

Reducing Communication in Parallel Breadth-First Search on Distributed Memory Systems

Huiwei Lu^{*†}, Guangming Tan^{*}, Mingyu Chen^{*}, Ninghui Sun^{*}

^{*}State Key Laboratory of Computer Architecture,
Institute of Computing Technology, Chinese Academy of Sciences

[†]Argonne National Laboratory

Email: huiweilu@mcs.anl.gov, tgm@ict.ac.cn, cmy@ict.ac.cn, snh@ncic.ac.cn

Abstract—Breadth-first search (BFS) is a key operation in data-intensive graph analysis applications. However, for distributed BFS algorithm on large distributed memory systems, data communication often limits the scalability of the algorithm as it costs significantly more than arithmetic computation. In this work, we try to reduce the communication cost in distributed BFS by sieving and compressing the messages. First, we propose a novel distributed directory to sieve the redundant data in collective communications. Then we leverage a bitmap compression algorithm to further reduce the size of messages in communication. Experiments on a 6,144-core Intel Westmere based cluster show our algorithm achieve a BFS performance rate of 12.1 billion edge visits per second on an undirected graph of 8 billion vertices and 128 billion edges with scale-free distribution. Compared to the “replicated-csr” version BFS in Graph500, our algorithm reduces communication cost by 79.0% and gets a speedup of 2.2 \times .

I. INTRODUCTION

Recently, graph has been extensively used to abstract complex systems and interactions in emerging “big data” applications, such as social network analysis, world wide web, biological systems and data mining. With the increasing growth in these areas, petabyte-sized graph datasets are produced for knowledge discovery [1], [2], which could only be solved by distributed memory systems. These data-intensive applications are new but increasingly important high performance computing (HPC) workloads. To guide the design of hardware architectures and software systems intended to support such applications, a new benchmark, Graph500 [3], is established in 2010 by HPC community to evaluate the data processing ability of current hardware platforms, and breadth-first search (BFS) is chosen as the first representative application kernel. As it serves as a building block for a great many graph algorithms such as minimum spanning tree, betweenness centrality, and shortest paths [4], [5], [6].

However, implementing a parallel BFS algorithm on a distributed memory system with high performance is a challenging task due to inherent characteristics of graph problems. First, data access in a BFS algorithm is highly irregular, which lead to poor locality and little data reuse. Second, BFS algorithms have very low computation to data access ratio. Its performance is limited by the data movement, – in distributed

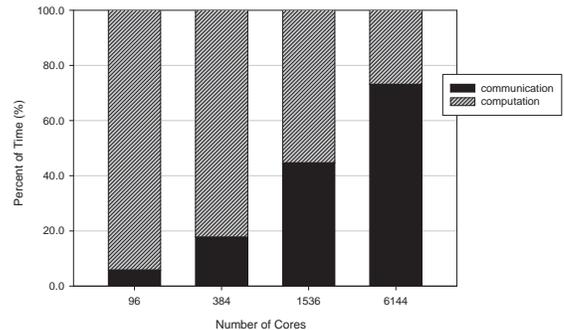


Fig. 1. Time breakdown of the “replicated-csr” BFS algorithm in Graph500 in a weak scaling experiment on an Intel Westmere based cluster that use fixed problem size per computing node (each node with about 16M vertices). Communication time becomes the bottleneck when scale goes up.

memory systems, the communication [2], [7]. Finally, the percent of communication time goes up when scale goes up. For example, on a 6,144-core Intel Westmere based cluster system, the “replicated-csr” BFS algorithm in Graph500 spends about 70% time on communication traversing a scale-free graph with 8 billion vertices (Fig. 1). Therefore, to optimize BFS algorithms in large-scale distributed memory systems, the most critical task is to minimize its communication.

Different approaches have been proposed to optimize communication in distributed BFS. Yoo [8] and Buluc [7] use two-dimensional partitioning of the graph to reduce communication overhead. The “replicated-csr” version BFS in Graph500 reference code using bitmap to reduce the size of messages [3]. In this paper, we will focus on minimizing the size of communication messages. We design a novel distributed cross directory to remove redundant data in the collective communication operations. We further leverage data compression techniques to reduce messages sizes, and evaluate the tradeoff of compression ratio and compression time in different compression techniques. The main contributions of this paper include:

- We design a novel distributed data structure, cross directory, to reduce communication cost in distributed BFS algorithms, with which we propose a new distributed BFS algorithm that eliminates the redundant communication operations. Experiments show that the

This work was done when Huiwei Lu was a student in the Institute of Computing Technology, Chinese Academy of Sciences.

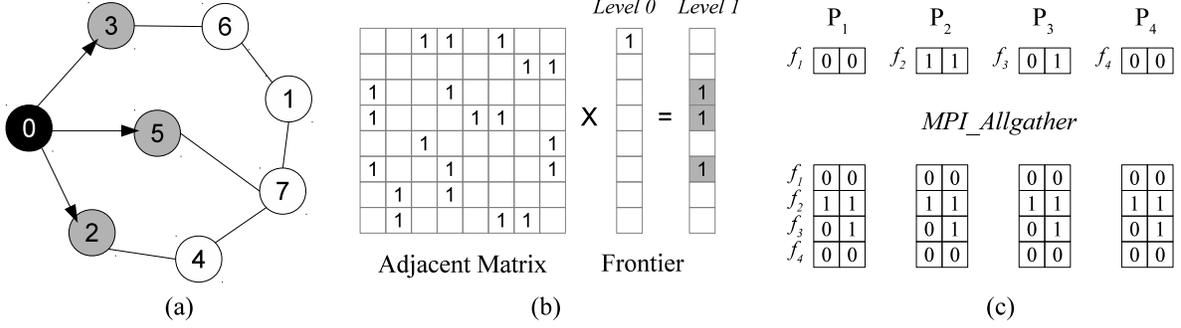


Fig. 2. The operation of (a) one BFS iteration on an undirected graph is equivalent to (b) one matrix vector multiplication. Frontier i can be obtained by multiplication of the adjacent matrix and frontier $i - 1$. (c) For distributed BFS, the frontier is partitioned among processors, which needs to be gathered together using `MPI_Allgather`.

new algorithm scales well on large-scale parallel computers.

