LAWRENCE
LIVERMORE
NATIONAL
LABORATORY

UCRL-CONF-213752

A Scalable Distributed Parallel Breadth-First Search Algorithm on BlueGene/L

Andy Yoo, Edmond Chow, Keith Henderson, William McLendon, Bruce Hendrickson, Umit Catalyurek

July 20, 2005

Supersomputing 2005 (SC05)
Seattle, WA, United States
November 12, 2005 through November 18, 2005

A Scalable Distributed Parallel Breadth-First Search Algorithm on BlueGene/L

Andy Yoo[†] Edmond Chow[¶] Keith Henderson[†] William McLendon[‡]
Bruce Hendrickson[‡] Ümit Çatalyürek[§]

[†]Lawrence Livermore National Laboratory, Livermore, CA 94551
[‡]Sandia National Laboratories, Albuquerque, NM 87185
[¶]D. E. Shaw Research and Development, New York, NY 10036
[§]Ohio State Univeristy, Columbus, OH 43210

Abstract

Many emerging large-scale data science applications require searching large graphs distributed across multiple memories and processors. This paper presents a distributed breadth-first search (BFS) scheme that scales for random graphs with up to three billion vertices and 30 billion edges. Scalability was tested on IBM BlueGene/L with 32,768 nodes at the Lawrence Livermore National Laboratory. Scalability was obtained through a series of optimizations, in particular, those that ensure scalable use of memory. We use 2D (edge) partitioning of the graph instead of conventional 1D (vertex) partitioning to reduce communication overhead. For Poisson random graphs, we show that the expected size of the messages is scalable for both 2D and 1D partitionings. Finally, we have developed efficient collective communication functions for the 3D torus architecture of BlueGene/L that also take advantage of the structure in the problem. The performance and characteristics of the algorithm are measured and reported.

1 Introduction

Data science has gained much attention in recent years owing to a growth in demand for techniques to explore large-scale data in important areas such as genomics, astrophysics, and national security. Graph search plays an important role in analyzing large data sets many cases since the relationship between data objects is often represented in the form of graphs, such as semantic graphs [4, 11, 14]. Breadth-first search (BFS) is of particular importance among different graph search methods and is used widely used in numerous applications. The nature of the relationship between two vertices in a semantic graph, for example, can be determined by the shortest path between them using BFS.

Searching very large graphs with billions of vertices and edges, however, poses challenges mainly due to the vast search space imposed by the large graphs. Especially, it is often impossible to store such large graphs in the main memory of a single computer. This makes the traditional PRAM-based parallel BFS algorithms [5, 7, 8, 10] unusable and calls for distributed parallel BFS algorithms where the computation moves to the processor owning the data. Obviously, the scalability of the distributed BFS algorithm for very large graphs becomes a critical issue, since the demand for local memory and inter-processor communication increases as the graph size increases. In this paper, we propose a scalable and efficient distributed BFS scheme that is capable of handling graphs with billions of vertices and edges. In this research we consider Poisson random graphs, where the probability of any two vertices being connected is equal.

We achieve high scalability through a set of clever memory and communication optimizations. First, a two-dimensional (2D) graph partitioning [13, 12, 3] is used instead of more conventional one-dimensional (1D) partitioning. With the 2D partitioning, the number of processes involved in collective communications is $O(\sqrt{P})$ in contrast to $O(P)$ of 1D partitioning, where P is the total number of processors. Next, we derive the bounds on the length of messages for Poisson random graphs. We show that given a random graph with n vertices, the expected message length is $O(n/P)$. This allows us to manage the local memory more efficiently and improve the scalability. Finally, for the collective operations, we explore the use of using reduce-scatter (where the reduction operation is set-union) rather than straightforward use of all-to-all. We implement reduce-scatter using point-to-point communications with an algorithm suitable for the BlueGene/L torus network.

Our BFS scheme exhibits good scalability as it scales to a graph with 3.2 billion vertices and 32 billion edges on a BlueGene/L system with 32,768 nodes. To the best of our knowledge, this is the largest explicitly formed graph ever explored by a distributed algorithm.

This paper is organized as follows. Section 2 describes proposed distributed BFS algorithm. The optimizations of the BFS scheme is discussed in Section 3. The experimental results are presented in Section 4, followed by concluding remarks and directions for future work in Section 5.

2 Proposed Distributed BFS Algorithm

In this section, we present the distributed BFS algorithm with 1D and 2D partitionings. The proposed algorithm is a level-synchronized BFS algorithm that proceeds level by level, starting with a source vertex, where the level of a vertex is defined as its graph distance from the source. In the following, we use P to denote the number of processors, n to denote the number of vertices in a Poisson random graph, and k to denote the average degree. The P processors are mapped to a two-dimensional logical processor array and R and C denote the row and column stripes of the processor array, respectively. We consider only undirected graphs in this paper.

2.1 Distributed BFS with 1D Partitioning

A 1D partitioning of a graph is a partitioning of its vertices such that each vertex and the edges emanating from it are owned by one processor[1]. The set of vertices owned by a processor is also called its *local vertices*. The following illustrates a 1D P-way partitioning using the adjacency matrix, A, of the graph, symmetrically reordered so that vertices owned by the same processor are contiguous.

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

The subscripts indicate the index of the processor owning the data. The edges emanating from vertex v form its *edge list* and is the list of vertex indices in row v of the adjacency matrix. For the partitioning to be balanced, each processor should be assigned approximately the same number of vertices and emanating edges.

A distributed BFS with 1D partitioning proceeds as follows. At each level, each processor has a set F which is the set of frontier vertices owned by that processor. The edge lists of the vertices in F are merged to form a set N of neighboring vertices. Some of these vertices will be owned by the same processor, and some will be owned by other processors. For vertices in the latter case, messages are sent to other processors (neighbor vertices are sent to their owners) to add these vertices to their frontier set for the next level. Each processor receives these sets of neighbor vertices and merges them to form $\bar{N}$, a set of vertices which the

[1]We assume that only one process is assigned to a processor and use process and processor interchangeably.

processor owns. The processor may have marked some vertices in $\bar{N}$ in a previous iteration. In that case, the processor will ignore this message, and all subsequent messages regarding those vertices.

