Graph Partitioning for Scalable Distributed Graph Computations

Aydın Buluç[1] and Kamesh Madduri[2]

[1] Lawrence Berkeley National Laboratory, USA,
abuluc@lbl.gov
[2] The Pennsylvania State University, USA,
madduri@cse.psu.edu

Abstract. Inter-node communication time constitutes a significant fraction of the execution time of graph algorithms on distributed-memory systems. Global computations on large-scale sparse graphs with skewed degree distributions are particularly challenging to optimize for, as prior work shows that it is difficult to obtain balanced partitions with low edge cuts for these graphs. In this work, we attempt to determine the optimal partitioning and distribution of such graphs, for load-balanced parallel execution of communication-intensive graph algorithms. We use Breadth-First Search (BFS) as a representative example, and derive upper bounds on the communication costs incurred with a two-dimensional partitioning of the graph. We present empirical results for communication costs with various graph partitioning strategies, and also obtain parallel BFS execution times for several large-scale DIMACS Challenge graph instances on a large supercomputing platform. Our performance results indicate that for several graph instances, reducing work and communication imbalance among partitions is more important than minimizing the total edge cut.

Keywords: graph partitioning, hypergraph partitioning, inter-node communication modeling, breadth-first search, 2D decomposition

1 Introduction

Graph partitioning is an essential preprocessing step for distributed graph computations. The cost of fine-grained remote memory references is extremely high in case of distributed memory systems, and so one usually restructures both the graph layout and the algorithm in order to mitigate or avoid inter-node communication. The goal of this work is to characterize the impact of common graph partitioning strategies that minimize edge cut, on the parallel performance of graph algorithms on current supercomputers. We use Breadth-First Search (BFS) as our driving example, as it is representative of communication-intensive graph computations. It is also frequently used as a subroutine for more sophisticated algorithms such as finding connected components, spanning forests, testing for bipartiteness, maximum flows, and computing betweenness centrality

on unweighted graphs. BFS has recently been chosen as the first representative benchmark for ranking supercomputers based on their performance on data intensive applications [4].

Given a distinguished "source vertex" s, Breadth-First Search (BFS) systematically explores the graph G to discover every vertex that is reachable from s. Let V and E refer to the vertex and edge sets of G, whose cardinalities are $n = |V|$ and $m = |E|$. We assume that the graph is unweighted; equivalently, each edge $e \in E$ is assigned a weight of unity. A *path* from vertex s to t is defined as a sequence of edges $\langle u_i, u_{i+1} \rangle$ (edge directivity assumed to be $u_i \rightarrow u_{i+1}$ in case of directed graphs), $0 \leq i < l$, where $u_0 = s$ and $u_l = t$. The *length* of a path is the sum of the weights of edges. We use $d(s, t)$ to denote the *distance* between vertices s and t, or the length of the shortest path connecting s and t. BFS implies that all vertices at a distance k (or "level" k) from vertex s should be first "visited" before vertices at distance $k + 1$. The distance from s to each reachable vertex is typically the final output. In applications based on a breadth-first graph traversal, one might optionally perform auxiliary computations when visiting a vertex for the first time. Additionally, a "breadth-first spanning tree" rooted at s containing all the reachable vertices can also be maintained.

2 Parallel BFS

Data distribution plays a critical role in parallelizing BFS on distributed-memory machines. The approach of partitioning vertices to processors (along with their outgoing edges) is the so-called 1D partitioning, and is the method employed by Parallel Boost Graph Library [5]. A two-dimensional edge partitioning is implemented by Yoo et al. [9] for the IBM BlueGene/L, and by us [1] for different generations of Cray machines. Our 2D approach is different in the sense that it does a 'checkerboard' partitioning (see below) of the sparse adjacency matrix of the underlying graph, hence assigning contiguous submatrices to processors. Both 2D approaches achieved higher scalability than their 1D counterparts. One reason is that key collective communication phases of the algorithm are limited to at most $\sqrt{p}$ processors, thus avoiding the expensive all-to-all communication among p processors. Yoo et al.'s work focused on low-diameter graphs with uniform degree distribution, and ours mainly focused on low-diameter graphs with skewed degree distribution. A thorough study of the communication volume in 1D and 2D partitioning for BFS, which involves decoupling collectives scaling from the number of words moved for a large set of graphs, has not been done. This paper attempts to fill that gap.

