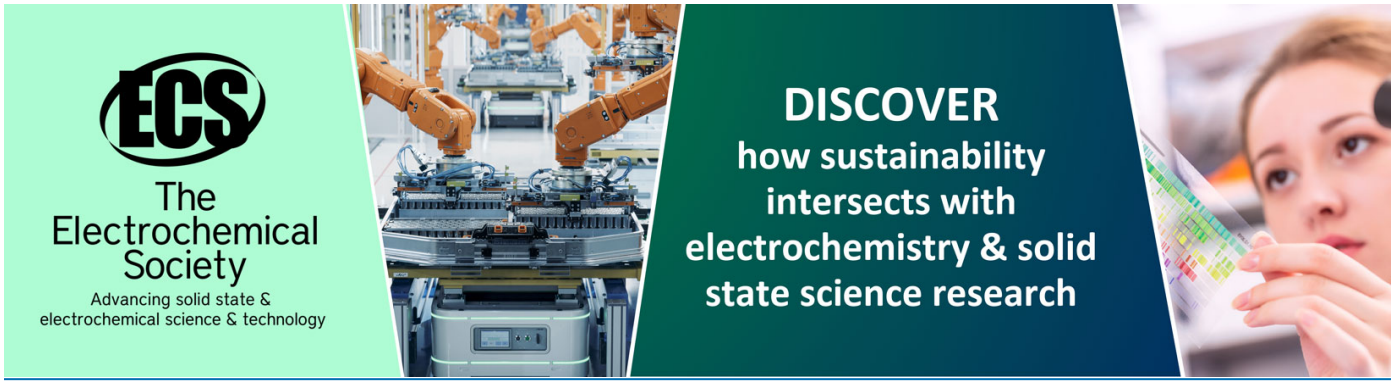
Optimization communication for BFS based on 1D-partition



Optimization communication for BFS based on 1D-partition

Chengyao Liu

National University of Defense Technology College of Computer, ChangSha, China 410073

Email: liuchengyaoi7@163.com

Abstract—Parallel Breadth First Search (BFS) is a famous algorithm in Graph500, a benchmark function for evaluating data-intensive applications on supercomputers. For parallel breadth first search (BFS) algorithms on large-scale distributed memory systems, the cost of communication is usually much higher than the arithmetic cost, which limits the scalability of the algorithm. However, the specific communication model of Graph500 brings challenges to computing in large-scale graphs. First, we use an adjustment method to delete redundant data in messages. Second, a data compression method was used to further reduce comm-unication. Evaluation results show that the performance of this method is more efficient than graph500 benchmark.

1. INTRODUCTION

Graph algorithm has been widely used in social interaction, email and telephone network communication data. Breadth-first search (BFS) is one of the most widely used graph-searching algorithms, which was established in 2010 by HPC community, and a new benchmark was proposed to rank supercomputers based on their performance on data-intensive applications [1]. Arithmetic and communication are two main parts of the algorithm. For distributed platform, communication often costs significantly more than arithmetic. For example, on a 256-node cluster, the baseline BFS algorithm in Graph500 spends nearly 60% time on communication on a Kronecker graph [2] with 64 billion edges (Figure 1). Therefore, optimizing the communi-cation part is extremely important in a distributed BFS algorithm.

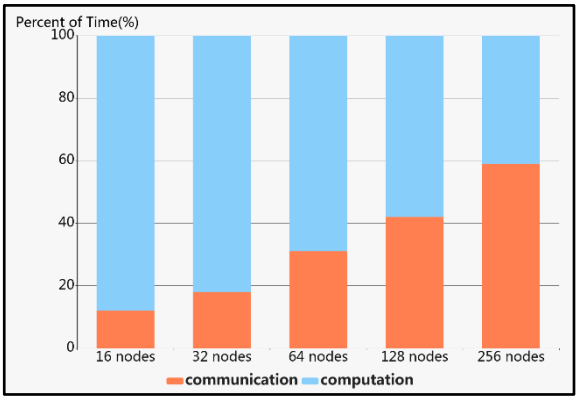


Figure.1. Proportion of time for distributed BFS implementation in weak scaling experiments

Here are the main contributions of this paper:

- The basic BFS implementation is analyzed in detail, and several shortcomings of the baseline BFS algorithm are summarized.