Algorithm 1 describes the distributed breadth-first expansion using the 1D partitioning, starting with a vertex v_s. In the algorithm, every vertex v becomes labeled with its level, $L_{v_s}(v)$, which denotes its graph distance from v_s. The data structure $L_{v_s}(v)$ is also distributed so that a processor only stores L for its local vertices.

2.2 Distributed BFS with 2D partitioning

A 2D partitioning of a graph is a partitioning of its edges such that each edge is owned by one processor. In addition, the vertices are also partitioned such that each vertex is owned by one processor. A process stores some edges incident on its owned vertices, and some edges that are not. This partitioning can be illustrated using the adjacency matrix, A, of the graph, symmetrically reordered so that vertices owned by the same processor are contiguous.

$$\begin{bmatrix} A^{(1)}_{1,1} & A^{(1)}_{1,2} & \cdots & A^{(1)}_{1,C} \\ A^{(1)}_{2,1} & A^{(1)}_{2,2} & \cdots & A^{(1)}_{2,C} \\ \vdots & \vdots & \ddots & \vdots \\ A^{(1)}_{R,1} & A^{(1)}_{R,2} & \cdots & A^{(1)}_{R,C} \\ & & \vdots & \\ A^{(C)}_{1,1} & A^{(C)}_{1,2} & \cdots & A^{(C)}_{1,C} \\ A^{(C)}_{2,1} & A^{(C)}_{2,2} & \cdots & A^{(C)}_{2,C} \\ \vdots & \vdots & \ddots & \vdots \\ A^{(C)}_{R,1} & A^{(C)}_{R,2} & \cdots & A^{(C)}_{R,C} \end{bmatrix}$$

Here, the partitioning is for $P = R \cdot C$ processors, logically arranged in a $R \times C$ processor mesh. We will use the terms *processor-row* and *processor-column* with respect to this processor mesh. In the 2D partitioning above, the adjacency matrix is divided into $R \cdot C$ block rows and C block columns. The notation $A^{(*)}_{i,j}$ denotes a block owned by processor (i, j). Each processor owns C blocks. To partition the vertices, processor (i, j) owns the vertices corresponding to block row $(j - 1) \cdot R + i$. For the partitioning to be balanced, each processor should be assigned approximately the same number of vertices and edges. The conventional 1D partitioning is equivalent to the 2D partitioning with $R = 1$ or $C = 1$.

For the 2D partitioning, we assume that the edge list for a given vertex is a *column* of the adjacency matrix. Thus each block in the 2D partitioning contains *partial* edge lists. In BFS using this partitioning, each processor has a set F which is the set of frontier vertices owned by that processor. Consider a vertex v in F. The owner of v sends messages to other processors in its processor-column to tell them that v is on the frontier, since any of these processors may contain partial edge lists for v. We call this communication step the *expand* operation. The partial edge lists on each processor are merged to form the set N, which are potential vertices on the next frontier. The vertices in N are then sent to their owners to potentially be added to the new frontier set on those processors. With 2D partitioning, these owner processors are in the same processor row. This communication step is referred to as the *fold* operation. The communication step in the 1D partitioning (steps 8–13 in the Algorithm 1) is the same as the fold operation in the 2D partitioning. The advantage of 2D partitioning over 1D partitioning is that the processor-column and processor-row communications involve R and C processors, respectively; for 1D partitioning, all P processors are involved in the communication operation. Algorithm 2 describes the proposed distributed BFS algorithm using 2D partitioning. Steps 7–11 and 13–18 correspond to expand and fold operations, respectively.

Algorithm 1 Distributed Breadth-First Expansion with 1D Partitioning

1: Initialize $L_{v_s}(v) = \begin{cases} 0, & v = v_s, \text{ where } v_s \text{ is a source} \\ \infty, & \text{otherwise} \end{cases}$
2: **for** $l = 0$ to ∞ **do**
3: $F \leftarrow \{v \mid L_{v_s}(v) = l\}$, the set of local vertices with level l
4: **if** $F = \emptyset$ for all processors **then**
5: Terminate main loop
6: **end if**
7: $N \leftarrow \{$neighbors of vertices in F (not necessarily local)$\}$
8: **for all** processors q **do**
9: $N_q \leftarrow \{$vertices in N owned by processor $q\}$
10: **Send** N_q to processor q
11: **Receive** $\bar{N}_q$ from processor q
12: **end for**
13: $\bar{N} \leftarrow \bigcup_q \bar{N}_q$ (The $\bar{N}_q$ may overlap)
14: **for** $v \in \bar{N}$ and $L_{v_s}(v) = \infty$ **do**
15: $L_{v_s}(v) \leftarrow l + 1$
16: **end for**
17: **end for**

Algorithm 2 Distributed Breadth-First Expansion with 2D Partitioning

1: Initialize $L_{v_s}(v) = \begin{cases} 0, & v = v_s, \text{ where } v_s \text{ is a source} \\ \infty, & \text{otherwise} \end{cases}$
2: **for** $l = 0$ to ∞ **do**
3: $F \leftarrow \{v \mid L_{v_s}(v) = l\}$, the set of local vertices with level l
4: **if** $F = \emptyset$ for all processors **then**
5: Terminate main loop
6: **end if**
7: **for all** processors q in this processor-column **do**
8: **Send** F to processor q
9: **Receive** $\bar{F}_q$ from processor q (The $\bar{F}_q$ are disjoint)
10: **end for**
11: $\bar{F} \leftarrow \bigcup_q \bar{F}_q$
12: $N \leftarrow \big\{$neighbors of vertices in $\bar{F}$ using edge lists on this processor$\big\}$
13: **for all** processors q in this processor-row **do**
14: $N_q \leftarrow \{$vertices in N owned by processor $q\}$
15: **Send** N_q to processor q
16: **Receive** $\bar{N}_q$ from processor q
17: **end for**
18: $\bar{N} \leftarrow \bigcup_q \bar{N}_q$ (The $\bar{N}_q$ may overlap)
19: **for** $v \in \bar{N}$ and $L_{v_s}(v) = \infty$ **do**
20: $L_{v_s}(v) \leftarrow l + 1$
21: **end for**
22: **end for**

In the expand operation, processors send the indices of the frontier vertices that they own to other processors. For dense matrices [6] (and even in some cases for sparse matrices [9]), this operation is traditionally implemented with an all-gather collective communication, since all indices owned by a processor need to be sent. For BFS, this is equivalent to the case where all vertices are on the frontier. This communication is not scalable as the number of processors increases.