We utilize a level-synchronous parallel BFS approach that works with 1D- and 2D-graph partitionings. For BFS with 1D graph partitioning (or row-wise adjacency matrix partitioning), there are three main steps:

- **Local discovery:** Inspect adjacencies of vertices in the current frontier.
- **Fold:** Exchange discovered adjacencies via an all-to-all communication step among processors.
- **Local update:** Update distances/parents locally for unvisited vertices.

2D checkerboard partitioning assumes the sparse adjacency matrix of the graph is partitioned as follows:

$$A = \begin{pmatrix} A_{1,1} & \cdots & A_{1,p_c} \\ \vdots & \ddots & \vdots \\ A_{p_r,1} & \cdots & A_{p_r,p_c} \end{pmatrix} \tag{1}$$

Processors are logically organized on a square $p = p_r \times p_c$ mesh, indexed by their row and column indices. Submatrix A_{ij} is assigned to processor $P(i, j)$.

The parallel BFS algorithm with 2D partitioning has four steps:

- **Expand:** Construct the current frontier of vertices on each processor by a collective gather step.
- **Local discovery:** Inspect adjacencies of vertices in the current frontier.
- **Fold:** Exchange newly-discovered adjacencies.
- **Local update:** Update distances/parents for unvisited vertices.

In contrast to the 1D case, communication in the 2D algorithm happens only along one processor dimension. If *Expand* happens along one processor dimension, then *Fold* happens along the other processor dimension. Detailed pseudo-code for the 1D and 2D algorithms can be found in our earlier paper [1].

The performance of both algorithms heavily depend on the performance and scaling of MPI collective `MPI_Alltoallv`, and the 2D algorithm also depends on the `MPI_AllGatherv` collective.

3 Analysis of Communication Costs

In previous work [1], we studied the performance of parallel BFS on synthetic Kronecker graphs used in the Graph 500 benchmark. We observed that the communication volume was $O(m)$ with a random ordering of vertices, and a random partitioning of the graph (i.e., assigning m/p edges to each processor). The edge cut was also $O(m)$ with random partitioning. While it can be shown that low-diameter real-world graphs do not have sparse separators [7], constants matter in practice, and any decrease in the communication volume, albeit not asymptotically, would translate into lower running times on graph algorithms that are typically communication-bound.

We outline the communication costs incurred in 2D partitioned BFS in this section. 2D-partitioned BFS also captures 1D-partitioned BFS as a degenerate case. For that, we first distinguish different ways of aggregating discovered edges before the fold communication step:

1. No aggregation at all, local duplicates are not pruned before fold.
2. Local aggregation at the current frontier only. Our simulations in Section 6.1 follows this assumption.
3. Local aggregation over all (current and past) locally discovered vertices by keeping a persistent bitmask. We implement this method in Section 6.2.

4. Global aggregation over all previously visited vertices. While this is future work, it has the potential to decrease fold communication to $O(n)$.

2D partitioning is often viewed as edge partitioning. However, a more structured way to think about it is that sets of vertices are collectively owned by all the processors in one dimension. Without loss of generality, we will consider that dimension to be the row dimension, which is of size p_c. These sets of vertices are labeled as $V_1, V_2, ..., V_{p_r}$ and their outgoing edges are labeled as $\text{Adj}^+(V_1), \text{Adj}^+(V_2), ..., \text{Adj}^+(V_{p_r})$. Each of these adjacencies are distributed to members of the row dimension: $\text{Adj}^+(V_1)$ is distributed to $P(1,:)$, $\text{Adj}^+(V_2)$ is distributed to $P(2,:)$, and so on. The colon notation is used to index a slice of processors, e.g. processors in the ith processor row are denoted with $P(i,:)$.

The partial adjacency owned by processor $P(i,j)$ corresponds to submatrix A_{ij} of the adjacency matrix of the graph. The indices of these submatrices need not be contiguous and the submatrices themselves need not be square in general.

For a given processor $P(i,j)$, the set of vertices that needs to be communicated (received) in the kth BFS expand step is $F_k \cap \text{find}(\text{sum}(A_{ij}) > 0)$, where F_k is the frontier at the kth iteration, $\text{sum}(A_{ij})$ computes the column sums of the submatrix A_{ij} and find() returns the set of indices for which the predicate is satisfied. Note that some of these vertices might be owned by the processor itself and need not be communicated.

A more graph-theoretic view is to consider the partitioning of adjacencies. If $\text{Adj}(v)$ of a single vertex v is shared among $\lambda^+ \leq p_c$ processors, then its owner will need to send it to $\lambda^+ - 1$ neighbors. Since each vertex is in the pruned frontier once, the total communication volume for the expand phases over all iterations is equal to the communication volume of the same phase of 2D SpMV [3].