- We introduce a method of cutting the number of communication messages, and elaborates the advantages of this method in detail.
- Compared with the baseline BFS algorithm, 50.3% communication time has been reduced.

2. PARALLEL BFS ALGORITHM

In this section, we give an overview of the distribute BFS algorithm, a parallel level-synchronized BFS method.

2.1. Breadth-First Search Overview

Given a random "source vertex" s, Breadth-First Search (BFS) systematically explores the graph G to discover every vertex that is reachable from s. Let E and V refer to the edge and vertex sets of G, the number of edge and vertex are m = |E| and n = |V|. We assume that the graph is unweighted, which means each edge e $\in$ E is assigned a weight of unity.

A simple way to distribute the vertices and edges of a graph on a distributed storage system is to let each process have its own n/p vertices and all the outgoing edges of these vertices [3]. We call this division of the graph "one-dimensional division" because it is transformed into the one-dimensional decompo-sition of the incidence matrix corresponding to the graph.

$$\begin{bmatrix} A_1 \\ \hline A_2 \\ \hline \vdots \\ \hline A_p \end{bmatrix}$$

The distributed BFS with "one-dimensional division" partition proceeds as follows. At level l, a processor P_i has a set $f_{l,i}$, which is a set of frontier vertices owned by the processor P_i, The edge lists of the vertices in $f_{l,i}$ are merged to form a set of neighboring vertices, some of which will be owned by P_i itself, and some vertices will be owned by other processors P_j. For vertices in the latter case, messages are sent to P_j to add these vertices to the next level of frontier set. Each processor receives these sets of adjacent vertices and merges them to form$f_{i+1,j}$, a set of vertices owned by the processor. The processor may have marked some vertices in the previous iteration. In this case, the processor will ignore this message and all subsequent messages about these vertices. Algorithm 1 gives the pseudo-code for baseline distributed BFS on a cluster of multicore or multi-threaded processors.

Algorithm1: Baseline distributed BFS

Input: G (V, E), source vertex s.
Output: π: means the predecessor vertex on the shortest path
procedure BFS_1D (A, s)
1: $f(s) \leftarrow s$
2: **for** $l = 1$ to ∞ **do**
3: **if** $f = \emptyset$ **then** terminate main loop
4: **for** all processors P_i **in parallel do**
5: $f_l \leftarrow ALLGATHERV(f_{l,i}, P_i)$;
6: $t_{l,i} \leftarrow A_i \otimes f_l$;
7: $f_{l+1,i} \leftarrow t_{l,i} \odot \overline{\pi_l}; \pi_{l+1} \leftarrow \pi_l + t_{l,i}$;

2.2. Some Shortcomings

Our algorithm was implemented based on Graph 500. Input datasets are generated use synthetic Kronecker graph which follows power law distributions: small diameters and heavy tails for the degree distribution which means most of vertices has a small number of neighboring vertices, and the graph is sparse. The problem with using bitmap storage nodes is that bits must be reserved for each node. For

example, for a graph with 2^{28} vertices, there is only one initial node s in the first layer, so 32MB is still needed to store this node, where most of the other nodes' bits are 0. On the other hand, the cost of ALLGATHER collective communication is huge, but broadcast all vertices to all processors is not necessary in some cases because some processors need only a small fraction of the frontier information.