For sparse graphs, however, it is advantageous to only send vertices on the frontier, and to only send to processors that have non-empty partial edge lists corresponding to these frontier vertices. This operation can now be implemented by an all-to-all collective communication or alternatively by multiple broadcasts. In the 2D case, each processor needs to store information about the edge lists of other processors in its processor-column. The storage for this information is proportional to the number of vertices owned by a processor, and therefore it is scalable. We will show in Section 3 that for Poisson random graphs, the message lengths are scalable when communication is performed this way.

The fold operation is traditionally implemented for dense matrix computations as an all-to-all communication. An alternative is to implement the fold operation as a reduce-scatter operation. In this case, each processor receives $\bar{N}$ directly and line 18 of the Algorithm 1 is not necessary. The reduction operation, which occurs within the reduction stage of the operation, is a set-union and eliminates all the duplicate vertices.

2.3 Bi-directional BFS

The BFS algorithm described above is uni-directional in that the search starts from the source and continues until it reaches the destination or all the vertices in the graph are visited. The BFS algorithm can be implemented in a bi-directional fashion as well. In a bi-directional search, the search starts from both source and destination vertices and continues until a path connecting the source and destination is found. An obvious advantage of the bi-directional search is that the frontier of the search remains small compared to the uni-directional case. This reduces the communication volume as well as the number of memory accesses, significantly improving the performance of the search. 1D or 2D partitionings can be used in conjunction with the bi-directional BFS. For additional details, see [16].

2.4 Implementation

2.4.1 Storage of edge lists

In the 2D partitioning, each processor owns $O(n/P)$ vertices but contains edge lists for $O(n/C)$ vertices, which is asymptotically larger than $O(n/P)$. For Poisson random graphs, however, the expected number of non-empty edge lists is $O(n/P)$. This can be demonstrated by examining the case where P is large.

Each edge list has on average k entries, and these are distributed among R processes randomly. As $R \to \infty$, it is increasingly unlikely that two of these entries will be in a given edge list. In the limit, then, the length of every edge list is either zero or one, and there is one edge list per edge. The total number of edges in the graph is nk, so we expect nk non-empty edge lists distributed over P processors.

When P is smaller some edge lists will be longer, so there are fewer non-empty edge lists in that case. An upper bound on the number of edge lists in the graph is nk, so the expected number of non-empty edge lists on a given processor is $O(n/P)$. Thus it is necessary not to index all edge lists, but only the non-empty ones.

Similarly, the number of unique vertices appearing in all edge lists on a processor is $O(n/P)$. This is demonstrated by noting that we could store edge lists for matrix rows rather than columns. The above analysis applies here, so the number of non-empty row edge lists would be $O(n/P)$ as well. Each non-empty row edge list on a processor corresponds to a unique entry in the column edge lists on that processor.

2.4.2 Local indexing

The index of a vertex in the original numbering of the graph vertices is called the *global* index of the vertex. To facilitate the storage of data associated with local vertices, L, the global index of a local vertex is mapped

to a *local* index. This local index is used to access L and other arrays. The mapping from global indices to local indices is accomplished through hashing. table.

Each processor stores three such mappings. The vertices owned by a processor are indexed locally, as noted above. Moreover, a mapping is generated for any vertices with non-empty edge lists on a processor. Lastly, a mapping is generated for any vertices appearing in an edge list on a processor. As described previously, the latter two mappings include $O(n/P)$ vertices each. The first mapping is obviously $O(n/P)$.

2.4.3 Sent neighbors

Each processor keeps track of which neighbor vertices it has already sent. Once a neighbor vertex is sent, it may be encountered again, but it never needs to be sent again. The storage required for this optimization is proportional to the number of unique vertices that appear in edge lists on a given processor, which is $O(n/P)$ as demonstrated above.

3 Optimizations for Scalability

It was shown in the previous section that the 2D partitioning reduces the number of processors involved in collective communications. In this section we show how the BFS algorithm can be further optimized to enhance its scalability.

3.1 Bounds on message buffer length and memory optimization

A major factor limiting the scalability of our distributed BFS algorithm is the fact that the length of message buffers used in all-to-all collective communications grows as the number of processors increases. A key to overcoming this limitation is to use message buffers of fixed length. In the following, we derive the upper bounds on the length of messages in our BFS algorithm for Poisson random graphs. Recall that we define n as the number of vertices in the graph, k as the average degree, and P as the number of processors. We assume P can be factored as $P = R \times C$, the dimensions of the processor mesh in the 2D case. For simplicity, we further assume that n is a multiple of P and that each processor owns n/P vertices.

Let A' be the matrix formed by any m rows of the adjacency matrix of the random graph. We define the useful quantity

$$\gamma(m) = 1 - \left(\frac{n-1}{n}\right)^{mk}$$

which is the probability that a given column of A' is nonzero. The quantity mk is the expected number of edges (nonzeros) in A'. The function γ approaches mk/n for large n and approaches 1 for small n.

For distributed BFS with 1D partitioning, processor i owns the A_i part of the adjacency matrix. In the communication operation, processor i sends the indices of the neighbors of its frontier vertices to their owner processors. If all vertices owned by i are on the frontier, the expected number of neighbor vertices is

$$n \cdot \gamma(n/P) \cdot (P-1)/P.$$

This communication length is nk/P in the worst case, which is $O(n/P)$. The worst case viewed another way is equal to the actual number of nonzeros in A_i; every edge causes a communication. This worst case result is independent of the graph.

