Distributed Breadth-First Search with 2-D Partitioning[*]

Edmond Chow, Keith Henderson, Andy Yoo
Lawrence Livermore National Laboratory

Abstract

Many emerging large-scale data science applications require searching large graphs distributed across multiple memories and processors. This paper presents a scalable implementation of distributed breadth-first search (BFS) which has been applied to graphs with over one billion vertices. The main contribution of this paper is to compare a 2-D (edge) partitioning of the graph to the more common 1-D (vertex) partitioning. For Poisson random graphs which have low diameter like many realistic information network data, we determine when one type of partitioning is advantageous over the other. Also for Poisson random graphs, we show that memory use is scalable. The experimental tests use a level-synchronized BFS algorithm running on a large Linux cluster and BlueGene/L. On the latter machine, the timing is related to the number of synchronization steps in the algorithm.

1 Introduction

Many data science applications are beginning to use sparse relational data in the form of graphs. For example, to determine the nature of the relationship between two vertices in a relational data graph, the shortest graph distance paths between the two vertices can be used [1, 5]. Breadth-first search (BFS) may be applied for this task. More efficient algorithms based on bi-directional or heuristic search are also possible but fundamentally have the same structure as BFS. Indeed, BFS is used in a myriad of applications and is related to sparse matrices and sparse matrix-sparse vector multiplication. Distributed BFS (and shortest path) algorithms are not new. There is in particular much literature on the case where each processing element stores a single vertex in the graph. See, for example, [15] for a review of results.

In this paper, we study the behavior of BFS on very large-scale graphs partitioned across a large distributed memory supercomputer. A critical question is how to partition the graph, which will affect the communication and overall time for BFS. This paper is novel in that it studies distributed BFS with a 2-D partitioning of the graph, corresponding to a partitioning of the edges. The more common 1-D partitioning corresponds to a partitioning of the vertices. 1-D partitioning is simpler and leads to low communication volume and number of messages for graphs arising in many scientific applications, such as the numerical solution of partial differential equations (PDEs). Many graphs arising in data science applications, however, have properties very different from PDE meshes. In particular, these graphs have low average path length, that is, the typical length of the shortest distance path between

[*]LLNL Technical Report UCRL-CONF-210829.

any two vertices is small. For example, the average path length may vary as $O(\log n)$ where there are n vertices in the graph. 1-D partitionings for graphs with this property leads to high communication volume and total number of messages.

We thus consider 2-D partitioning. Besides potentially lower communication volume, the advantage of 2-D partitioning over 1-D partitioning is that collective communications involve only $O(\sqrt{P})$ processors rather than P processors in the 1-D case. 2-D partitioning is most well-known for dense matrix computations [6] and has been proposed for sparse matrix computations where the structure of the matrix is difficult to exploit [9]. However, 2-D partitioning is rarely used, perhaps because most problems to date have sufficient structure so that 1-D partitioning is adequate. Further, whereas there are many algorithms and codes for generating 1-D partitionings by optimizing communication volume [8, 10, 3], algorithms and codes for 2-D partitioning have only recently become available [2, 4].

There are several variants of 2-D partitioning. The most useful variant has been called *transpose free doubling/halving* [12], *redistribution-free method* [11], and *2-D checkerboard* [4]. In this paper, we will simply call this 2-D partitioning, and will describe it in the next section. For other variants, see [2, 6, 9, 11, 12, 13].

In this paper, we study Poisson random graphs, where the probability of an edge connecting any two vertices is constant. We use arbitrary partitionings with the constraint that the partitions are balanced in terms of number of vertices and edges. For P processors and n vertices, we define a scalable implementation to be one that uses $O(n/P)$ storage per processor. For Poisson random graphs, we will show in particular that the sum of the sizes of the messages sent by one processor to other processors is $O(n/P)$ for fixed average degree k. This result applies to both 1-D and 2-D partitionings.

In other work we are also studying more structured graphs. In particular, a family of graphs called *spatial* graphs has as a parameter the probability of connectedness between neighbors of a vertex. Many realistic networks have a high value for this probability, called *clustering coefficient* [16], and these graphs are more amenable to partitioning. For these graphs, partitioning codes can be used to generate partitionings and the quality of 1-D and 2-D partitionings can be analyzed with respect to the clustering coefficient. Poisson random graphs are the worst case limit of this family of graphs.

