:* Compute boundary nodes n_i for every subset $V_i = \{n_i, \ldots, n_{(i+1)} - 1\}$ using the algorithms [1, 6].

LEMMA 2. *The algorithm for balancing loads for Phase 2 has a runtime complexity of $O(\frac{n+m}{P} + P + \max_i M_i)$ and a space requirement of $O(\frac{n}{P})$, where M_i is the number of messages received by processor i in Step 1.*

Proof: For *Step 1* of the above load balancing scheme, executing Line 1-4 (Figure 6) requires $O(|E_i|)$=$O(\frac{m}{P})$ time. The cost for executing Line 5-6 is $O(\frac{n}{P})$ since there are $\frac{n}{P}$ nodes v such that $\left\lfloor \frac{v}{n/P} \right\rfloor = i$. Each processor i sends a total of $O(\frac{m}{P})$ messages since $|E_i| = \frac{m}{P}$. If the number of messages received by processor i is M_i, then Line 7-8 of the algorithm has a complexity of $O(M_i)$ (we compute bounds for M_i in Lemma 3). Computing *Step 2* has a computational cost of $O(\frac{n}{P} + P)$ [4]. *Step 3* of the load balancing scheme requires $O(\frac{n}{P} + P)$ time [1, 6]. Thus the runtime complexity of the load balancing scheme is $O(\frac{n+m}{P} + P + \max_i M_i)$. Storing $f(v)$ for $\frac{n}{P}$ nodes has a space requirement of $O(\frac{n}{P})$. □

LEMMA 3. *Number of messages M_i received by processor i in Step 1 of load balancing scheme is bounded by $O(\min\{n, \sum_{in/P}^{(i+1)n/P-1} d_v\})$.*

Proof: Referring to Figure 6, each processor i computes d_v for $\frac{n}{P}$ nodes v, where v starts from $\frac{in}{P}$ to $\frac{(i+1)n}{P} - 1$. For each v, processor i may receive messages from at most $(P - 1)$ other processors. Thus, the number of received messages is at most $\frac{n}{P} \times (p-1) = O(n)$. Now, notice that, when all neighbors $u \in N_v$ of v reside in different partitions E_j, processor i might receive as much as $|N_v| = d_v$ messages for node v. This gives another upper bound, $M_i = O(\sum_{in/P}^{(i+1)n/P-1} d_v)$. Thus we have $M_i = O(\min\{n, \sum_{in/P}^{(i+1)n/P-1} d_v\})$. □

In most of the practical cases, each processor receives much smaller number of messages than that specified by the theoretical upper bound. Now, for each node v, processor i receives messages actually from fewer than $P - 1$ processors. Let, for node v, processor i receives messages from $O(P.l_v)$ processors, where l_v is a real number ($0 \leq l_v \leq 1$). Thus total number of message received, $M_i = O(\sum_{in/P}^{(i+1)n/P-1} P.l_v)$. To get a crude estimate of M_i, let $l_v = l$ for all v. The term l can be thought of as the average over all l_v. Then $M_i = O(\frac{n}{P}.P.l) = O(n.l)$. As shown in Table 1, the actual number of messages received M_i is up to 7× smaller than the theoretical bound.

LEMMA 4. *Using the load balancing scheme discussed in this section, Phase 2 of our parallel algorithm has a runtime*

Network	n	$\sum_{\frac{in}{P}}^{\frac{(i+1)n}{P}-1} d_v$	M_i	l(avg.)
Miami	2.1M	2.17M	600K	0.27
LiveJournal	4.8M	2.4M	$560K$	0.14
PA($5M, 20$)	$5M$	$2.48M$	$1.4M$	0.28

Table 1: Number of messages received in practice compared to the theoretical bounds. This results report $\max_i M_i$ with $P = 50$. Summary of our dataset is provided in Table 2.

complexity of $O(\frac{m}{P})$. Further, the space required to construct all final adjacency lists N_v in a partition is $O(\frac{m}{P})$.

Proof: Line 1-2 in the algorithm shown in Figure 4 requires $O(|E_i|)$=$O(\frac{m}{P})$ time for sending at most $|E_i|$ edges to other processors. Now, with load balancing, each processor receives and merges at most $O(\sum_{v \in V} d_v/P) = O(\frac{m}{P})$ edges (Line 5-6). Thus the cost for merging local lists N_v^j into final list N_v has a runtime of $O(\frac{m}{P})$. Since the total size of the local and final adjacent lists in a partition is $O(\frac{m}{P})$, the space requirement is $O(\frac{m}{P})$. □

The runtime and space complexity of our complete parallel algorithm are formally presented in Theorem 1.