In 2D expand communication, the indices of the vertices on the frontier set are sent to the $R-1$ other processors in the processor-column. In the worst case, if all n/P vertices owned by a processor are on the frontier (or if all-gather communication is used and all n/P indices are sent) the number of indices sent by the processor is

$$\frac{n}{P}(R-1)$$

which increases with R and thus the message size is not controlled when the number of processors increases.

The maximum expected message size is bounded as R increases, however, if a processor only sends the indices needed by another processor (all-to-all communication, but requires knowing which indices to send). A processor only sends indices to processors that have partial edge lists corresponding to vertices owned by it. The expected number of indices is

$$\frac{n}{P} \cdot \gamma(n/R) \cdot (R-1).$$

The result for the 2D fold communication is similar:

$$\frac{n}{P} \cdot \gamma(n/C) \cdot (C-1).$$

These two quantities are also $O(n/P)$ in the worst case. Thus, for both 1D and 2D partitionings, the length of the communication from a single processor is $O(n/P)$, proportional to the number of vertices owned by a processor. Once an upper bound on the message size is determined, we can use message buffers of fixed length independent of the number of processors used.

3.2 Optimization of collectives for BlueGene/L

It can be deduced from the equations presented in Section 3.1 that the expected message size approaches $\frac{n}{P} \cdot k$ for large n. This implies that all-to-all communication may not be used for very large graphs with high average degree, due to the memory constraint. To limit the size of message buffers to a fixed length, independent of k, the collectives must be implemented based on point-to-point communication. We have developed scalable collectives using point-to-point communications specifically designed for the BlueGene/L. Here, we attempt to reduce the number of point-to-point communications, taking advantage of the high-bandwidth torus interconnect of the BlueGene/L and to reduce the volume of messages transmitted.

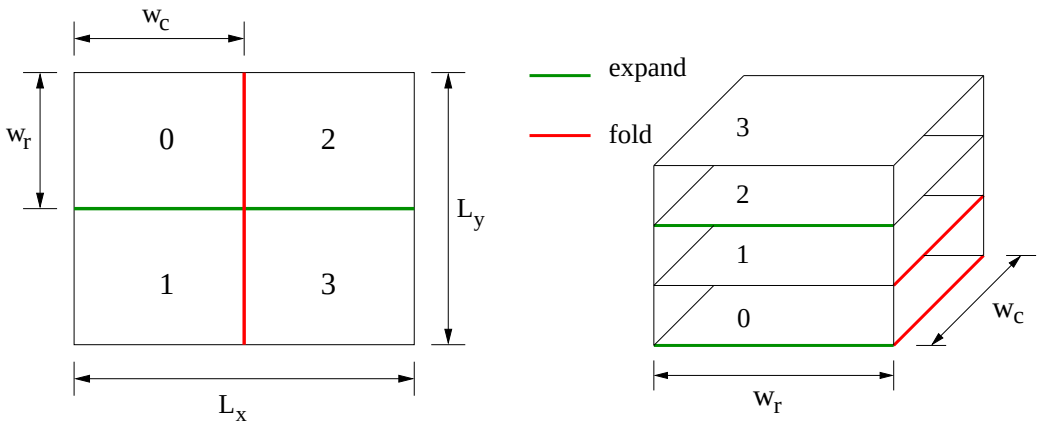
3.2.1 Task mapping

Our BFS scheme assumes that a given graph is distributed to a two-dimensional logical (processor) array. This logical processor array is then mapped to a three-dimensional torus of the BlueGene/L. Figure 1 illustrates this mapping. In this example, an $L_x \times L_y$ logical processor array is mapped to a $w_c \times w_r \times 4$ torus. The given $L_x \times L_y$ logical processor array is first divided into a set of $w_c \times w_r$ *planes*, and then each plane is mapped to the torus in such a way that the planes in the same column are mapped to *adjacent* physical planes as shown in Figure 1.b. With this mapping the expand operation is performed by those processors in the same column of adjacent physical planes. On the other hand, the processors performing fold operation are not in adjacent planes. These processors form processor grids on multiple planes on which expand and fold operations are performed as indicated by green and red lines in Figure 1.b. We concentrate on improving the performance of the collectives on these grids.

3.2.2 The optimization of the collectives

Basically, the optimized collectives for BlueGene/L are implemented using ring communication, a point-to-point communication that naturally works very well on a torus interconnect, to make them scalable. We improve the performance of these collectives by shortening the diameter of the ring in our optimization. In this scheme, the collective communications are performed in two phases. The idea is to divide the processors in the ring into several groups and perform the ring communication within each group in parallel. To ensure that processors in a group can receive and process messages from the processors in all other groups, processors in each group initially send messages targeted to other processor groups. A processor sends messages to only one processor in each group in this stage (phase 1). These messages will eventually be received by all the processors in the targeted group during the ring communication (phase 2). The processes are mapped to processors in such a way that the processors in each group form a physical ring with (wraparound edges).