- We evaluate several compression techniques to reduce communication volume in distributed BFS algorithms. In this case, by compressing the messages (bitmap vectors) on the fly before communication, we reduce the message size in communication effectively at the cost of relatively cheap arithmetic operations.
- Experiments on a 6,144-core Intel Westmere cluster shows our proposed distributed BFS algorithm is capable of traversing a scale-free graph of 8 billion (2^{33}) vertices and 128 billion edges at a speed of 12.1 billion of traversed edges per seconds, and achieves a total 79.0% communication reduction and an averaged $2.2\times$ performance improvement over the baseline algorithm.

II. BASELINE BFS

A. Baseline BFS Described in Linear Algebra

We will first describe the baseline algorithm in linear algebra. Let A denote the adjacency matrix of the graph G , f_{Lk} denote the frontier at level k , and $\pi_k = \bigcup_{i=1}^k f_{Li}$ denote the visited information of previous frontiers. The exploration of level k in BFS is algebraically equivalent to a sparse matrix vector multiplication (SpMV): $f_{L(k+1)} \leftarrow A^T \otimes f_{Lk} \odot \overline{\pi_k}$. For example, in Fig. 2 (a), traversing from level 0 to level 1 is equivalent to the matrix vector multiplication in Fig. 2 (b). The syntax \otimes denotes the matrix-vector multiplication operation, \odot denotes element-wise multiplication, where $(a_1, a_2, \dots, a_n)^T \odot (b_1, b_2, \dots, b_n)^T = (a_1 b_1, a_2 b_2, \dots, a_n b_n)^T$, and overline represents the complement operation. In other words, $\overline{v_i} = 0$ for $v_i \neq 0$ and $\overline{v_i} = 1$ for $v_i = 0$.

Algorithm 1 describes the baseline BFS. Each loop block (starting in line 3) performs a single level traversal. f represents the current frontier, which is initialized as an empty bitmap; t is a bitmap that holds the temporary parent information for that iteration only; π is the visited information of previous frontiers. The computational step (line 4,5,6) can be efficiently parallelized with multithreading. For SpMV operation in line 4, the matrix data is naturally split into pieces for multithreading. At the end of each loop, `ALLGATHER`

Algorithm 1: Baseline BFS described in linear algebra.

```

Input : s: source vertex id
1  $f(s) \leftarrow s$ ;
2 foreach processor  $P_i$  in parallel do
3   while  $f \neq \emptyset$  do
4      $t_i \leftarrow A_i \otimes f$ ;
5      $t_i \leftarrow t_i \odot \overline{\pi_i}$ ;  $\pi_i \leftarrow \pi_i + t_i$ ;
6      $f_i \leftarrow t_i$ ;
7      $f \leftarrow \text{ALLGATHER}(f_i, P_i)$ ;

```

updates f with `MPI_Allgather`, which gathers all f_i from P_i (Fig. 2 (c)).

There are several reference BFS algorithms in Graph500. We choose the “replicated-csr” implementation as the baseline BFS for following reasons: First, the reference “replicated-csr” BFS in Graph500 is much faster than the “simple” and the “replicated-csc” [9]. The “replicated-csr” is faster than “simple” because it use bitmap for the frontier vertices. Instead of use 64-bit for each vertex, bitmap use only one bit for each vertex, thus considerably reduces the size of the communication messages. For a scale-free graph of diameter d , the use of bitmap save $64/d$ in the size of communication messages. Second, it is used as a subroutine in many state-of-the-art BFS algorithms [10], [7], [8]. The improvement of “replicated-csr” BFS is also applicable to these algorithms. However, the “replicated-csr” BFS has several limitations: 1. it use broadcast to update the frontier vertices, which cause unnecessary communication; 2. the bitmap needs to include all vertices in the frontier to form a set, even when there is only a few vertices in the frontier. In the next section we will propose two improvements to the “replicated-csr” BFS algorithm.

III. BFS WITH SIEVE AND COMPRESSION

A. Sieve

The problem of bitmap in Algorithm 1 is that each processor gets all frontier vertices from all the processors, regardless whether a vertex is useful to each processor. For example, in Fig. 3 (a), v_2 does not has a direct edge connecting to the vertices of P_4 , so sending it to P_4 will be useless. However, since v_2 is included in the bitmap, it will be sent anyway (Fig. 3 (b)).

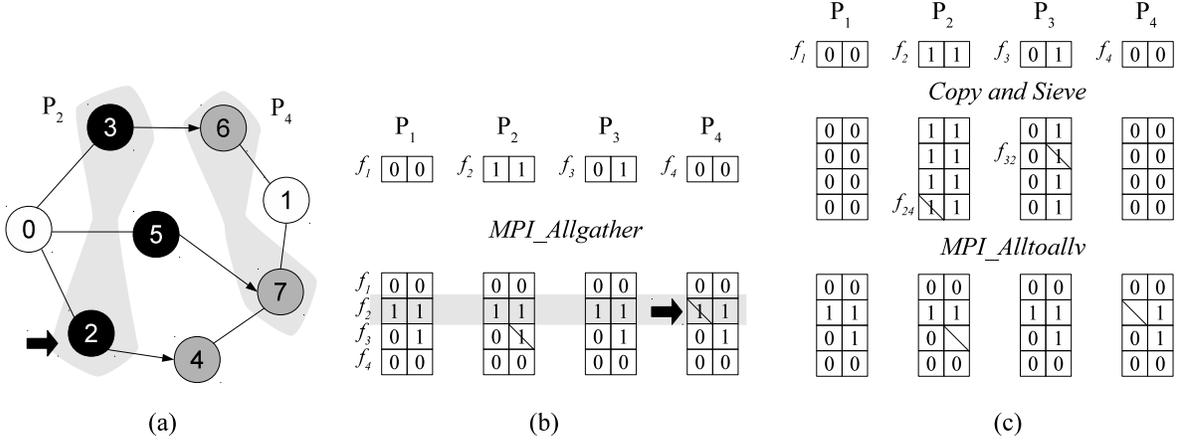


Fig. 3. BFS with sieve. (a) v_2 has no direct connection with P_4 . (b) The wasted communication of v_2 in $MPI_Allgather$. (c) Use sieving to eliminate unnecessary messages.