This paper is organized as follows. In Section 2, we discuss implementations of distributed BFS for 1-D and 2-D partitioning. In particular, there are several choices for the communication operations. In Section 3, for a Poisson random graph, we discuss the size of the messages in these communication operations in terms of parameters of the graph and the partitioning. In Section 4, we show the performance results of parallel BFS in the 1-D and 2-D cases. Section 5 concludes the paper with several directions for future research.

2 Implementation of Distributed BFS

In this section, we present the implementation of distributed BFS with 1-D and 2-D partitioning. The 1-D algorithm is a special case of the 2-D algorithm. The algorithms are described in the next two subsections. Then we discuss two important issues: options for the communication steps, and scalable data structures.

2.1 Terms and Definitions

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. We will consider undirected graphs only in this paper.

We define the *level* of a vertex as its graph distance from a start vertex. Level-synchronized BFS algorithms proceed level-by-level starting with the start vertex (or a set of start vertices). We define the *frontier* as the set of vertices in the current level of the BFS algorithm. A *neighbor* of a vertex is a vertex that shares an edge with that vertex.

2.2 Distributed BFS with 1-D Partitioning

A 1-D 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. The set of vertices owned by a processor is also called its *local vertices.* The following illustrates a 1-D 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 1-D 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 potentially add these vertices to their frontier set for the next level. Each processor receives these sets of neighbor vertices and merges them to form a set $\bar{N}$ of vertices which the 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 distributed breadth-first expansion using a 1-D partitioning, starting with a vertex v_s. In the algorithm, every vertex v becomes labeled with its level, $L_{v_s}(v)$, i.e., its graph distance from v_s. (BFS is a simple modification of breadth-first expansion.) The data structure $L_{v_s}(v)$ is also distributed so that a processor only stores L for its local vertices.

2.3 Distributed BFS with 2-D partitioning

A 2-D (checkerboard) 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, like in 1-D partitioning. 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.

$$\left[\begin{array}{c|c|c|c}
A_{1,1}^{(1)} & A_{1,2}^{(1)} & \cdots & A_{1,C}^{(1)} \\ \hline
A_{2,1}^{(1)} & A_{2,2}^{(1)} & \cdots & A_{2,C}^{(1)} \\ \hline
\vdots & \vdots & \ddots & \vdots \\ \hline
A_{R,1}^{(1)} & A_{R,2}^{(1)} & \cdots & A_{R,C}^{(1)} \\ \hline
\multicolumn{4}{c}{\vdots} \\
\multicolumn{4}{c}{\vdots} \\
\multicolumn{4}{c}{\vdots} \\ \hline
A_{1,1}^{(C)} & A_{1,2}^{(C)} & \cdots & A_{1,C}^{(C)} \\ \hline
A_{2,1}^{(C)} & A_{2,2}^{(C)} & \cdots & A_{2,C}^{(C)} \\ \hline
\vdots & \vdots & \ddots & \vdots \\ \hline
A_{R,1}^{(C)} & A_{R,2}^{(C)} & \cdots & A_{R,C}^{(C)}
\end{array}\right]$$

Here, the partitioning is for $P = RC$ 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 2-D partitioning above, the adjacency matrix is divided into RC 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 $(j-1)R+i$. For the partitioning to be balanced, each processor should be assigned approximately the same number of vertices and edges. The 1-D case is equivalent to the 2-D case with $R = 1$ or $C = 1$.

For 2-D partitioning, we assume that the edge list for a given vertex is a *column* of the adjacency matrix, which will slightly simplify the description of the algorithm. Thus each block 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. 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 2-D partitioning, these owner processors are in the same processor row. The advantage of 2-D partitioning over 1-D partitioning is that the processor-column and processor-row communications involve R and C processors, respectively; for 1-D partitioning, all P processors are involved in the communication operation.

Algorithm 2 describes distributed breadth-first expansion using a 2-D partitioning, starting with a vertex v_s. In the algorithm, every vertex v becomes labeled with its level $L_{v_s}(v)$, i.e., its graph distance from v_s.