Characterizing communication for the fold phase is harder. Consider a vertex v of incoming degree 9 as shown in Figure 1. In sparse matrix terms, this corresponds to a column with 9 nonzeros. We labeled its incoming adjacency, $\text{Adj}^-(v)$ with a superscript that denotes the earliest iteration they were discovered. In other words, all $v^k \in F_k$. The figure partitions the adjacency to three parts, for which we use different colors.

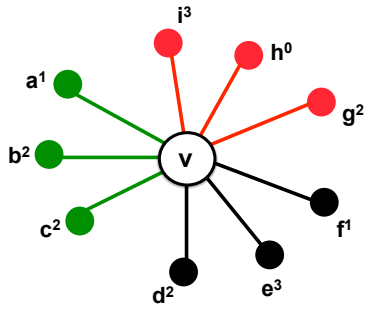
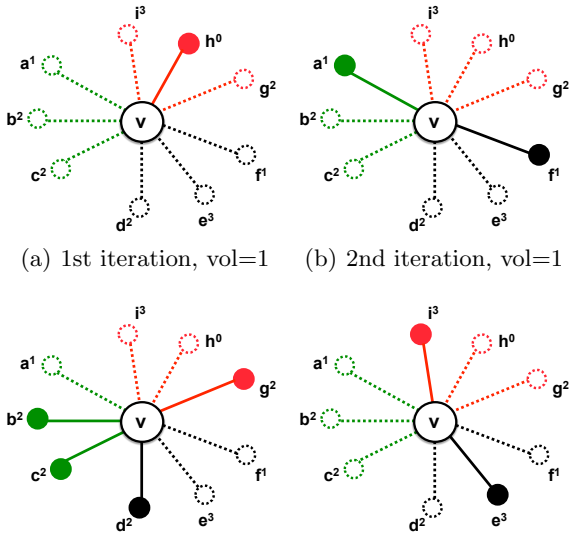


Fig. 1. Partitioning of $\text{Adj}^-(v)$.

Let us consider a *space-time partitioning* of edges. Each edge e is labeled with two integers: (p, k). p denotes the partitioning this edge belongs to, while k is the BFS phase in which e is traversed (remember each edge is traversed exactly once).

The volume of communication due to v in the fold phase is at most degree(v), which is realized when every $e \in \text{Adj}^-(v)$ has a distinct label, i.e. no two edges that are traversed by the same process during the same iteration. Another upper bound is $O(\text{diameter} \cdot (\lambda^- - 1))$, which might be lower than degree. $\lambda^- \leq p_r$ is the number of processors among which $\text{Adj}^-(v)$ is partitioned, and diameter gives the maximum number of BFS iterations. Consequently, for the vertex in our example, $\text{comm}(v) \leq \min(\text{diameter} \cdot (\lambda^- - 1), \text{degree}(v)) = \min(8, 9) = 8$.



(a) 1st iteration, vol=1 (b) 2nd iteration, vol=1

(c) 3rd iteration, vol=2 (d) 4th iteration, vol=1

Fig. 2. Partitioning of $\text{Adj}^-(v)$ per BFS iteration.

Figure 2 shows the partitioning of $\text{Adj}^-(v)$ per BFS iteration. Note that v itself belongs to the "black" partition. At the first iteration, communication volume is 1 due to the "red" processor discovering v through the edge (h, v) and sending the discovered v to the black processor for marking. At the second iteration, both the green and black processors discover v and communication volume is 1 from green to black. Continuing this way, we see that the aggregate communication due to folding of v is 5.

If we implement global aggregation (global replication of discovered vertices), the total communication volume in the fold phase will decrease all the way down to the SpMV case of $(\lambda^- - 1)$. However, this involves an additional communication step similar to the expand phase, in which processors in the column dimension exchange newly visited vertices.

4 Graph/hypergraph partitioning

Our baseline scenario is to take the given ordering of vertices and edges as-is (i.e., the natural ordering), and partitioning the graph into 1D or checkerboard (2D). The second scenario is to randomly permute vertex identifiers, and then partitioning/distributing edges in a 1D or a checkerboard manner. These two scenarios do not explicitly optimize for an objective function.