3. Optimization of Bfs

The problem of ALLGATHERV operation for distribute BFS is that it sends all frontier vertices to all the processors regardless whether a vertex is useful to them. in Figure.2, there is no direct edge between v_3 in P_2 and v_4, v_5 in P_3, so P_3 only needs v_2 from P_2, which means P_2 does not need to send the information of v_3 to P_3,

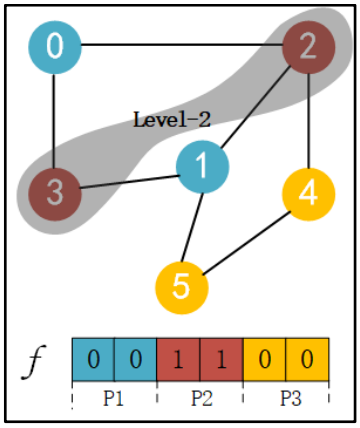


Figure.2. The value of frontier at the second level

$$f_2 = A \otimes f_1 = \begin{bmatrix} 0 & 0 & 1 & 1 & 0 & 0 \\ 0 & 0 & 1 & 1 & 0 & 1 \\ 1 & 1 & 0 & 0 & 1 & 0 \\ 1 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 1 \\ 0 & 1 & 0 & 0 & 1 & 0 \end{bmatrix} \otimes \begin{bmatrix} 1 \\ 0 \\ 0 \\ 0 \\ 0 \\ 0 \end{bmatrix} = \begin{bmatrix} 0 \\ 0 \\ 1 \\ 1 \\ 0 \\ 0 \end{bmatrix}$$

For a more detailed explanation, let's take a look at how the f matrix is calculated in this example. After the first round of calculation, we get $f_2 = [0,0,1,1,0,0]^T$,which means v_2 and v_3 are in the next level. In "one-dimensional division", we partition the matrix A into P blocks named $A_i (i \in [0, p))$. To facilitate the description of the calculation process, partition the matrix A_i into P parts named $A_{i,j} (j \in [0, p))$.

while calculating $f_3 = \begin{bmatrix} \sum_{j=1}^{3} A_{1,j} \otimes f_{2,j} \\ \sum_{j=1}^{3} A_{2,j} \otimes f_{2,j} \\ \sum_{j=1}^{3} A_{3,j} \otimes f_{2,j} \end{bmatrix}$,

$A_{3,2} \otimes f_{2,2} = \begin{bmatrix} 1 & 0 \\ 0 & 0 \end{bmatrix} \otimes \begin{bmatrix} 1 \\ 1 \end{bmatrix} = \begin{bmatrix} 1 \\ 0 \end{bmatrix}$, because the second column of $A_{3,2}$ are all 0, then the second element of product will always be 0, which means v_3 is useless for P_3 to calculate $f_{3,3}$ in next level. So, we can record this information and determine whether to send some vertices to other processor. A data structure [4] was define as follow: 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.

$V_{i,j} = (v_1, v_2, \cdots, v_n)$

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

For the example above，$V_{3,2}$ is initialized to $(1,0)$ at beginning, before traversing, $f_{2,2}$ is pruned from $(1,1)$ to $f_{2,2}\odot V_{3,2} = (1,1)^T\odot(1,0) = (1,0)$, so we need not send v_3 to P_3. The pruned matrix is established during initialization and used for checking whether a vertex should be transmitted.

$V_{i,1}$	$\cdots$	$V_{i,k}$	$\cdots$	$V_{i,p}$
$A_{i,1}$	$\cdots$	$A_{i,k}$	$\cdots$	$A_{i,p}$

Figure.3. The pruned matrix data structure in P_i

Although the modified method reduces the number of vertices sent, because of the representation method of the bitmap itself, it needs to maintain all the information bits in the set, so it does not really reduce the amount of information sent. Fortunately, through the pruning, we reduced the number of 1 in the bitmap, making the bitmap sparser, so that we can reduce the number of communications through further compression.

In the bitmap index, each bitmap often contains a large number of 0. This feature makes it very suitable for compression. A good bitmap index compression algorithm needs to achieve two goals: (1) increase the rate of bitwise logical operations; (2) reduce query response time. Therefore, the research in the industry mainly focuses on "Run-Length Encoding (RLE)" and "Delta-Encoding (DE)" algorithms. The main idea of RLE is to compress multiple consecutive identical values into the form of $number \times value$, for example: $000000 \rightarrow 6 \times (0)$.