THEOREM 1. *The runtime and space complexity of our parallel algorithm is $O(\frac{m}{P} + P + n)$ and $O(\frac{m}{P})$, respectively.*

Proof: The proof follows directly from Lemma 1, 2, 3, and 4.□

The total space required by all processors to process m edges is $O(m)$. Thus the space complexity $O(\frac{m}{P})$ of our parallel algorithm is optimal.

Performance gain with load balancing: Cost for merging incurred on each processor i is $\Theta(\sum_{v \in V_i} d_v)$ (Figure 4). Without load balancing, this cost $\Theta(\sum_{v \in V_i} d_v)$ can be as much as $\Theta(m)$ (it is easy to construct such skewed graphs) leading the runtime complexity of the algorithm $\Theta(m)$. With load balancing scheme our algorithm achieves a runtime of $O(\frac{m}{P} + P + n) = O(\frac{m}{P} + \frac{m}{d_{avg}})$, for usual case $n > P$. Thus, by simple algebraic manipulation, it is easy to see, the algorithm with load balancing scheme achieves a $\Omega(\min\{P, d_{avg}\})$-factor gain in runtime efficiency over the algorithm without load balancing scheme. In other words, the algorithm gains a $\Omega(P)$-fold improvement in speedup when $d_{avg} \geq P$ and $\Omega(d_{avg})$-fold otherwise. We demonstrate this gain in speedup with experimental results in our *performance analysis* section.

DATA AND EXPERIMENTAL SETUP

We present the datasets and the specification of computing resources used in our experiments below.

Datasets. We used both real-world and artificially generated networks for evaluating the performance of our algorithm. A summary of all the networks is provided in Table 2. The number of nodes in our networks ranges from 37K to 1B and the number of edges from 0.36M to 20B. Twitter [17], web-BerkStan, Email-Enron and LiveJournal [20] are real-world networks. Miami is a synthetic, but realistic, social contact network [6, 9] for Miami city. Networks Gnp(n, d) and PA(n, d) are random networks. Gnp(n, d) is generated using the Erdős-Réyni random graph model [10] with n nodes and d average degree. Network PA(n, d) is generated using preferential attachment (PA) model [8] with n nodes and average degree d. Both the real-world and PA(n, d) networks have skewed degree distributions which make load balancing a challenging task and give us a chance to measure the performance of our algorithms in some of the worst case scenarios.

Network	Nodes	Edges	Source
Email-Enron	37K	0.36M	SNAP [20]
web-BerkStan	0.69M	13M	SNAP [20]
Miami	2.1M	100M	[9]
LiveJournal	4.8M	86M	SNAP [20]
Twitter	42M	2.4B	[17]
Gnp(n, d)	n	$\frac{1}{2}nd$	Erdős-Réyni
PA(n, d)	n	$\frac{1}{2}nd$	Pref. Attachment

Table 2: Dataset used in our experiments. Notations K, M and B denote thousands, millions and billions, respectively.

Experimental Setup. We perform our experiments using a high performance computing cluster with 64 computing nodes, 16 processors (Sandy Bridge E5-2670, 2.6GHz) per node, memory 4GB/processor, and operating system SLES 11.1.

PERFORMANCE ANALYSIS

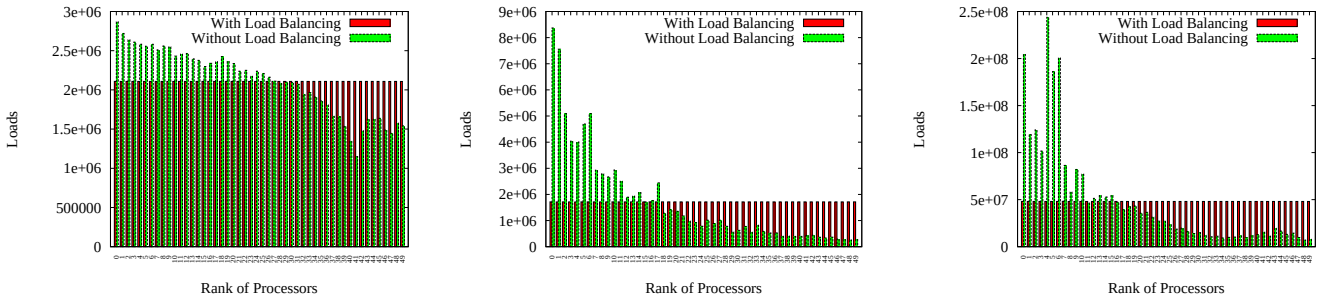
In this section, we present the experimental results evaluating the performance of our algorithm.