Algorithm 1 Distributed Breadth-First Expansion with 1-D Partitioning

```
1: Initialize $L_{v_s}(v) = \begin{cases} 0, & v = v_s \\ \infty, & \text{otherwise} \end{cases}$
2: for $l = 0$ to $\infty$ 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 2-D Partitioning

```
1: Initialize $L_{v_s}(v) = \begin{cases} 0, & v = v_s \\ \infty, & \text{otherwise} \end{cases}$
2: for $l = 0$ to $\infty$ 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:  $N \leftarrow \{$neighbors of vertices in $\bar{F}$ using edge lists on this processor$\}$
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
```

2.4 Communication Options

In this subsection, we describe the communication operations of the above algorithms in detail. Algorithm 1 for 1-D partitioning has a single communication step, in lines 8–13. Algorithm 2 for 2-D partitioning, in contrast, has two communication steps: lines 7–11, called *expand* communication, and lines 13–18, called *fold* communication. The communication step in the 1-D algorithm is the same as the fold communication in the 2-D algorithm.

2.4.1 Expand communication

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, is it 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 would now be implemented by an all-to-all collective communication. In the 2-D 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, i.e., 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.

We thus have all-gather and all-to-all options for the expand communication. On hypercubes, the all-gather operation is often implemented with a recursive-doubling algorithm [6], which is also better than other options for shorter messages. On mesh and torus architectures all-gather may be implemented with a ring algorithm; see, e.g., [14].

2.4.2 Fold communication

The fold communication is traditionally implemented for dense matrix computations as an all-to-all operation. We will show in Section 3 that the message lengths are scalable in this operation for Poisson random graphs.

An alternative (which we have not used) is to implement the fold communication as a reduce-scatter operation. In this case, in Algorithm 2, each processor receives its $\bar{N}$ directly and line 18 of the algorithm is not required. The reduction operation occurs within the reduction stage of the operation and eliminates duplicate vertex indices being sent to a processor. This reduction operator is not provided by MPI, and thus this reduce-scatter operation must be coded using sends and receives.