To explain this in linear algebra, let

$$A \otimes f = \begin{pmatrix} A_1 \\ A_2 \\ \vdots \\ A_p \end{pmatrix} \begin{pmatrix} f_1 \\ f_2 \\ \vdots \\ f_p \end{pmatrix}, A_i = (A_{i,1} \ A_{i,2} \ \cdots \ A_{i,p}), \quad (1)$$

then $f_4 = \sum_{i=1}^4 A_{4,i} \otimes f_i$. The connection of P_2 and P_4 is

$$A_{4,2} \otimes f_2 = \begin{pmatrix} 01 \\ 00 \end{pmatrix} \begin{pmatrix} x_1 \\ x_2 \end{pmatrix} = \begin{pmatrix} y_1 \\ y_2 \end{pmatrix}. \quad (2)$$

Denote $A_{i,j} = [a_{i,j}]_{m \times n}$, because $a_{0,0}$ and $a_{1,0}$ of $A_{4,2}$ are always zero, then whatever x_1 is, y_1 will always be zero. So we always set x_1 to be zero in communication as step one, the benefit of this is that we can eliminate the zeros by using compression in step two.

To set vertices like x_1 to be zero, we formally define a *directory vector* as follows: for each item v_k in vector $V_{i,j}$, v_k is set to one if column k in $A_{i,j}$ contains at least one non-zero.

$$V_{i,j} = (v_1, v_2, \dots, v_n) \quad (3)$$

$$\text{where } v_k = \begin{cases} 1, & \exists a_{i,k} = 1, i \in [1, m], k \in [1, n] \\ 0, & \text{otherwise} \end{cases}$$

For the above example, $V_{4,2} = (0, 1)$ is sent to P_2 from P_4 during initialization. When traversing begins, f_2 is sieved into $f_{2,4} = f_2 \odot V_{4,2} = (1, 1)^T \odot (0, 1)^T = (0, 1)^T$, so we only send one vertex (in compressed bitmap format) back instead of two (Fig. 3). This ‘‘sieve effect’’ is where communication is reduced.

The *cross directory* of processor P_i is defined as:

$$\mathbb{C}_i = \{V_{x,i} \text{ or } V_{i,x} \mid x = 1, 2, \dots, p\} \quad (4)$$

Besides a row of directory vectors $V_i = \{V_{i,y} \mid y = 1, 2, \dots, p\}$, P_i own a copy of the directory vectors $\{V_{x,i} \mid x = 1, 2, \dots, p\}$ in column i . The directory in the column direction is established during initialization and used to provide a local lookup for sieving.

Algorithm 2: Distributed BFS with sieving and compression.

Data: $f'_i = \{f'_{i,1}, f'_{i,2}, \dots, f'_{i,n-1}\}$:send buffer;
 $g'_i = \{g'_{i,1}, g'_{i,2}, \dots, g'_{i,n-1}\}$:receive buffer;
 \mathbb{C}_i :cross directory for P_i .

```

1  $f(s) \leftarrow s$ ;
2 initialize  $\mathbb{C}_i$ ;
3 foreach processor  $P_i$  in parallel do
4   while  $f \neq \emptyset$  do
5      $t_i \leftarrow \sum_{j=1}^n A_{i,j} \otimes f_{i,j}$ ;
6      $t_i \leftarrow t_i \odot \pi_i$ ;
7      $\pi_i \leftarrow \pi_i + t_i$ ;  $f_i \leftarrow t_i$ ;
8     foreach  $j \in [0, n)$  in parallel do
9        $f_{i,j} = f_i \odot V_{j,i}$ ; /* sieving */;
10       $f'_{i,j} \leftarrow \text{Compress}(f_{i,j})$ ;
11       $g'_i \leftarrow \text{ALLTOALLV}(f'_i, P_i)$ ;
12      foreach  $j \in [0, n)$  in parallel do
13         $f_{i,j} \leftarrow \text{Uncompress}(g'_{i,j})$ ;

```

B. Compression

Sieving helps to eliminate the unnecessary vertices in the frontier vector. However, instead of being completely removed from the bitmap, the unnecessary vertices are set to zero after sieving, as the bitmap need to maintain all bits in the set to index each bit. This limitation of bitmap holds back the benefit of sieving to reduce the message size in communication. A closer examination of the bitmap reveals that most bits in the bitmap are zero. This leads us to the idea of using lossless compression to eliminate the redundancy of the zeros. Fig. 4 illustrates the process of sieving and compressing. Algorithm 2 is our directory-based algorithm with compression and sieve: based on Algorithm 1, Algorithm 2 first sieves the frontier bitmap with the directory vector (line 9), making the bitmap sparser; then it compresses this sieved bitmap (line 10) and send it with ALLTOALLV (line 11); after received the compressed bitmap, the original vector could be restored with uncompression (line 13).

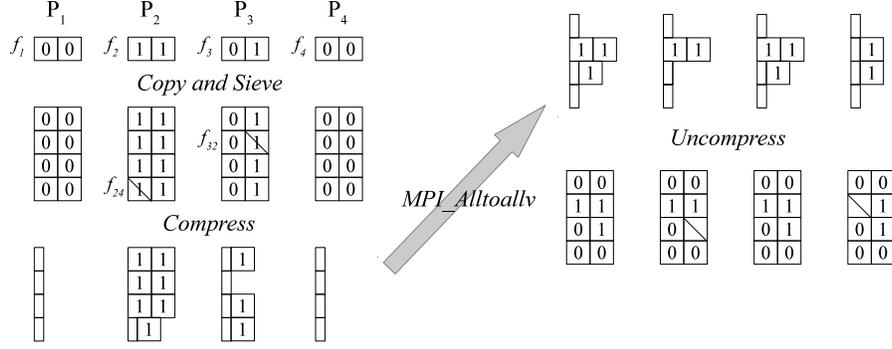


Fig. 4. BFS with sieve and compression.

TABLE I. A WAH COMPRESSED BITMAP.

16 bits	1000000000000000
3-bit groups	100 000 000 000 000 0
WAH	0100 1100 0000

We use word-aligned hybrid (WAH) [11] for *Compress* and *Uncompress* functions. We can use any lossless compression algorithm here, for example, LZ77 [12] or other text compression schemes. However, the benefit depends on the compression cost and compression ratio of a compression algorithm. WAH is chosen for its fast compression speed and reasonable compression ratio. We also evaluate multiple compression methods from Zlib library [13] in Section V.