Load distribution

Load distribution among processors can be very uneven without applying our load balancing scheme, as discussed in *partitioning and load balancing* section. We show a comparison of load distribution on various networks with and without load balancing scheme in Figure 7. Our scheme provides an almost equal load among the processors, even for graphs with very skewed degree distribution such as LiveJournal and Twitter. Loads are significantly uneven for such skewed networks without load balancing scheme.

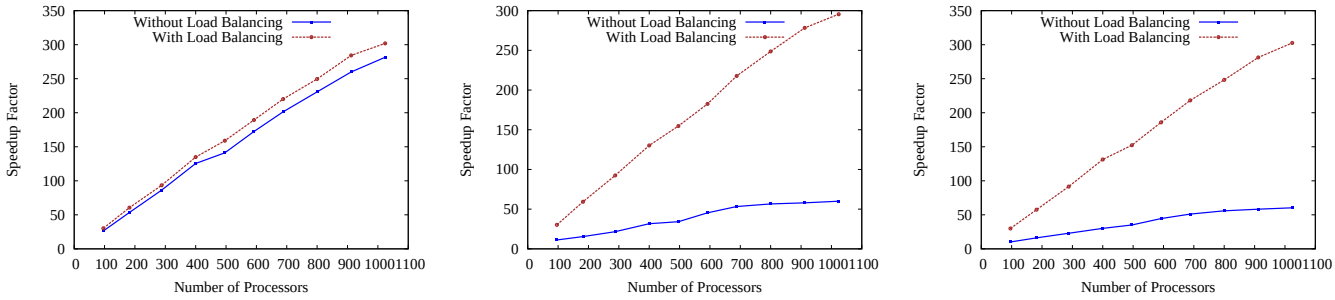
Strong Scaling

Figure 8 shows strong scaling (speedup) of our algorithm on LiveJournal, Miami and Twitter networks with and without load balancing scheme. Our algorithm demonstrates very good speedups, e.g., it achieves a speedup factor of ≈ 300 with 1024 processors for Twitter network. Speedup factors increase almost linearly for all networks, and the algorithm scales to a large number of processors. Figure 8 also shows the speedup factors the algorithm achieves without load balancing scheme. Speedup factors with load balancing scheme are significantly higher than those without load balancing scheme. For Miami network, the differences in speedup factors are not very large since Miami has a relatively even degree distribution and loads are already fairly balanced without load balancing scheme. However, for real-world skewed



(a) Miami network (b) LiveJournal network (c) Twitter network

Figure 7: Load distribution among processors for LiveJournal, Miami and Twitter networks by different schemes.



(a) Miami network (b) LiveJournal network (c) Twitter network

Figure 8: Strong scaling of our algorithm on LiveJournal, Miami and Twitter networks with and without load balancing scheme. Computation of speedup factors includes the cost for load balancing.

networks, our load balancing scheme always improves the speedup quite significantly– for example, with 1024 processors, the algorithm achieves a speedup factor of 297 with load balancing scheme compared to 60 without load balancing scheme for LiveJournal network.

This experiment also demonstrates that our algorithm scales to a large number of processors. The speedup factors continue to grow almost linearly up to 1024 processors.

Comparison between Message-based and External-memory Merging

We compare the runtime and memory usage of our algorithm with both message-based and external-memory merging. Message-based merging is very fast and uses message buffers in main memory for communication. On the other hand, external-memory merging saves main memory by using disk space even though it requires large runtime for I/O operations. Thus these two methods provide desirable alternatives to trade-off between space and runtime. However, as shown in Table 3, message-based merging is significantly faster (up to 20×) than external-memory merging albeit taking a little larger space. Thus message-based merging is the preferable method in our fast parallel algorithm.

Network	Memory (MB)		Runtime (s)	
	EXT	MSG	EXT	MSG
Email-Enron	1.8	2.4	3.371	0.078
web-BerkStan	7.6	10.3	10.893	1.578
Miami	26.5	43.34	33.678	6.015
LiveJournal	28.7	42.4	31.075	5.112
Twitter	685.93	1062.7	1800.984	90.894
Gnp(500K, 20)	6.1	9.8	6.946	1.001
PA(5M, 20)	68.2	100.1	35.837	7.132
PA(1B, 20)	9830.5	12896.6	14401.5	1198.30

Table 3: Comparison of external-memory (EXT) and message-based (MSG) merging (using 50 processors).

Weak Scaling

Weak scaling of a parallel algorithm shows the ability of the algorithm to maintain constant computation time when the problem size grows proportionally with the increasing number of processors. As shown in Figure 9, total computation time of our algorithm (including load balancing time) grows slowly with the addition of processors. This is expected since the communication overhead increases with additional processors. However, the growth of runtime of our algorithm is

rather slow and remains almost constant, the weak scaling of the algorithm is very good.