Concise algorithm, the full name is “Compressed ‘n’ Composable Integer Set”, which is an improved version of WAH algorithm [5]. In the WAH method, excluding the highest bit and the second highest bit, the remaining 30 bits can represent up to $2^{30}-1$ consecutive all 0 or all 1 sequence, but there are few single sequences of this length in actual use scenarios. So, there is still room for optimization. The Concise algorithm is an improved WAH algorithm that efficiently compresses the low n bits in fill words. In the experiment, the compression performance of the Concise algorithm is about 50% higher than that of WAH.

In the Concise algorithm, word is divided into two categories: literal words and fill words, and the highest bit is set to 1 to represent literal words, that is, a mixed sequence of 0 and 1 follows. 0 means fill words, which means there are a large number of consecutive 0 or 1 sequences, all 0 sequences set the next high bit to 0, all 1 sequences set the next high bit to 1, and the next 5 bits called "Position Bits", it represents the position of a “flipped” bit, which means that 0 and 1 are reversed from which bit in the first 31-bitsgroup. As shown below:

Algorithm2: distributed BFS with optimization

$CSB()$and$u_CSB()$means compress and uncompress operation using concise bitmap algorithm.

procedure BFS_optimized (A, s)

1: $f(s) \leftarrow s$;
2: $f_1 \leftarrow ALLGATHERV(f_{1,i}, P_i)$;
3: **for** $l = 2$ to ∞ **do**
4: **if** $f = \emptyset$ **then** terminate main loop
5: **for** all processors P_i **in parallel do**
6: $t_{l,i} \leftarrow A_i \otimes f_l$;
7: $f_{l+1,i} \leftarrow t_{l,i} \odot \overline{\pi_l}; \pi_{l+1} \leftarrow \pi_l + t_{l,i}$;
8: $f_{l+1,i\rightarrow j} = f_{l+1,i} \odot V_{j,i}$;
9: $f^C_{l+1,i\rightarrow j} \rightarrow CSB(f_{l+1,i\rightarrow j})$;
10: $f^C_{l+1} \leftarrow ALLGATHERV(f^C_{l+1,i}, P_i)$;
11: **for** all processors P_j **in parallel do**
12: $f_{l+1,j} \leftarrow u\_CSB(f^C_{l+1,i\rightarrow j})$;

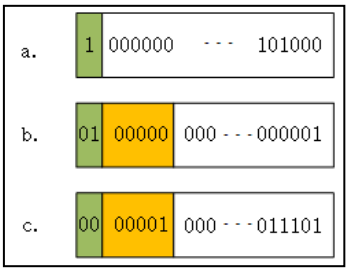


Figure.4. An example of concise bitmap algorithm

Taking a word length of 64 bits as an example, Word "a" is a literal word, except for the highest bit, the remaining 63 bits can represent the 63 integers [0,62]; Word "b" is a fill word, where the next highest bit is 1, which means a sequence of all 1, position bits is 00000, which means that the semantics of this fill word is consistent with WAH, and the last 58 bits is 1, which means that it contains two 63-bits groups, the first 63-bits group represents the 63 integers [62,124], and the second 63-bits group represents the [125,187], so word "b" represents the 126 integers in [62,187]; Word "c" is the fill word, the second highest bit is 0, which means all 0 sequence, position bits is 00001, which means $0 \rightarrow 1$ inverted from the lowest bit of the first 63-bits group, the last 58 bits are 11101, which means besides the first 63-bits group, there are 29×63bits groups, the maximum can represent $188(start) + 63 + 29 \times 63 - 1 = 2077$, that is, word "c" represents integers in the range [188, 2077].