We can also use a graph partitioner to generate a 1D row-wise partitioning minimizing a specific objective function. We use Metis [6] in this study. Lastly, we experiment with hypergraph partitioning, which exactly captures total communication costs of sparse matrix-dense vector multiplication in its objective function. We used PaToH [2] and report on results from its row-wise and checkerboard partitioning algorithms. Our objective is to study how graph and hypergraph partitioning affect computational load balance and communication costs. In both cases of PaToH, we generate a symmetric permutation as output, since input and output vectors have to be distributed in the same way to avoid data shuffling after each iteration.

We define $V(d, p)$ to be the number of words sent by processor p at the dth BFS phase, on a run with P processors that takes D level-synchronous iterations to finish. We report the following:

1. Total communication volume over the course of BFS iterations:

$$TotalVolume = \sum_{d=1}^{D} \sum_{p=1}^{P} V(d, p).$$

2. Sum of maximum communication volumes for each BFS iteration:

$$MaxVolume = \sum_{d=1}^{D} \max_{p \in \{1 \ldots P\}} V(d, p).$$

Although we report on total communication volume over the course of BFS iterations, we are most concerned with *MaxVolume* metric. It is a better proxy for the time spent on remote communication, since the slowest processor in each phase determines the overall time spent in communication.

5 Experimental Setup

Our parallel BFS implementation is level-synchronous, and so it is primarily meant to be applied to low-diameter graphs. However, to quantify the impact of barrier synchronization and load balance on the overall execution time, we run our implementations on several graphs, both low- and high-diameter.

We categorize the following DIMACS Challenge instances as low diameter: the synthetic Kronecker graphs (`kron_g500-simple-logn` and `kron_g500-logn` families), Erdos-Renyi graphs (`er-fact1.5` family), web crawls (`eu-2005` and others), citation networks (`citationCiteseer` and others), and co-authorship

networks (`coAuthorsDBLP` and others). Some of the high-diameter graphs that we report performance results on include `hugebubbles-00020`, graphs from the `delaunay` family, road networks (`road_central`), and random geometric graphs.

Most of the DIMACS test graphs are small enough to fit in the main memory of a single machine, and so we are able to get baseline serial performance numbers for comparison. We are currently using serial partitioning software to generate vertex partitions and vertex reorderings, and this has been a limitation for scaling to larger graphs. However, the performance trends with DIMACS graphs still provide some interesting insights.

We use the k-way multilevel partitioning scheme in Metis (v5.0.2) with the default command-line parameters to generate balanced vertex partitions minimizing total edge cut. We relabel vertices and distribute edges to multiple processes based on these vertex partitions. Similarly, we use PaToH's column-wise and checkerboard partitioning schemes to partition the sparse adjacency matrix corresponding to the graph. While we report communication volume statistics related to checkerboard partitioning, we are still unable to use these partitions for reordering, since PaToH edge partitions are not necessarily aligned.

We report parallel execution times on 'Hopper', a 6392-node Cray XE6 system located at Lawrence Berkeley National Laboratory. Each node of this system contains two twelve-core 2.1 GHz AMD Opteron 'MagnyCours' processors. There are eight DDR3 1333-MHz memory channels per node, and the observed memory bandwidth with the STREAM [8] benchmark is 49.4 GB/s. The main memory capacity of each node is 32 GB, of which 30 GB is usable by applications. A pair of compute nodes share a 'Gemini' network chip, and these network chips are connected to form a 3D torus (of dimensions $17 \times 8 \times 24$). The observed MPI point-to-point bandwidth for large messages between two nodes that do not share a network chip is 5.9 GB/s. Further, the measured MPI latency for point-to-point communication is 1.4 microseconds, and the cost of a global barrier is about 8 microseconds. The maximum injection bandwidth per node is 20 GB/s.

We use the GNU C compiler (v4.6.1) for compiling our BFS implementation. For inter-node communication, we use Cray's MPI implementation (v5.3.3), which is based on MPICH2. For intra-node threading, we use the GNU C compiler's OpenMP library. We report performance results up to 256-way MPI process concurrency in this study. In all experiments, we use four MPI processes per socket and one OpenMP thread per process. We did not explore multithreading within a node in the current study. This may be another potential source of load imbalance, and we will quantify this in future work. More details on multithreading within a node can be found in our recent parallel BFS paper [1].