The fold operation is implemented as reduce-scatter in our optimization, where the reduction operation is set-union. That is, all the messages are scanned while being transmitted to ensure that the messages



(a) An $L_x \times L_y$ logical processor array

(b) Mapping on $w_c \times w_r \times 4$ torus

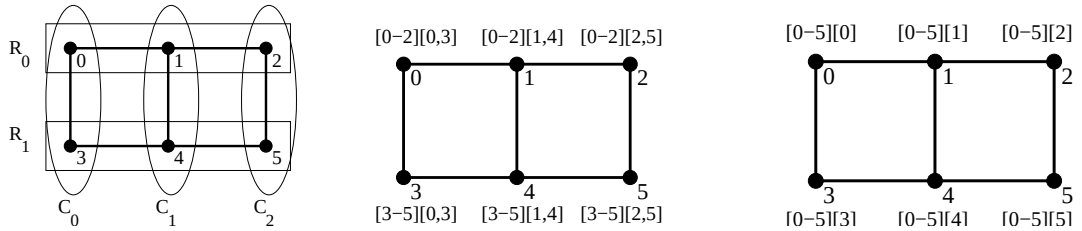
Figure 1: Mapping of the $L_x \times L_y$ logical processor array to $w_c \times w_r \times 4$ torus.

do not contain duplicate vertices. This union operation reduces the total message volume and therefore improves the communication performance. In addition, the decrease in the message volume reduces the memory accesses for processing the received vertices. The proposed communication scheme is similar to the all-to-all personalized communication technique proposed in [17] but differs in that our scheme performs the set-union operation on transmitted messages.

In this scheme, the processors in the same rows and columns of a processor grid are grouped together. In phase 1, all the processors in the same row group exchange messages in a ring fashion. In this row-wise communication, a processor combines messages for all the processors in each column group and sends them to the processors in the same row. When a process adds its vertices to a received message, it only adds those that are not already in the message. Each process has a set of vertices from all the processes in its row group (including itself) to the processes in its column group after the phase 1. These vertices are then distributed to appropriate processes in its column group in phase 2 using point-to-point communication to complete the fold operation.

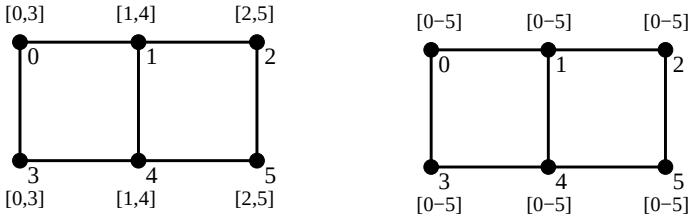
This is illustrated in Figure 2. In this example, the fold operation is performed on a 2×3 processor grid. The processors are grouped in two row groups and three column groups as shown in Figure 2.a. After phase 1, each processor in a row group contains the messages from all the processes in the row group to the processes in the column group that the processor belongs to (Figure 2.b). After these messages are exchanged among the column processors in phase 2, each processor has received all the messages destined to the processor (Figure 2.c).

The expand operation is a simpler variation of the fold operation. The difference is that each processor sends the same message to all the other processors on a processor grid. We describe this using an example, where an expand operation is performed on a 2×3 processor grid that is depicted in Figure 2.a. In the first phase, the processors in the same column group send messages to each other. Therefore, all the messages to be sent to a row processor group have received by the processors in the row group after phase 1, as shown in Figure 3.a. These messages are then circulated in row-wise ring communications in phase 2. After the phase 2, each processor has received messages from all other processors (Figure 3.b). The time complexity of both fold and expand operations is $O(m + n)$ for an $m \times n$ processor grid.



(a) A 2×3 processor array (b) Messages received after phase 1 (c) Messages received after phase 2

Figure 2: A fold operation on a 2×3 processor grid (The notation [S][R] denotes a set of messages sent by processors in group S to processors in group R. The sending and receiving groups are represented as a range of comma-separated list of processors.).



(a) Messages received after phase 1 (b) Messages received after phase 2

Figure 3: An expand operation on a 2×3 processor grid (The notation [S] denotes a set of messages sent by processors in group S to the receiving processor. The receiving processor is not specified in the notation for clarity).

4 Performance Evaluation

This section presents experimental results for the distributed BFS for Poisson random graphs. We have conducted most of the experiments on IBM BlueGene/L [1]. We also have conducted some experiments on MCR [15], a large Linux cluster with Quadrics interconnect, for the comparative study of the performance of the proposed BFS algorithm on a more conventional computing platform.

4.1 Overview of BlueGene/L system

BlueGene/L is a massively parallel system developed by IBM jointly with Lawrence Livermore National Laboratory [1]. BlueGene/L comprises 65,536 compute nodes (CNs) interconnected as a $64 \times 32 \times 32$ 3D torus. Each CN contains two 32-bit PowerPC 440 processors, each with dual floating-point units. The peak performance of each CN is 5.6 GFlops/s running at 700 MHz, allowing the BlueGene/L system to achieve the total peak performance of 360 TFlops/s. BlueGene/L is also equipped with 512 MB of main memory per CN (and 32 TB of total memory).

Each CN contains six bi-directional torus links directly connected to nearest neighbors in each of three dimensions. At 1.4 Bbits/s per direction, the BlueGene/L system achieves the bisection bandwidth of 360 GB/s per direction. The CNs are also connected by a separate tree network in which any CN can be a root. The torus network is used mainly for communications in user applications and supports point-to-point as well as collective communications. The tree network is also used for CNs to communicate with I/O nodes. It can be also used for some collectives such as broadcast and reduce.

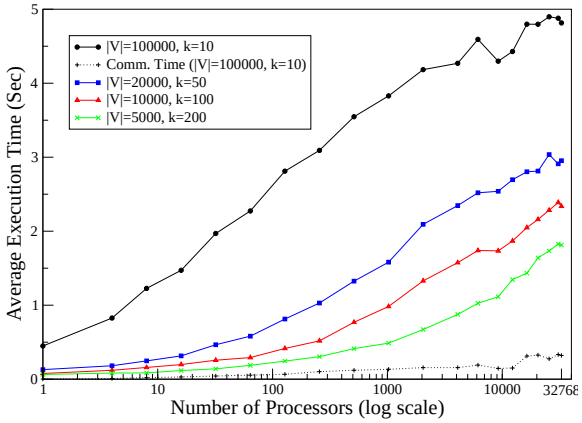
A CN runs on a simple run-time system called compute node kernel (CNK) that has a very small memory footprint. The main task of the CNK is to load and execute user applications. The CNK does not provide virtual memory and multi-threading support and provides a fixed-size address space for a single user process. Many conventional system calls including I/O requests are function-shipped to a separate I/O node which runs on a conventional Linux operating system.