On hypercubes, a recursive-halving algorithm may be used for the reduce-scatter operation [6]. On mesh and torus architectures, a ring algorithm may be used.

2.5 Scalable data structures and optimizations

Storage of edge lists

In the 2-D case, 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. nk is an upper bound on the number of edge lists in the graph, 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 nonempty row edge lists would be $O(n/P)$ as well. Each nonempty row edge list on a processor corresponds to a unique entry in the column edge lists on that processor.

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 (e.g., 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 with a hash 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 $O(n/P)$.

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 Message Lengths for Random Graphs

For Poisson random graphs, we can derive bounds on the length of the communication messages in distributed BFS. 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 2-D 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 1-D 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 2-D 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 2-D 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 1-D and 2-D partitioning, the length of the communication from a single processor is $O(n/P)$, proportional to the number of vertices owned by a processor.

For illustration purposes, Figure 1 shows the message lengths for a single processor for 1-D and 2-D partitioning (sum of expand and fold communication in the 2-D case) as a function of the average degree, k. The results show that 2-D partitioning is advantageous when average degree is large. When more processors are used, the cross-over point is at a higher k.

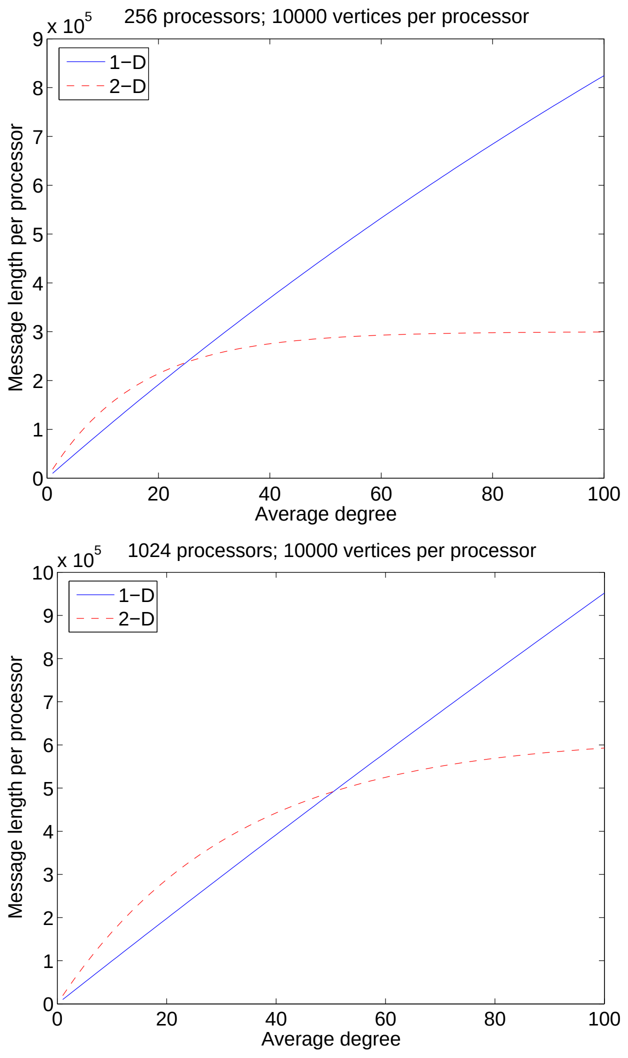

Figure 1: Message lengths for 256 and 1024 processors.

4 Parallel Performance Results