Table I gives an example of a WAH compressed bitmap. In WAH, there are three types of words: literal words, fill words and active words. The most significant bit of a word is used to distinguish between a literal word (0) and a fill word (1). And a active word stores the last few bits. We assume that each computer word contains 4 bits and all fill bits are 0 in this example. Under this assumption, each literal word stores 3 bits from the bitmap, and each fill word represents a multiple of 3 bits. The second line in Table I shows the bitmap as 3-bit groups. The last line shows the WAH words. The first two words are regular words, the first is a literal word, and the second a fill word. The fill word 1100 indicates a 0-fill of 4 words long (containing 12 consecutive 0 bits). Note that the fill word stores the fill length as 4 rather than 12. The third word is the active word; it stores the last few bits that could not be stored in a regular word. For sparse bitmaps, where most of the bits are 0, a WAH compressed bitmap would consist of pairs of a fill word and a literal word [11].

IV. ALGORITHMIC ANALYSIS

A. Proof the Correctness of Sieving

We prove the correctness of Algorithm 2 by proving its equivalence to Algorithm 1. Recall the observation that there is wasted communication in the collective communication ALLGATHER, which leads to the motive for using cross directory to sieve the unnecessary messages in section III-A. The fact behind this scene is that the unconnected parts of the two communication sides will always result in a zero in the product of matrix-vector multiplication. For example in

Fig. 3, whatever f_2 is, the product of $A_{4,2} \otimes f_2$ will always be $(0, y_2)^T$ because the first column of $A_{4,2}$ is all zero (in the graph, it means no vertex of P_4 is connected to v_2). And the directory vector defined in equation 3 is used to record this information and help identify the zero columns out. The columns that consists only of zeros will be marked as 0 in the cross vector. And the result of matrix-vector multiplication remains correct as long as $V_{j,i}$ is correct at the zero positions. More formally, we prove it as follows.

Lemma 4.1: $A_{i,j} \otimes f_j = A_{i,j} \otimes f_j \odot V_{j,i}$.

Proof: Let $X = (x_1, x_2, \dots, x_n)^T = A_{i,j} \otimes f_j$, $V_{j,i} = (v_1, v_2, \dots, v_n)^T$, $Y = (y_1, y_2, \dots, y_n)^T = A_{i,j} \otimes f_j \odot V_{j,i}$, then $Y = X \odot V_{j,i} = (x_1 v_1, x_2 v_2, \dots, x_n v_n)^T$. To prove $X = Y$, we prove $\forall i \in [1, n]$, $x_i = x_i v_i$.

Denote matrix $A_{i,j} = [a_{k,l}]_{m \times n}$, $f_j = (z_1, z_2, \dots, z_n)^T$, as $X = A_{i,j} \otimes f_j$, then $x_k = \sum_{l=1}^n a_{k,l} z_l$.

According to the definition of directory vector $V_{j,i}$ in section III-A, if $v_k = 0 \Rightarrow \forall a_{k,i} = 0, i \in [1, n] \Rightarrow x_k = \sum_{l=1}^n a_{k,l} z_l = 0$, so $y_k = x_k v_k = 0 = x_k$; if $v_k = 1 \Rightarrow x_k = x_k v_k = y_k$. Thus for $i \in [1, n]$, $x_i = y_i$, then $X = Y$. ■

The above Lemma proves that each partial matrix-vector multiplication product is effectively the same after unnecessary messages has been filtered out by the cross directory. In Algorithm 2 (line 5), these partial products are combined to form a whole temporary frontier t_i . Compared with Algorithm 1 (line 4) before sieving, these two approaches will get the same t_i as long as each partial product of t_i remains the same. The equivalence is proved formally as follows.

Lemma 4.2: $t_i = \bigcup_{j=1}^n A_{i,j} \otimes f_{i,j}$ in Algorithm 2 (line 5) is equivalent to $t_i = A_i \otimes f$ in Algorithm 1 (line 4).

Proof: In Algorithm 2, for $\forall j \in [1, n]$, $f_{i,j} = f_i \odot V_{j,i}$, according to Lemma 4.1, $\bigcup_{j=1}^n A_{i,j} \otimes f_{i,j} = \bigcup_{j=1}^n A_{i,j} \otimes f_i \odot V_{j,i} = \bigcup_{j=1}^n A_{i,j} \otimes f_i = A_i \otimes f = t_i$. ■

Let's take a look at a detail example. Using Algorithm 1 in Fig. 3, P_4 gets its t_i as

$$t_4 = A_4 \otimes f = \begin{pmatrix} 01010000 \\ 01001100 \end{pmatrix} \otimes (00110100)^T = \begin{pmatrix} 1 \\ 1 \end{pmatrix} \quad (5)$$

Using Algorithm 2, P_4 get t_4 as

$$\begin{aligned} t_4 &= A_{4,1} \otimes f_{4,1} \cup A_{4,2} \otimes f_{4,2} \cup A_{4,3} \otimes f_{4,3} \cup A_{4,4} \otimes f_{4,4} \\ &= \begin{pmatrix} 01 \\ 01 \end{pmatrix} \otimes (0,0)^T \cup \begin{pmatrix} 01 \\ 00 \end{pmatrix} \otimes (0,1)^T \cup \begin{pmatrix} 00 \\ 11 \end{pmatrix} \otimes (0,1)^T \\ &\quad \cup \begin{pmatrix} 00 \\ 00 \end{pmatrix} \otimes (0,0)^T = \begin{pmatrix} 1 \\ 1 \end{pmatrix} \end{aligned} \quad (6)$$

The two different algorithms eventually get the same result.