4.2 Performance Study Results

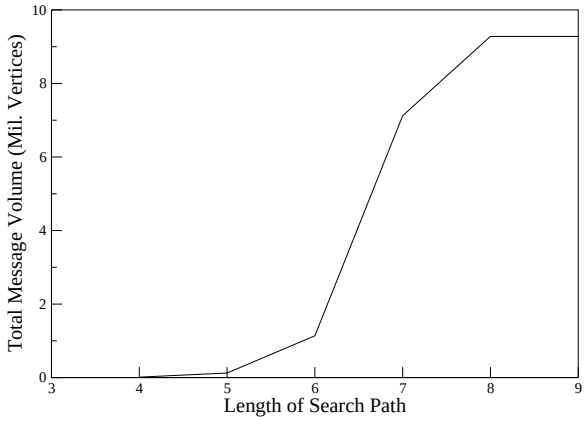
First, we measured the scalability of the proposed BFS algorithm in weak scaling experiments on a 32768-node BlueGene/L system and present the results in Figure 4. In a weak scaling study, we increase the global problem size as the number of processors increases. Therefore, the size of local problem (i.e., the number of vertices) assigned to each processor remains constant. The local problem size used in these experiments is 100000 vertices and the average degree of the graphs varies from 10 to 200.

The scalability of our BFS scheme is clearly demonstrated in Figure 4.a. The largest graph used in this study has 3.2 billion vertices and 32 billion edges. To the best of our knowledge, this is the largest graph ever explored by a distributed graph search algorithm. Such high scalability of our scheme can be attributed to the fact that the length of message buffers used in our algorithm does not increase as the size of graphs grows. In Figure 4.a it is also revealed that the communication time is very small compared to the computation time (in the case with local problem size of 100000 vertices and the average degree of 10). This indicates that our algorithm is highly memory-intensive as it involves very little computation. Profiling of the code has confirmed that it spends most of its time in a hashing function that is invoked to process the received vertices. The communication time for other graphs with different degrees is also very small and is omitted in the figure for clarity.

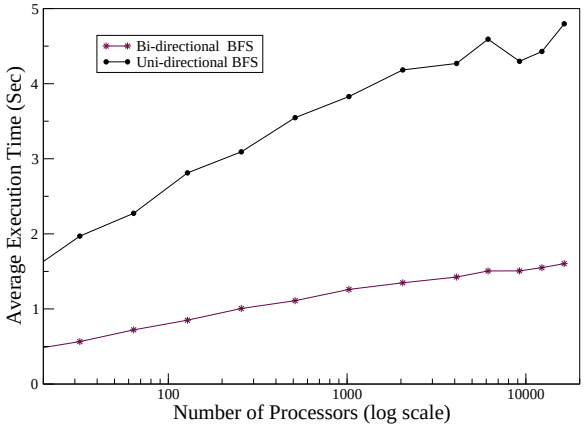
It is shown in Figure 4.a that the execution time curves increase in proportion to $\log P$, where P is the number of processors, and it is confirmed by a regression analysis. Part of the reason for the logarithmic scaling factor is that the search time for a graph is dependent on the length of path between the source and destination vertices, and the path length is bounded by the diameter of the graph, which is $O(\log n)$ for a random graph with n vertices [2]. That is, n increases proportionally as P increases in weak scaling study, and therefore the diameter of the graph (and the search time) increases in proportion to $\log P$. The performance of the BFS algorithm improves as the average degree, k, increases. This is obvious, because



(a) Mean search time



(b) Message volume per level



(c) Bi-directional search

Figure 4: Weak scaling results of the distributed BFS on 32768-node BlueGene/L system. $|V|$ and k denote the number of vertices assigned to each processor and the average degree, respectively.

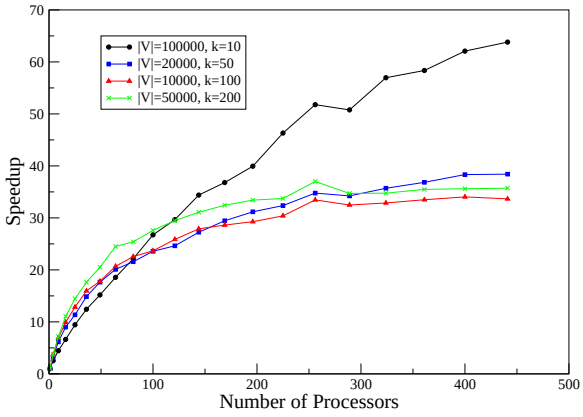


Figure 5: Strong scaling results of the distributed BFS on BlueGene/L system. $|V|$ denotes the number of vertices per processor and k denotes the average vertex degree.

as the degree of vertices increases the length of a path being searched decreases, and hence the search time decreases. Note, however, that for larger average degree, the execution time increases faster than $\log(n)$.

The Figure 4.b shows the total volume of messages received by our BFS algorithm as a function of the number of levels used in the search. These results are for a small graph with 12 million vertices and 120 million edges. It can be clearly seen in the figure that the message volume increases quickly as the path length increases until the path length reaches the diameter of the graph.

The scalability of the bi-directional BFS algorithm is compared with that of the uni-directional BFS for the case with the average degree of 10 as shown in Figure 4.c. Similar to the uni-directional search, the scaling factor is $\log P$. As expected, the bi-directional search outperforms the uni-directional search. The search time of the bi-directional BFS in the worst case is only 33% of that of the uni-directional BFS. This is mainly because the bi-directional search walks shorter distance than the uni-directional search and significantly reduces the volume of overall messages to be processed. We have verified that the total volume of messages received by each processor in a bi-directional search is orders of magnitude smaller than that in a uni-directional search.

We have conducted strong-scaling experiments and present the results in Figure 5. In contrast to weak scaling, we fix the size of a graph while increasing the number of processors in our strong scaling experiments. In Figure 5, the speedup curves grows in proportion to $\sqrt{P}$ for small P, where P is the number of processors. For larger P, the speedup tapers off as the local problem size becomes very small and the communication overhead becomes dominant.