This section shows timing results for distributed BFS for Poisson random graphs with arbitrary but load balanced 1-D and 2-D partitionings. For 2-D partitioning, $R = C = \sqrt{P}$ was used in all cases. For 1-D partitioning, $R = 1$ and $C = P$ was used, which performed better than $R = P$ and $C = 1$. One hundred pairs of start and target vertices were generated randomly, and the timings reported are the average of the final 99 trials. Each search terminated when either the target vertex was reached or the size of the fringe on every process was zero. The code was run on MCR (a large Linux Cluster with a Quadrics switch) and IBM BlueGene/L located at Lawrence Livermore National Laboratory.

First, the actual message lengths in distributed BFS were measured and are plotted in Figure 2. The results are qualitatively similar to the bounds derived in Section 3 and plotted in Figure 1.

Figure 3 shows a weak scaling test on MCR using average degree $k = 100$. The global number of vertices increases proportionally with the number of processors in this test. The largest problem size is 102.4 million vertices and 10.24 billion edges. As expected for this value of k, 2-D partitioning shows better timings than 1-D partitioning. Qualitatively similar results were observed on BG/L.

Figure 4 shows a weak scaling test on MCR using average degree $k = 10$. The largest problem size is 1.024 billion vertices, although the number of edges is the same as that for the previous problem. 1-D partitioning is better in this regime of $k = 10$.

Figure 5 shows a strong scaling test, where the global number of vertices is constant while the number of processors is increased. The number of local vertices decreases. The scaling is worse for 1-D partitioning because 1-D partitioning behaves worse for small local problem sizes. The decline in speedup signals the point where increased communication overhead outweighs any performance gain obtained by increased parallelism. 1-D partitioning reaches this point for smaller P than 2-D partitioning, since the number of messages is asymptotically larger in the 1-D case.