B. Communication Cost

We study the parallel BFS problem in the message passing model of distributed computing: every processor has its own local memory, and data exchange between processors are done by message passing. The time taken to send a message between any two processors can be modeled as $T(n) = \alpha + n\beta$, where α is the latency (or startup time) per message, independent of message size, β is the transfer time per byte (inverse of bandwidth), and n is the number of bytes transferred [14]. This time cost model is generally used to model data movement either between levels of a memory hierarchy or over a network connecting processors. In this paper, we focus on the latter case. To simplify the analysis, we assume bandwidth cost is much bigger than latency cost ($n\beta \gg \alpha$), — as the dataset of distributed BFS is big, — therefore $T(n)$ will be dominated by the bandwidth cost $n\beta$. For a given network, β is constant, so the communication cost is in direct proportion to the message size n . Let *communication volume of a processor* \mathcal{V}_i be the size of all messages communicated on processor P_i in an algorithm. The *communication volume of an algorithm* is defined as $\mathcal{V} = \max\{\mathcal{V}_i \mid i \in [1, p]\}$.

The communication volume of MPI collective communication is derived from [15], [16]: For p processors, when each processor needs to broadcast n/p size of message to others, the communication volume of both allgather and alltoall are $O(n)$. There are many algorithms for allgather, for example, ring and recursive doubling [15]. The ring algorithm finishes in $p-1$ steps, in each step process i send n/p data to process $i+1$ and receives n/p from process $i-1$; the recursive doubling algorithm finishes in $\log p$ steps, in step s process i exchanges ns/p data with process $(i+s)\%p$. The time taken for these two algorithm is $T_{ring} = (p-1)\alpha + \frac{p-1}{p}n\beta$ and $T_{rec_dbl} = \log p\alpha + \frac{p-1}{p}n\beta$, respectively. No matter what algorithm is used, the bandwidth cost is the same $\frac{p-1}{p}n\beta$. In data-intensive applications like BFS, we assume bandwidth cost is much bigger than latency cost, so its communication volume is bound to $O(n)$. The communication volume of alltoall can be done in the same manner [16].

For graph $G(V, E)$, let $m = |E|$, $n = |V|$, let d be the diameter of the graph. At each level of BFS, the communication volume of allgather (Algorithm 1, line 7) is $O(n)$; the algorithm will finish at level d . So the communication volume of Algorithm 1 is $d \times O(n)$.

For Algorithm 2, let p be the number of the processors, $e = m/n$ be the average degree of a vertex, and σ' be the compression ratio factor of Algorithm 2. The communication volume of Algorithm 2 is $\sigma' \times d \times O(n)$. After sieve by the distributed cross directory algorithm, a vertex is sent to

at most $\min(e, p)$ processors in Algorithm 2 instead of p in Algorithm 1. Thus the messages in Algorithm 2 will contain less non-zeros than those in Algorithm 1, which leads to a compression ratio $\sigma' < 1$.

C. Memory Consumption

For Algorithm 1, the memory consumption of f is $O(n)$; t_i and π_i are $O(n/p)$. So the memory consumption of each processor of Algorithm 1 is $O(n)$. Compared to Algorithm 1, Algorithm 2 adds f' and V_i , both of which costs $O(n)$ memory. So the memory consumption of Algorithm 2 is also bound to $O(n)$.

V. EXPERIMENTAL RESULTS

A. Experiment Setup

Our performance results is collected on a 6,144-core Intel Westmere based cluster, connected by Infiniband of 40 Gb/s. Each computing node has an SMP architecture with two Xeon X5650 CPUs (Westmere), which are connected through Intel QuickPath Interconnect (QPI) of 6.4 GT/s. The Xeon X5650 has six cores, each supports simultaneous multithreading (SMT) up to two threads. Each computing node has 24 GB DDR3-1333 RAM. In our experiments we used up to 6,144 cores, to run the experiment. We use gcc 4.3.4 and MPICH2 1.4.1 to compile our algorithms. The GNU OpenMP library is used for intra-node threading.

The input datasets are generated use synthetic kronecker graphs [17] in Graph500 benchmark which follow power law distributions: heavy tails for the degree distribution; small diameters; and densification and shrinking diameters over time. That means most of vertices has a small number of neighboring vertices and the graph is sparse. The graph size is determined by two parameters: “scale” and “edge factor”, where the total number of vertices N equals 2^{scale} , and the number of edges, $M = edgefactor * N$. The default edgefactor is set 16. Different from TOP500 [18], Graph500 use a new rate called traversed edges per second (TEPS) to measure BFS performance. Let *time* be the measured execution time for running BFS. Let m be the number of input edge tuples within the component traversed by the search, counting any multiple edges and self-loops. The normalized performance rate *traversed edges per second (TEPS)* is defined as: $TEPS = m/time$.

B. Experiment Results

Fig. 5 shows the weak scaling performance of our BFS algorithms. We run this experiment on a 6,144-core Intel Westmere based cluster, with one process per SMP node (12 cores for each node). For intra-node threading, we use the GNU OpenMP library. Algorithm 2 (DIR-WAH) outperforms all other algorithms and have the best scalability. DIR-WAH achieves 1.21E+10 TEPS at scale 33 with 6,144 cores, 2.24× faster than Algorithm 1 (BIT). As a comparison, we also implement another BFS algorithm with only compression (WAH in the plot) to see the effect of sieve: with only compression, WAH is 1.69× faster than BIT; with sieve, DIR-WAH is another 1.33× than WAH. The performance gap between DIR-WAH and BIT becomes wider as the number of CPU cores increases. This is because the larger the number of CPU cores used, the

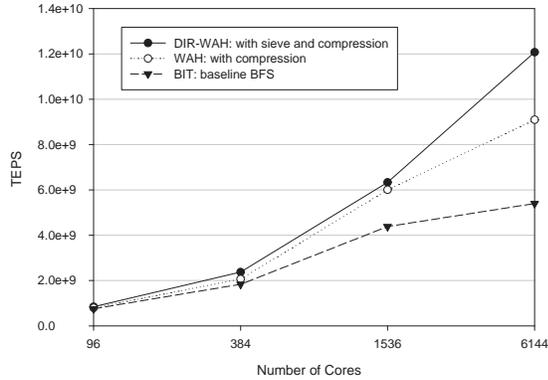


Fig. 5. Weak scaling performance of different BFS algorithms. The experiment use fixed problem size per computing node (each node has about 16M vertices).