The performance of the 2D and 1D partitioning are compared in Table 1 for different processor topologies. We have used two graphs, which have 3.2 billion and 0.32 billion edges respectively, in the experiments. It can be clearly seen in the table that the communication time of 1D partitioning is much higher than that of 2D partitioning. The average length of messages received by each processor per level is measured for the expand and fold operations in addition to the total execution and communication time. The higher communication time of the 1D partitioning is due to the larger number of processors involved in collective communications. In the worst case, the communication takes about 40% of the total execution time. These results show that 2D partitioning can reduce communication time.

It is interesting to note that in some cases with lower degree, where row-wise partition is used, the 1D

($\|V\|$, k)	$R \times C$	Execution Time	Comm. Time	Avg. Message Length per Level	
				Expand	Fold
$\|V\|$=100000 k=10	128×256	4.800	0.318	64016.70	65371.19
	256×128	4.843	0.324	65315.12	64124.96
	32768×1	5.649	2.147	66640.10	9032.11
	1×32768	4.180	2.246	6379.10	66640.50
$\|V\|$=10000 k=100	128×256	2.283	0.157	95573.54	115960.29
	256×128	2.385	0.164	114285.92	98418.21
	32768×1	3.172	1.391	138265.36	1760.00
	1×32768	2.681	1.363	1361.99	138280.39

Table 1: Performance results for various processor topologies. $|V|$ denotes the number of vertices per processor and k denotes the average vertex degree. The larger communication timings for 1D partitioning is due to more processors involved in the collective communications.

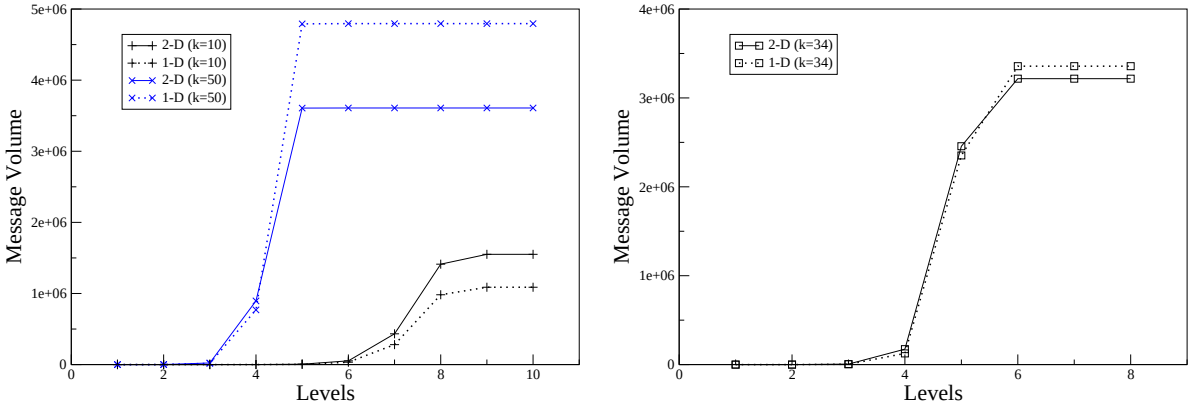
partitioning outperforms the 2D partitioning with the same problem size, despite the increased communication cost. The average length of the fold messages in a 1D partitioning is comparable to that of 2D partitioning. On the other hand, much shorter messages are exchanged during an expand operation. Not only those expand messages are transmitted locally, the shorter messages result in reduction in memory accesses and performance improvement. In other words, with the 1D partitioning there is a trade-off between higher communication cost and lower memory accessing time. The 2D partitioning should outperform 1D partitioning for the graphs with higher degree, and this was verified for a graph with fewer vertices (0.32 billion) but higher degree (100) in the table.

The effect of the average degree of a graph on the performance of partitioning schemes is analyzed further and shown in Figure 6, which plots the volume of messages received by a processor at each level-expansion of a search as a function of level in the search. Graphs with 40 million vertices with varying average degrees, partitioned over 20×20 processor mesh, are analyzed in this study. We have used an unreachable target vertex in the search to capture the worst-case behavior of the partitioning schemes. Figure 6.a, where graphs with the average degrees of 10 and 50 are analyzed, shows that the message volume increases more slowly with 1D partitioning than 2D partitioning for the low-degree graph as the search progresses. For the high-degree graph, 2D partitioning generates less messages than 1D partitioning. Further, we can determine the average degree of a Poisson random graph with which 1D and 2D partitionings exhibit identical performance. That is, assuming $R = C = \sqrt{P}$, we can calculate the value of the k by solving an equation

$$n \cdot \gamma(\frac{n}{P}) \cdot \frac{P-1}{P} = 2 \cdot \frac{n}{P} \cdot \gamma(\frac{n}{\sqrt{P}}) \cdot (\sqrt{P} - 1)$$

for given n and P. The left and right hand sides of the equation represent the message lengths per level-expansion for 1D and 2D partitionings, respectively. We have computed the value of such k (34) for P=400 and n=40000000 and compared the performance of 1D and 2D partitionings with the graph in Figure 6.b. As expected, both 1D and 2D partitionings show nearly identical performance.

We demonstrate the effectiveness of our union-fold operation for the BlueGene/L in the Figure 7. We have used the redundancy ratio as performance metric in this experiment. The redundancy ratio is defined as the ratio of duplicate vertices eliminated by the union-fold operation to the total number of vertices received a processor. Obviously, more redundant vertices can be eliminated by the union-fold operation for the graph with the higher degree (100). It is shown that the union-fold operation can save as much as 80% of vertices received by each processor. Although the proposed union operation requires copying of received messages incurring additional overhead, it reduces the total number of vertices to be processed by each processor and ultimately improves overall performance by reducing memory accessing time of the processor. The redundancy ratio declines for both graphs, however, as the number of processors increases. It has been shown in Figure 4.b that the message length increases exponentially as search expands its frontiers until



(a) $k = 10$ and $k = 50$ (b) $k = 34$

Figure 6: Message volume as a function of level in a search. Graphs with 40 million vertices are used. In (b), the value of k is derived from an equation, $n \cdot \gamma(\frac{n}{P}) \cdot \frac{P-1}{P} = 2 \cdot \frac{n}{P} \cdot \gamma(\frac{n}{\sqrt{P}}) \cdot (\sqrt{P} - 1)$, where $P = 400$ and $n = 40000000$.