Figure 6 shows a weak scaling test using 2-D partitioning with BG/L. The timing appears to increase logarithmically with the global number of vertices, suggesting that the time dependency is correlated to the number of synchronization steps, or levels. We note that the average path length between two arbitrary vertices in a random graph grows as a logarithm of the number of vertices [7]. However, this growth, from length 5.6 for the smallest problem to length 7.1 for the largest problem is slower than the growth in the timings. Thus the communication time is also increasing as the number of processors increases. Unfortunately, the current data are not precise enough to determine the asymptotic rate of growth.

5 Conclusions

This paper has demonstrated large-scale distributed BFS with Poisson random graphs with as many as 1 billion vertices and 10 billion edges. Although the run time is almost entirely due to communication, good speedups are possible. We considered 1-D and 2-D partitionings theoretically and experimentally. 2-D partitionings are to be preferred when the average degree of the graph is large. Further, for both 1-D and 2-D partitionings of Poisson random graphs, the memory usage per processor is scalable.

Future work should address graphs besides Poisson random graphs, e.g., graphs with

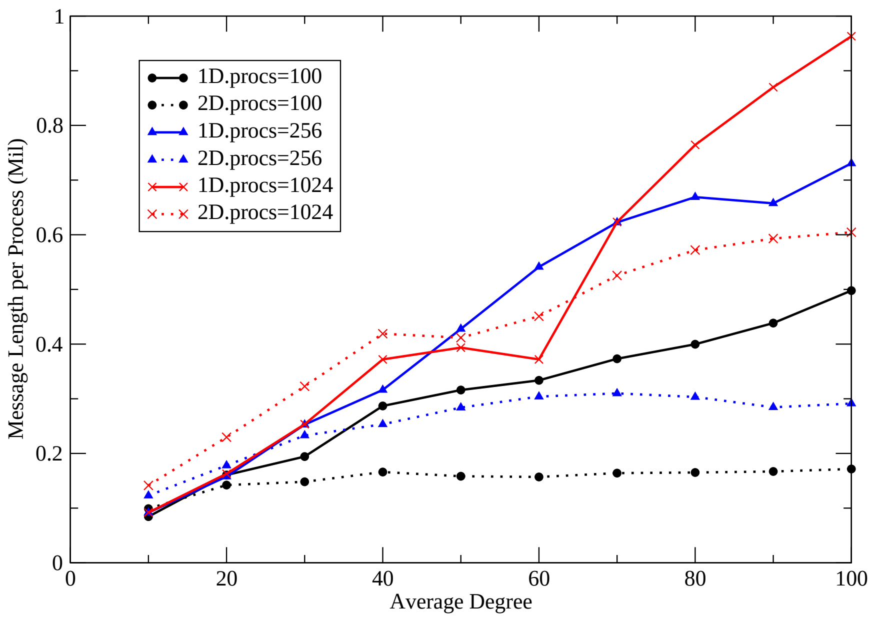

Figure 2: Effect of average degree on message lengths, 10000 vertices per processor.

large clustering coefficient mentioned in the Introduction, and scale-free graphs, which are graphs with a few vertices of very large degree. Graphs with large diameter should also be investigated for completeness, although a level-synchronized algorithm may perform poorly for these cases.

Acknowledgements

The authors are pleased to thank Doruk Bozdağ, Ümit Çatalyürek, and Tina Eliassi-Rad for helpful discussions on this work. 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] M. Barthélemy, E. Chow, and T. Eliassi-Rad. Knowledge representation issues in semantic graphs for relationship detection. In *2005 AAAI Spring Symposium on AI Technologies for Homeland Security*, 2005.

[2] Ü. V. Çatalyürek. *Hypergraph Models for Sparse Matrix Partitioning and Reordering.* PhD thesis, Bilkent University, Computer Engineering and Information Science, Nov 1999.

[3] Ü. V. Çatalyürek and C. Aykanat. Hypergraph-partitioning based decomposition for parallel sparse-matrix vector multiplication. *IEEE Transactions on Parallel and Distributed Systems*, 10(7):673–693, 1999.

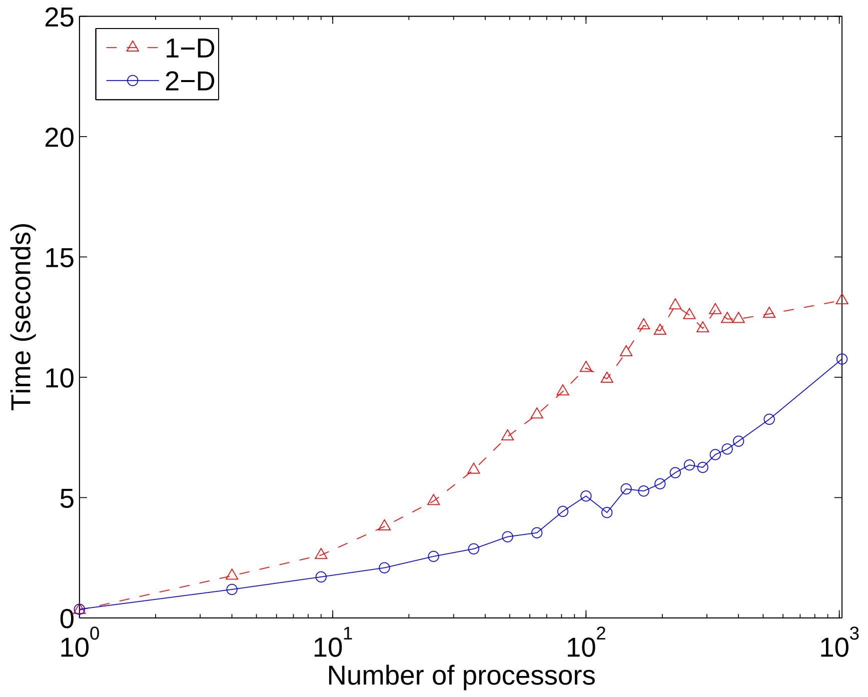