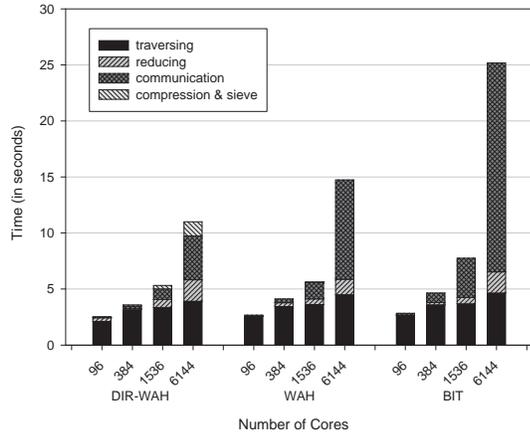


Fig. 6. Time breakdown of different BFS algorithms.

more distributed BFS algorithm will depend on communication, and the more benefits compression and sieve will bring. We will see the time breakdown in the next figure.

Fig. 6 is the time breakdown of the algorithms in Fig. 5: “traversing” time is the time spent on local computing; “reducing” time is the time spent on a MPI reduction operation to get the total vertex count of the frontier; “communication” time is the time spent on communication; “compression & sieve” time is the time spent on compression and sieve. For all three algorithms, as the number of CPU cores increases, “communication” times increase exponentially. For BIT, it accounts for as much as 73.2% of the total time for 6,144 cores. The “reducing” times also increase because the imbalance of a graph become more severe as the graph becomes larger; the local “traversing” times remain more or less the same because the problem size per computing node is fixed. At 6,144 cores, WAH reduces the “communication” time by 52.4% compared to BIT; DIR-WAH reduces the “communication” time by another 55.9% compared to WAH, achieving a total 79.0% reduction compared to BIT, from 18.6 seconds to 3.9 seconds. On one hand, the “compression & sieve” time of WAH

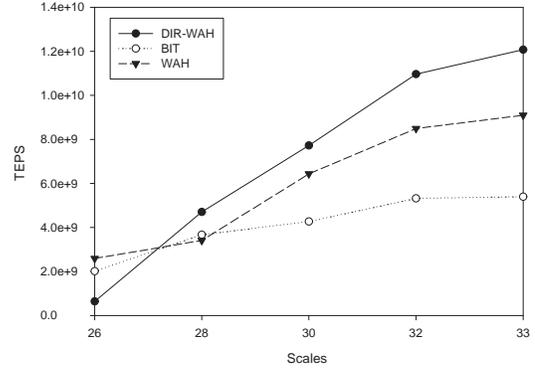


Fig. 7. Performance of different BFS algorithms at different scales. The experiment runs on 6,144 cores.

(only compression time is counted for WAH) at 6,144 cores is less than 0.1% of the total run time and not shown in the figure. This means the benefit of compression is at very little cost. On the other hand, the time of “compression & sieve” in DIR-WAH, — the computing time traded for bandwidth — accounts for 11.1% of the total. This is because Algorithm 2 (line 9) needs to copy the frontier for each process before sieve. This copying time is expensive because it is in direct proportion to the number of processes. Overall, comparing DIR-WAH to WAH (6,144 cores), sieve costs about 1.3 seconds but saves 5.0 seconds in communication.

Fig. 7 plots the performance of different BFS algorithms at different scales. The experiment runs on 6,144 cores. We can learn from this plot that the compression and sieve method favors larger messages. The size of messages will affect the results: at scale 26, DIR-WAH, WAH and BIT need to exchange 8MB bitmap globally using MPI collective communications; at scale 33, 1GB. DIR-WAH is the slowest when the scale is small, but it gradually catches up and surpasses all other algorithms when scale gets bigger.

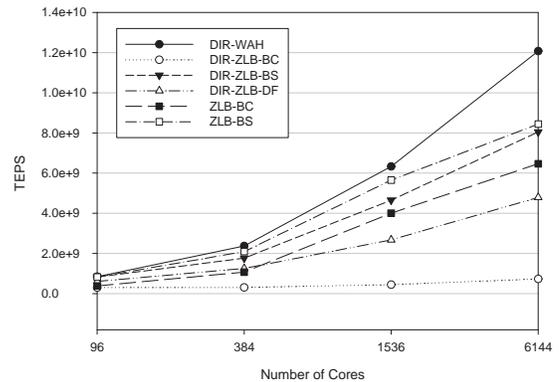


Fig. 8. Weak scaling performance result of BFS of different compression methods. The experiment use fixed problem size per computing node (each has about 16M vertices).

We compare the efficiency of different compression methods: WAH [11] and Zlib [13] with different compression levels. Here, DIR stands for with sieve, ZLB stands for Zlib, and

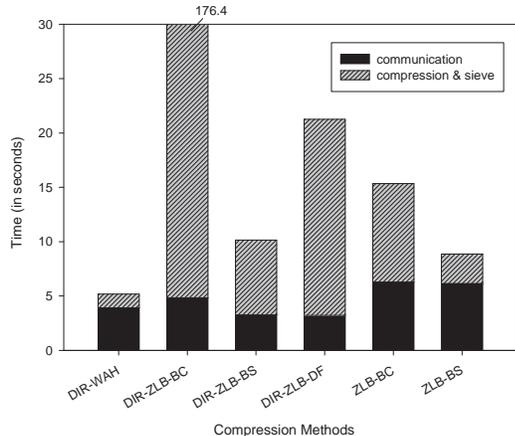


Fig. 9. Time profiling of different compression implementations.

WAH stands for WAH compression; DF, BS and BC stands for “default”, “best speed” and “best compression ratio” options for Zlib. Fig. 8 shows the weak scaling performance of BFS algorithms with different compression and sieve methods. BFS with Zlib best compression ZLB-BC is the slowest. With 6,144 cores, DIR-WAH provides the best performance, followed by ZLB-BS (69.9% of DIR-WAH), DIR-ZLB-BS (66.7%), ZLB-BC (53.5%), and DIR-ZLB-DF (39.7%) respectively. Fig. 9 shows the time breakdown of these algorithms. At scale 33 with 6,144 cores, DIR-ZLB-DF’s “communication” time is the smallest, 0.82× of DIR-WAH, followed by DIR-ZLB-BS (0.83×), DIR-ZLB-BC (1.23×), ZLB-BS (1.57×) and ZLB-BC (1.61×). Although DIR-ZLB-DF and DIR-ZLB-BS’s communication times are less than DIR-WAH, their “compression and sieve” times are 14.25× and 5.44× of DIR-WAH. So the overall performance of DIR-ZLB-DF and DIR-ZLB-BS are worse than DIR-WAH. For all three compression levels in Zlib we tested, default method, not the best compression method, provides the best compression ratio. In fact, the Zlib best compression method is not suited for bitmap compression: it is not only the slowest, but also provides the worst compression ratio.