4. Analysis Of Algorithm

Now we will analyze the cost of the optimized algorithm. We study the parallel BFS problem in the message passing model of distributed computing. The time it takes to send messages between any two processors can be modeled as $T(n) = \alpha + n\beta$, where α is the waiting time for each message Time (set-up time), independent of data size, β is the transmission time per byte, and n is the number of bytes transmitted [6]. For a given network, β is constant. In order to simplify the analysis, we assume that $n\beta \gg \alpha$, which means the bandwidth cost is much greater than the waiting time cost, because the data set of distributed BFS is large, so $T(n)$ will be determined by the bandwidth cost, that is to say, the communication cost is proportional to the message size n proportional. For graph $G(V, E)$, $G's$ diameter is d, p is the number of processors. When each processor needs to broadcast n/p size of message to other processors, the communication volume of both "Allgather" and "AlltoAll" are O(n). the bandwidth cost is $\frac{p-1}{p}n\beta$. BFS is a data-intensive algorithm, so we can assume that latency cost is much smaller than bandwidth cost, so its communication is bound to $O(|V|)$. Because algorithm will finish at level d, the communication volume of Algorithm1 is $d \times O(n)$.

In Algorithm2, assume C_i is the compression ratio of $CSB()$ function at level$i$, let $C = \frac{1}{d}\sum_{i=1}^{d}\frac{1}{C_i}$ be the compression ratio factor; After pruning, a vertex will be sent to at most $\min\left(\frac{|E|}{|V|}, p\right)$ processors instead of p in Algorithm1, which make a higher compression ratio. In addition, because of the characteristics of Kronecker Graphs, a more obvious compression effect will be obtained in the traversal of the first few and the last few levels.

5. Experimental Results

The experimental tests in this paper are all done on the computing nodes in the Tianhe-2 supercomputer system. Each computing node of Tianhe-2 is equipped with two Intel Ivy Bridge multi-core CPUs - Intel Xeon E5-2692 v2, with 12 processors integrated on each CPU and a data width of 64 bits. All cores are

tightly interconnected via a high-speed bus. Each processor can expand a virtual processor, so it can achieve parallel operation of up to 24 threads.

Our algorithms are based on Graph500 benchmark and dataset were generated by synthetic kronecker generator. The graph size is determined by "Scale" and "Edge factor", which means the graph owns $N = 2^{sacle}$ vertices and $M = edgefactor \times N$ edges, $edgefactor = 16$ as default. To save memory, we use a data structure calls "CSR" to store vertex information. In order to compare the performance of Graph 500 implementations across different architectures, A normalized evaluation standard $traversed\ edges\ per\ seco$
$nd(TEPS)$is proposed, and $TEPS = \frac{m}{time}$.

The default experiment condition is the parallel environment of 32 processors. Figure 5 shows the results of the experiment. We can see that under CSB optimization, the optimized algorithm can achieve a performance of 2.91E+9 in the case of 512 cores and scale=29. Compared with the 1.70E+9 of the baseline BFS algorithm, it is improved by 71%. In addition, if we adopt the dual optimization of CSB and pruning, we can further achieve a better performance of 3.62E+9, which is an increase of 113% compared to the baseline BFS. As the number of cores increases, the proportion of communication overhead will become larger and larger, this optimization will be more effective.