Figure 3: Weak scaling test on MCR, $k = 100$, 100k vertices per processor. The largest problem size is 102.4 million vertices and 10.24 billion edges. 2-D partitioning is better in this regime of $k = 100$.

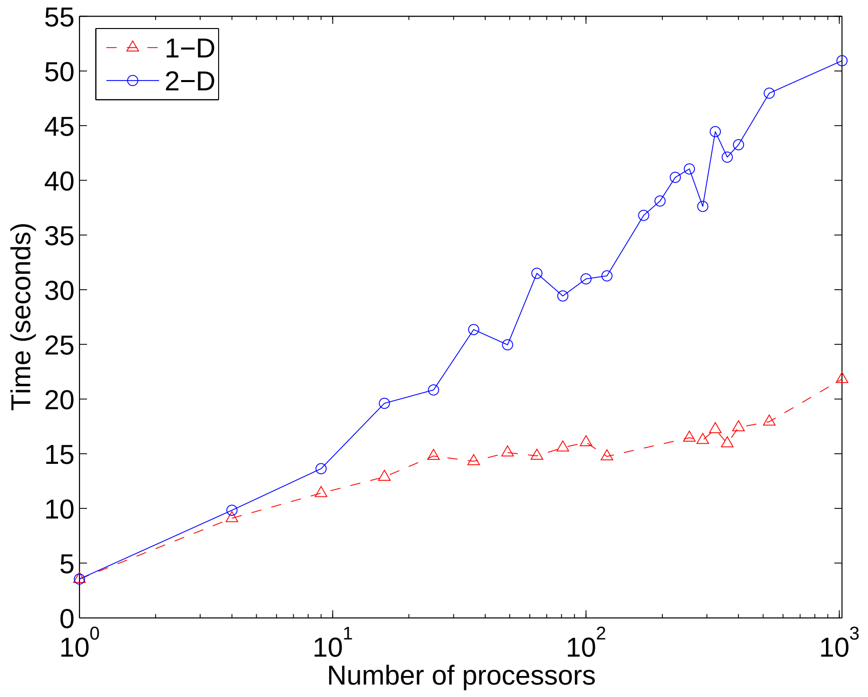

Figure 4: Weak scaling test on MCR, $k = 10$, 1M vertices per processor. The largest problem size is 1.024 billion vertices and 10.24 billion edges. 1-D partitioning is better in this regime of $k = 10$.

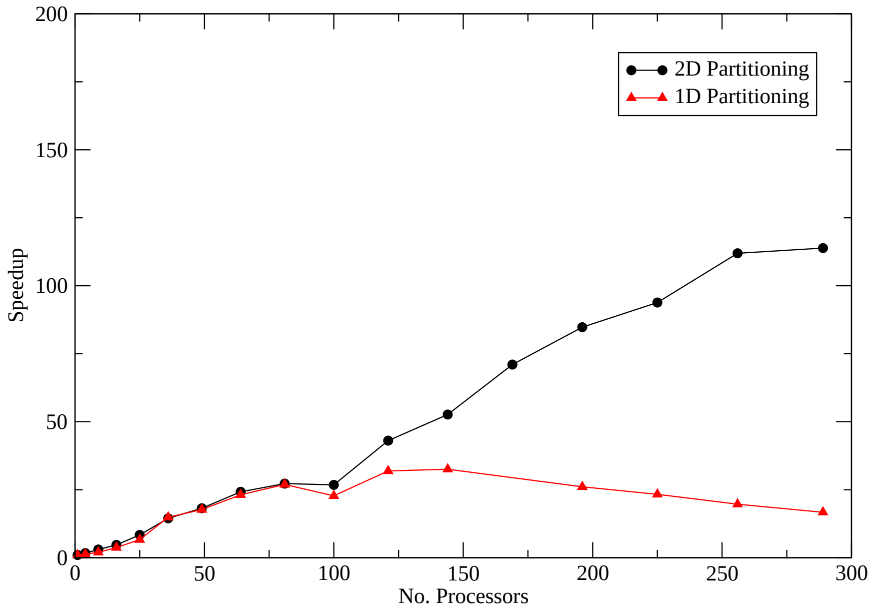

Figure 5: Strong scaling test on MCR, $k = 10$, 2 million vertices total.

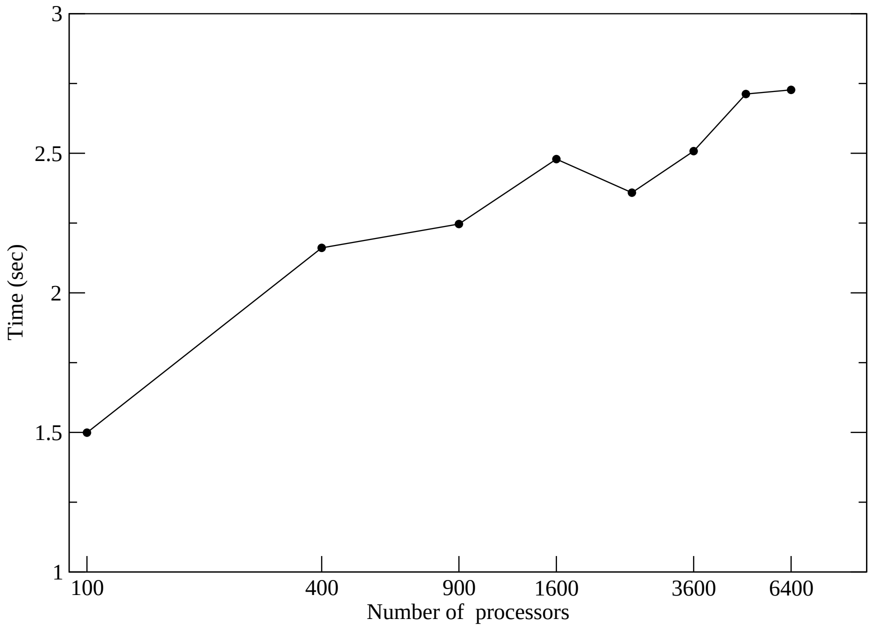

Figure 6: Weak scaling test on BG/L with 2-D partitioning, $k = 10$, 100k vertices per processor.

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

[5] C. Faloutsos, K. McCurley, and A. Tomkins. Fast discovery of connection subgraphs. In *Proceedings of the 10th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, pages 118–127, Seattle, WA, USA, 2004. ACM Press.

[6] G. Fox, M. Johnson, G. Lyzenga, S. Otto, J. Salmon, and D. Walker. *Solving Problems on Concurrent Processors*, volume 1. Prentice Hall, 1988.

[7] A. Fronczak, P. Fronczak, and J. A. Holyst. Average path length in random networks. *Physical Review E*, page 056109, 2004.

[8] B. Hendrickson and R. Leland. A multilevel algorithm for partitioning graphs. Technical report, Sandia National Laboratories, 1993.

[9] B. Hendrickson, R. Leland, and S. Plimpton. An efficient parallel algorithm for matrix-vector multiplication. *Int. J. High Speed Computing*, 7(1):73–88, 1995.

[10] G. Karypis and V. Kumar. A fast and high quality multilevel scheme for partitioning irregular graphs. *SIAM Journal on Scientific Computing*, 20(1), 1999.

[11] 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.

[12] 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.

[13] Andrew T. Ogielski and William Aiello. Sparse matrix computations on parallel processor arrays. *SIAM Journal on Scientific Computing*, 14(3):519–530, 1993.

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

[15] J. L. Träff. An experimental comparison of two distributed single-source shortest path algorithms. *Parallel Computing*, 21:1505–1532, 1995.

[16] D. J. Watts and S. H. Strogatz. Collective dynamics of small-world networks. *Nature*, 393:440–442, 1998.