VI. RELATED WORKS

Several different approaches are proposed to reduce the communication in distributed BFS. Yoo et al. [8] run distributed BFS on IBM Blue Gene/L with 32,768 nodes. Its high scalability is achieved through a set of memory and communication optimizations, including a two-dimensional partitioning of the graph to reduce communication overhead. Buluç and Madduri [7] improved Yoo et al.’s work by adding hybrid MPI/OpenMP programming to optimize computation on state-of-the-art multicore processors, and managed to run distributed BFS on a 40,000-core machine. The method of two-dimensional partitioning reduces the number of processes involved in collective communications. Our algorithm reduces the communication overhead in a different way: minimizing the size of messages with compression and sieve. Moreover, these two optimizations could be combined together to further reduce the communication cost in distributed BFS. A preliminary result is presented in Section VII to demonstrate its potential. Beamer et al. [10] use a hybrid top-down and

bottom-up approach that dramatically reduces the number of edges examined. The sample code in Graph500 [3] use bitmap (bit array) in communication, reducing its message size.

Benchmarks, algorithms and runtime systems for graph algorithms have gained much popularity in both academia and industry. Earlier works on Cray XMT/MTA [19], [20] and IBM Cyclops-64 [21] prove that both massive threads and fine-grained data synchronization improve BFS performance. Bader and Madduri [19] designed a fine-grained parallel BFS which utilizes the support for hardware threading and synchronization provided by MTA-2, and ensures that the graph traversal is load-balanced to run on thousands of hardware threads. Mizell and Maschhoff [20] discussed an improvement on Cray XMT. Using massive number of threads to hide latency has long been employed in these specialized multi-threaded machines. With the recent progress of multi-core and SMT, this technique can be popularized to more commodity users. Both core-level parallelism and memory-level parallelism are exploited by Agarwal et al. [22] for optimized parallel BFS on Intel Nehalem EP and EX processors. They achieved performances comparable to special purpose hardware like Cray XMT and Cray MTA-2 and first identified the capability of commodity multi-core systems for parallel BFS algorithms. Scarpazza et al. [23] use an asynchronous algorithm to optimize communication between SPE and SPU for running BFS on STI CELL processors. Leiserson and Schardl [24] use Cilk++ runtime model to implement parallel BFS. Cong et al. [25] present a fast PGAS implementation of distributed graph algorithms. Another trend is to use GPU for parallel BFS, for they provide massively parallel hardware threads, and are more cost-effective than the specialized hardware. Generally, GPUs are good at regular problems with contiguous memory accesses. The challenge of designing an effective BFS algorithm on GPU is to solve the imbalance between threads and to hide the cost of data transfer between CPU and GPU. There are several works [26], [27] working on this direction.

The cross directory proposed in this paper is inspired by Pinar and Hendrickson’s distributed directory [28] and Baker et al.’s assumed partition algorithm [29]. In their work, the communication pattern is dynamically determined and more general, while in our case, the communication parties are static. So we store the directory on both side of the communication, and update them synchronously on each side instead of send the updated directory over the network each time.

VII. CONCLUSION

The purpose of this paper is to reduce the communication cost in distributed breadth-first search (BFS), which is the bottleneck of the algorithm at scale. We found two problems in previous distributed BFS algorithms: first, their message formats are not condensed enough; second, broadcasting messages causes waste. We propose to reduce the message size by compressing and sieving. By compressing the messages, we reduce the communication time by 52.4%. By sieving the messages with a distributed directory before compression, we reduce the communication time by another 55.9%, achieving a total 79.0% reduction in communication time and 2.2× performance improvement over the baseline implementation.

For future works, we would like to combine our optimization of message size with other methods such as two-

dimensional partitioning [7] and hybrid top-down and bottom-up algorithm [10]. The potential of compression and sieve is clear. A preliminary optimization of the distributed BFS algorithm in combinational BLAS library [30], compressing the sparse vector using Zlib library, reduces the communication time by 41.9% and increases overall performance by 1.11 \times . By using compressed bitmap and adding sieve, we expect to further improve its performance.

ACKNOWLEDGEMENTS

This work is supported by National 863 Program(2009AA 01A129), the National Natural Science Foundation of China (60803030, 61033009, 60921002, 60925009, 61003062) and 973 Program (2011CB302500 and 2011CB302502). This material is based upon work supported by the U.S. Department of Energy, Office of Science, under contract number DE-AC02-06CH11357.