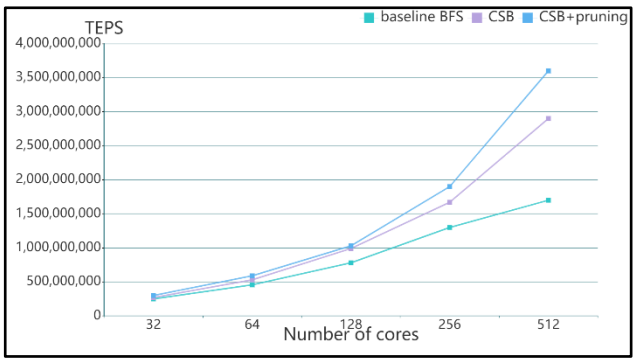


Figure.5. Performance of different BFS algorithms on scale=29

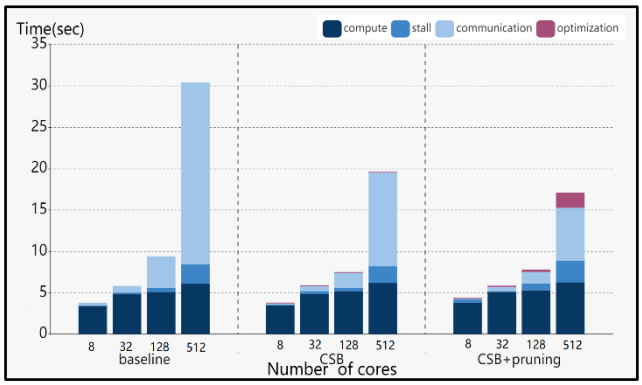


Figure.6. Time breakdown of three kinds of BFS

Figure.6 shows the breakdown of BFS algorithm and its optimized version. We fix the amounts of vertices processed by each core to be constant, with the increase in the number of cores, it can be seen that the proportion of communication is gradually increasing. For the basic BFS algorithm, it even accounts for 71.7% of the total time. In addition, because the graph division will be uneven when the number of cores is high, the stall time also has a growing trend. Because the number of vertices processed by each core remains the same, the total calculation time is similar. In the case of 512 cores, we can reduce the communication volume by 50.3% only through the concise bitmap method. If we combine

the concise bitmap and pruning methods, the communication volume can be further reduced to 36.6% of the basic BFS algorithm. The algorithm reduces lots of communication with only 10.2% of extra processing time. This cost is really worthwhile, with the scale of the graph increases, the optimization effect will be more obvious.

6. Conclusion And Future Work

The baseline algorithm provided in graph500 needs more optimization, especially the problem that distributed algorithm communication volume increases significantly with the number of cores, which has become the bottleneck of distribute BFS algorithm. Due to the characteristics of the kronecker graph, most of vertices have a small number of neighboring vertices, in the first few levels of traversal, the number of vertices is small, and there will be a lot of vacancies in bitmap storage, which allows us to have more compression space. By adopting the implementation method based on matrix multiplication, we can cut off the nodes without specified outgoing edges according to the matrix allocated by each processor, thereby reducing the amounts of communication need to be sent.

The current algorithm can be further improved in the future, such as combining direction optimization [7] and 2-D partition [8], this algorithm may achieve better performance.

References

[1] Jose, J., Potluri, S., Tomko, K., Panda, D.K.: Designing scalable graph500 benchmark with hybrid MPI+ OpenSHMEM programming models (2013).
[2] J. Leskovec, D. Chakrabarti, J. Kleinberg, and C. Faloutsos, "Realistic, mathematically tractable graph generation and evolution, using kronecker multiplication," in Conf. on Principles and Practice of Knowledge Discovery in Databases, 2005.
[3] IAndy Yoo, Edmond Chow, Keith Henderson,William McLendon, Bruce Hendrickson, Umit Catalyurek, “A Scalable Distributed Parallel Breadth-First Search Algorithm on BlueGene/L” Supersomputing 2005 (SC05), July 20, 2005.
[4] Lu, H., Tan, G., Chen, M., & Sun, N. (2014). Reducing Communication in Parallel Breadth-First Search on Distributed Memory Systems. 2014 IEEE 17th International Conference on Computational Science and Engineering.
[5] Alessandro Colantonio, Roberto Di Pietroa,"Concise: Compressed ’n’ Composable Integer Set", Information Processing Letters 110 (2010) pp.644–650.
[6] V. Kumar, *Introduction to Parallel Computing*, 2nd ed. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 2002.
[7] S. Beamer, K. Asanovi, and D. A. Patterson, “Searching for a parent instead of fighting over children: A fast breadth-first search implementation for graph500,” EECS Department, University of California, Berkeley, Tech. Rep. UCB/EECS-2011-117, Nov 2011.
[8] A. Buluc¸, 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.