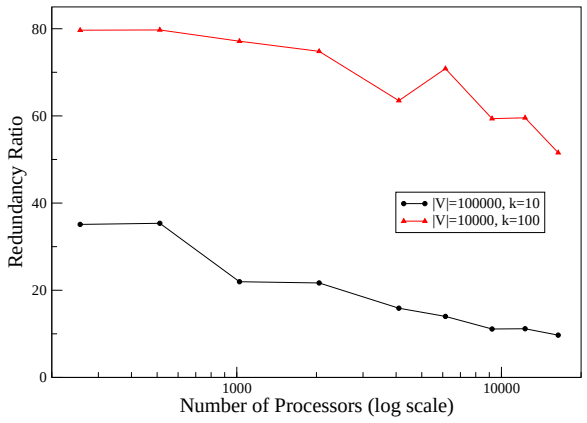


Figure 7: Performance of the proposed union-fold operation for BlueGene/L. $|V|$ denotes the number of vertices per processor and k denotes the average vertex degree.

the path length approaches the diameter of the graph, after which the message length remains constant. This means that the total number of vertices (or total message length) received by each processor should be almost constant independent of the number of processors in a weak scaling run, since the diameter of the graph increases very slowly especially for large graphs. What this implies is that the number of duplicate vertices in received messages should be constant as well. However, in our union-fold operation each processor receives more messages as the number of processors increases, because it passes the messages using ring communications. This is why the redundancy ratio declines as more processors are used.

5 Conclusions

We have developed a scalable parallel distributed BFS algorithm and demonstrated its scalability on BlueGene/L with 32,768 processors in this paper. The proposed algorithm uses 2D edge partitioning. We have shown that for Poisson random graphs the length of messages from a single processor is proportional to the number of vertices assigned to the processor. We use this information to confine the length of message buffers for better scalability. We also have developed two efficient collective communication operations based on point-to-point communication designed for BlueGene/L, which utilizes the high-bandwidth torus network of the machine. Using this algorithm, we have searched very large graphs with more than 3 billion vertices and 30 billion edges. To the best of our knowledge, this is the largest graph searched by any graph search methods. Furthermore, this work provides insight on how to design scalable algorithms for data- and communication-intensive applications for very large parallel computers like BlueGene/L.

Acknowledgements

This work was performed under the auspices of the U.S. Department of Energy by University of California Lawrence Livermore National Laboratory under contract No. W-7405-Eng-48.

References

[1] Blue Gene/L. http://cmg-rr.llnl.gov/asci/platforms/bluegenel.

[2] B. Bollobás. The diameter of random graphs. *Trans. American Mathematical Society*, 267:41–52, 1981.

[3] U. V. Çatalyürek and C. Aykanat. A hypergraph-partitioning approach for coarse-grain decomposition. In *ACM/IEEE SC2001*, Denver, CO, November 2001.

[4] M. Chein and M.-L. Mugnier. Conceptual graphs: Fundamental notions. *Revue d'intelligence artificielle*, 6(4):365–406, 1992.

[5] A. Crauser, K. Mehlhorn, U. Meyer, and P. Sanders. A parallelization of Dijkstra's shortest path algorithm. *Lecture Notes in Computer Science*, 1450:722–731, 1998.

[6] G. Fox, M. Johnson, S. Otto, J. Salmon, and D. Walker. *Solving Problems on Concurrent Processors.* Prentice-Hall, Inc., 1988.

[7] A. Y. Grama and V. Kumar. A survey of parallel search algorithms for discrete optimization problems, 1993.

[8] Y. Han, V. Y. Pan, and J. H. Reif. Efficient parallel algorithms for computing all pair shortest paths in directed graphs. In *ACM Symposium on Parallel Algorithms and Architectures*, pages 353–362, 1992.

[9] B. Hendrickson, R. Leland, and S. Plimpton. An efficient parallel algorithm for partitioning irregular graphs. *Int. Journal of High Speed Computing*, 7(1):73–88, 1995.

[10] P. N. Klein and S. Subramanian. A randomized parallel algorithm for single-source shortest paths. *J. Algorithms*, 25(2):205–220, 1997.

[11] R. Levinson. Towards domain-independent machine intelligence. In G. Mineau, B. Moulin, and J. Sowa, editors, *Proc. 1st Int. Conf. on Conceptual Structures*, volume 699, pages 254–273, Quebec City, Canada, 1993. Springer-Verlag, Berlin.

[12] J. G. Lewis, D. G. Payne, and R. A. van de Geijn. Matrix-vector multiplication and conjugate gradient algorithms on distributed memory computers. In *Proceedings of the Scalable High Performance Computing Conference*, pages 542–550, 1994.

[13] J. G. Lewis and R. A. van de Geijn. Distributed memory matrix-vector multiplication and conjugate gradient algorithms. In *Proceedings of Supercomputing'93*, pages 484–492, Portland, OR, November 1993.

[14] K. Macherey, F. Och, and H. Ney. Natural language understanding using statistical machine translation, 2001.

[15] Multiprogrammatic Capability Cluster (MCR). http://www.llnl.gov/linux/mcr.

[16] I. Pohl. Bi-directional search. *Machine Intelligence*, 6:127–140, 1971. eds. Meltzer and Michie, Edinburgh University Press.

[17] Y.-J. Suh and K. G. Shin. All-to-all personalized communication in multidimensional torus and mesh networks. *IEEE Trans. on Parallel and Distributed Systems*, 12:38–59, 2001.