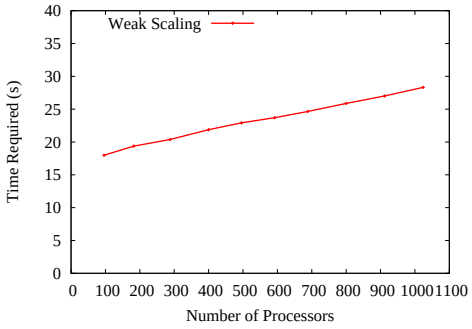


Figure 9: Weak scaling of our parallel algorithm. For this experiment we use networks PA($x/10 \times 1M, 20$) for x processors.

CONCLUSION

We present a parallel algorithm for converting *edge-list* of a graph to *adjacency-list*. The algorithm scales well to a large number of processors and works on massive graphs. We devise a load balancing scheme that improves both space efficiency and runtime of the algorithm, even for networks with very skewed degree distributions. To the best of our knowledge, it is the first parallel algorithm to convert edge list to adjacency list for large-scale graphs. It also allows other graph algorithms to work on massive graphs which emerge naturally as edge lists. Furthermore, this work demonstrates how a seemingly trivial problem becomes challenging when we are dealing with Bigdata.

ACKNOWLEDGMENT

We thank our external collaborators, members of Network Dynamics & Simulation Science Laboratory, and anonymous reviewers for their suggestions and comments. This work has been partially supported by DTRA Grant HDTRA1-11-1-0016, DTRA CNIMS Contract HDTRA1-11-D-0016-0001, NSF NetSE Grant CNS-1011769, and NSF SDCI Grant OCI-1032677.

REFERENCES

1. Alam, M., and Khan, M. Efficient algorithms for generating massive random networks. Technical Report 13-064, NDSSL at Virginia Tech, May 2013.
2. Alam, M., Khan, M., and Marathe, M. Distributed-memory parallel algorithms for generating massive scale-free networks using preferential attachment model. In *Proc. of the Intl. Conf. on High Performance Computing, Networking, Storage and Analysis* (2013).
3. Alon, N., Yuster, R., and Zwick, U. Finding and counting given length cycles. *Algorithmica 17* (1997), 209–223.
4. Aluru, S. Teaching parallel computing through parallel prefix. In *the Intl. Conf. on High Performance Computing, Networking, Storage and Analysis* (2012).
5. Apte, C., et al. Business applications of data mining. *Communications of the ACM 45*, 8 (Aug. 2002), 49–53.
6. Arifuzzaman, S., Khan, M., and Marathe, M. Patric: A parallel algorithm for counting triangles in massive networks. In *Proc. of the 22nd ACM Intl. Conf. on Information and Knowledge Management* (2013).
7. Attewell, P., and Monaghan, D. *Data Mining for the Social Sciences: An Introduction*. University of California Press, 2015.
8. Barabasi, A., et al. Emergence of scaling in random networks. *Science 286* (1999), 509–512.
9. Barrett, C., et al. Generation and analysis of large synthetic social contact networks. In *Proc. of the 2009 Winter Simulation Conference* (2009), 103–114.
10. Bollobas, B. *Random Graphs*. Cambridge Univ. Press, 2001.
11. Broder, A., et al. Graph structure in the web. *Computer Networks* (2000), 309–320.
12. Chen, J., and Lonardi, S. *Biological Data Mining*, 1st ed. Chapman & Hall/CRC, 2009.
13. Chu, S., and Cheng, J. Triangle listing in massive networks and its applications. In *Proc. of the 17th ACM SIGKDD Conf. on Knowledge Discovery and Data Mining* (2011), 672–680.
14. Chung, F., et al. *Complex Graphs and Networks*. American Mathematical Society, Aug. 2006.
15. Coppersmith, D., and Winograd, S. Matrix multiplication via arithmetic progressions. In *Proc. of the 19th Annual ACM Symposium on Theory of Computing* (1987), 1–6.
16. Girvan, M., and Newman, M. Community structure in social and biological networks. *Proc. Natl. Acad. of Sci. USA 99*, 12 (June 2002), 7821–7826.
17. Kwak, H., et al. What is twitter, a social network or a news media? In *Proc. of the 19th Intl. Conf. on World Wide web* (2010), 591–600.
18. Leskovec, J. Dynamics of large networks. In *Ph.D. Thesis, Pittsburgh, PA, USA*. (2008).
19. Newman, M. Coauthorship networks and patterns of scientific collaboration. *Proc. Natl. Acad. of Sci. USA 101*, 1 (April 2004), 5200–5205.
20. SNAP. Stanford network analysis project. `http://snap.stanford.edu/`, 2012.