REFERENCES

- [1] D. A. Bader, "Petascale computing for large-scale graph problems," in *Proceedings of the 7th international conference on Parallel processing and applied mathematics*, ser. PPAM'07. Berlin, Heidelberg: Springer-Verlag, 2008, pp. 166–169.
- [2] A. Lumsdaine, "Challenges in parallel graph processing," *Parallel Processing Letters*, vol. 17, no. 1, p. 5, 2007.
- [3] "The Graph 500 List," <http://www.graph500.org/>.
- [4] B. Chazelle, "A minimum spanning tree algorithm with inverse-ackermann type complexity," *J. ACM*, vol. 47, pp. 1028–1047, November 2000.
- [5] U. Brandes, "A faster algorithm for betweenness centrality*," *The Journal of Mathematical Sociology*, vol. 25, no. 2, pp. 163–177, 2001.
- [6] B. Cherkassky, A. Goldberg, and T. Radzik, "Shortest paths algorithms: Theory and experimental evaluation," *Mathematical Programming*, vol. 73, pp. 129–174, 1996.
- [7] A. Buluç and K. Madduri, "Parallel breadth-first search on distributed memory systems," in *Proceedings of 2011 International Conference for High Performance Computing, Networking, Storage and Analysis*, ser. SC '11. New York, NY, USA: ACM, 2011, pp. 65:1–65:12.
- [8] A. Yoo, E. Chow, K. Henderson, W. McLendon, B. Hendrickson, and U. Catalyurek, "A scalable distributed parallel breadth-first search algorithm on bluegene/l," in *Proceedings of the 2005 ACM/IEEE conference on Supercomputing*, ser. SC '05. Washington, DC, USA: IEEE Computer Society, 2005, pp. 25–.
- [9] T. Suzumura, K. Ueno, H. Sato, K. Fujisawa, and S. Matsuoka, "Performance characteristics of graph500 on large-scale distributed environment," in *Proceedings of the 2011 IEEE International Symposium on Workload Characterization*, ser. IISWC '11. Washington, DC, USA: IEEE Computer Society, 2011, pp. 149–158.
- [10] S. Beamer, K. Asanović, and D. Patterson, "Direction-optimizing breadth-first search," in *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*, ser. SC '12. Los Alamitos, CA, USA: IEEE Computer Society Press, 2012, pp. 12:1–12:10.
- [11] K. Wu, E. J. Otoo, and A. Shoshani, "Optimizing bitmap indices with efficient compression," *ACM Trans. Database Syst.*, vol. 31, no. 1, pp. 1–38, Mar. 2006.
- [12] J. Ziv and A. Lempel, "A universal algorithm for sequential data compression," *Information Theory, IEEE Transactions on*, vol. 23, no. 3, pp. 337 – 343, may 1977.
- [13] "Zlib Home Page," <http://zlib.net>.
- [14] V. Kumar, *Introduction to Parallel Computing*, 2nd ed. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 2002.
- [15] R. Thakur, "Improving the performance of collective operations in MPICH," in *Recent Advances in Parallel Virtual Machine and Message Passing Interface. Number 2840 in LNCS, Springer Verlag (2003) 10th European PVM/MPI User's Group Meeting*. Springer Verlag, 2003, pp. 257–267.

- [16] J. Pješivac-Grbović, T. Angskun, G. Bosilca, G. E. Fagg, E. Gabriel, and J. J. Dongarra, "Performance analysis of mpi collective operations," *Cluster Computing*, vol. 10, pp. 127–143, June 2007.
- [17] J. Leskovec, D. Chakrabarti, J. Kleinberg, and C. Faloutsos, "Realistic, mathematically tractable graph generation and evolution, using kroecker multiplication," in *Knowledge Discovery in Databases: PKDD 2005*, ser. Lecture Notes in Computer Science, A. Jorge, L. Torgo, P. Brazdil, R. Camacho, and J. Gama, Eds. Springer Berlin / Heidelberg, 2005, vol. 3721, pp. 133–145.
- [18] "TOP500 Supercomputer Sites," <http://www.top500.org/>.
- [19] D. A. Bader and K. Madduri, "Designing multithreaded algorithms for breadth-first search and st-connectivity on the cray mta-2," in *Proceedings of the 2006 International Conference on Parallel Processing*, ser. ICPP '06. Washington, DC, USA: IEEE Computer Society, 2006, pp. 523–530.
- [20] D. Mizell and K. Maschhoff, "Early experiences with large-scale cray xmt systems," in *Proceedings of the 2009 IEEE International Symposium on Parallel & Distributed Processing*. Washington, DC, USA: IEEE Computer Society, 2009, pp. 1–9.
- [21] G. Tan, V. Sreedhar, and G. Gao, "Analysis and performance results of computing betweenness centrality on ibm cyclops64," *The Journal of Supercomputing*, vol. 56, pp. 1–24, 2011.
- [22] V. Agarwal, F. Petrini, D. Pasetto, and D. A. Bader, "Scalable graph exploration on multicore processors," in *Proceedings of the 2010 ACM/IEEE International Conference for High Performance Computing, Networking, Storage and Analysis*, ser. SC '10. Washington, DC, USA: IEEE Computer Society, 2010, pp. 1–11.
- [23] D. P. Scarpazza, O. Villa, and F. Petrini, "Efficient breadth-first search on the cell/be processor," *IEEE Trans. Parallel Distrib. Syst.*, vol. 19, pp. 1381–1395, October 2008.
- [24] C. E. Leiserson and T. B. Schardl, "A work-efficient parallel breadth-first search algorithm (or how to cope with the nondeterminism of reducers)," in *Proceedings of the 22nd ACM symposium on Parallelism in algorithms and architectures*, ser. SPAA '10. New York, NY, USA: ACM, 2010, pp. 303–314.
- [25] G. Cong, G. Almasi, and V. Saraswat, "Fast pgas implementation of distributed graph algorithms," in *Proceedings of the 2010 ACM/IEEE International Conference for High Performance Computing, Networking, Storage and Analysis*, ser. SC '10. Washington, DC, USA: IEEE Computer Society, 2010, pp. 1–11.
- [26] S. Hong, S. K. Kim, T. Oguntebi, and K. Olukotun, "Accelerating cuda graph algorithms at maximum warp," in *Proceedings of the 16th ACM symposium on Principles and practice of parallel programming*, ser. PPOPP '11. New York, NY, USA: ACM, 2011, pp. 267–276.
- [27] L. Luo, M. Wong, and W.-m. Hwu, "An effective gpu implementation of breadth-first search," in *Proceedings of the 47th Design Automation Conference*, ser. DAC '10. New York, NY, USA: ACM, 2010, pp. 52–55.
- [28] A. Pinar and B. Hendrickson, "Communication support for adaptive computation," in *Proc. SIAM Conf. on Parallel Processing for Scientific Computing*, 2001.
- [29] A. H. Baker, R. D. Falgout, and U. M. Yang, "An assumed partition algorithm for determining processor inter-communication," *Parallel Comput.*, vol. 32, pp. 394–414, June 2006.
- [30] A. Buluç and J. R. Gilbert, "The combinatorial blas: Design, implementation, and applications," in *International Journal of High Performance Computing Applications (IJHPCA)*, 2011.

GOVERNMENT LICENSE

The submitted manuscript has been created by UChicago Argonne, LLC, Operator of Argonne National Laboratory ("Argonne"). Argonne, a U.S. Department of Energy Office of Science laboratory, is operated under Contract No. DE-AC02-06CH11357. The U.S. Government retains for itself, and others acting on its behalf, a paid-up nonexclusive, irrevocable worldwide license in said article to reproduce, prepare derivative works, distribute copies to the public, and perform publicly and display publicly, by or on behalf of the Government.