To compare performance across multiple systems using a rate analogous to the commonly-used floating point operations/second, we normalize the serial and parallel execution times by the number of edges visited in a BFS traversal and present a 'Traversed Edges Per Second' (TEPS) rate. For an undirected graph with a single connected component, the BFS algorithm would visit every edge in the component twice. We only consider traversal execution times from vertices that appear in the largest connected component in the graph (all the

DIMACS test instances we used have one large component), compute the mean search time (harmonic mean of TEPS) using at least 20 randomly-chosen sources vertices for each benchmark graph, and normalize the time by the cumulative number of edges visited to get the TEPS rate.

6 Performance Analysis and Results

6.1 Empirical modeling of communication

We first report machine-independent measures for communication costs. For this purpose, we simulate parallel BFS using a MATLAB script whose inner kernel, a single BFS step local to a processor, is written in `C++` using `mex` for speed. For each partition, the simulator does multiple BFS runs (in order) starting from different random vertices to report an accurate average, since BFS communication costs, especially the *MaxVolume* metric, depend on on the starting vertex. The imbalance values reported in Tables 5 and 6 are for storage of the graph itself, and exclude the imbalance among the frontier vertices. When reporting the edge cut ratio to the total number of edges in Table 1, the denominator counts each edge twice (since an adjacency is stored twice).

The reported communication volume for the expand phase is exact, in the sense that a processor receives a vertex v only if it owns one of the edges in $\text{Adj}^+(v)$ and it is not the owner of v itself. We this count a vertex as one word of communication. In contrast, in the fold phase, the discovered vertices are sent in $\langle parent, vertex\_id \rangle$ pairs, resulting in two words of communication per discovered edge. This is why values in Table 1 sometimes exceed 100% (i.e. more total communication than the number of edges), but are always less than 200%. For these simulations, we report numbers for both 1D row-wise and 2D checkerboard partitioning when partitioning with the natural ordering, partitioning after random vertex relabeling, and partitioning using PaToH. The 1D partitions obtained with Metis are mostly similar to PaToH 1D row-wise numbers, and we do not report them in current work.

Graph	$p = 4 \times 1$			$p = 16 \times 1$			$p = 64 \times 1$		
	N	R	P	N	R	P	N	R	P
kron-simple-logn18	7.7%	7.6%	6.3%	22.7%	23.1%	19.5%	47.5%	53.4%	45.0%
coAuthorsDBLP	45.2%	81.3%	10.9%	74.9%	148.9%	19.8%	90.1%	182.5%	27.2%
eu-2005	5.3%	23.2%	0.3%	8.7%	63.8%	1.9%	12.3%	107.4%	7.2%
coPapersCiteseer	4.7%	14.7%	1.9%	8.7%	47.9%	3.4%	**10.8%**	**102.5%**	4.8%
delaunay_n20	52.4%	123.7%	0.2%	59.3%	178.0%	0.6%	60.6%	**194.4%**	**1.4%**
rgg_n_2_20_s0	0.2%	85.5%	0.1%	**0.6%**	**160.1%**	0.3%	2.5%	188.9%	0.6%

Table 1. Percentage of *TotalVolume* for 1D row-wise partitioning to the total number of edges (lower is better). *N* denotes the natural ordering, *R* denotes the ordering with randomly-permuted vertex identifiers, and *P* denotes reordering using PaToH.

For the cases of natural and random ordering, the load-balanced '2D vector distribution' [1] is assumed. This vector distribution matches the 2D matrix

distribution. Each processor row (except the last) is responsible for $t = \lfloor n/p_r \rfloor$ elements. The last processor row gets the remaining $n - \lfloor n/p_r \rfloor (p_r - 1)$ elements Within the processor row, each processor (except the last) is responsible for $l = \lfloor t/p_c \rfloor$ elements. PaToH distributes both the matrix and the vectors in order to optimize the communication volume, so PaToH simulations might have unbalanced vector distribution.

Graph	$p = 2 \times 2$			$p = 4 \times 4$			$p = 8 \times 8$		
	N	R	P	N	R	P	N	R	P
kron-simple-logn18	0.71	0.73	0.52	0.51	0.51	0.42	0.43	0.39	0.34
coAuthorsDBLP	1.35	0.92	0.69	1.31	0.91	0.76	1.40	1.00	0.85
eu-2005	1.89	0.73	1.29	1.90	0.56	0.60	1.63	0.57	0.48
coPapersCiteseer	1.32	0.67	0.64	1.25	0.46	0.74	1.35	0.39	0.81
delaunay_n20	1.79	0.95	0.60	2.16	1.09	0.59	2.45	1.24	0.60
rgg_n_2_20_s0	**135.54**	0.75	0.61	**60.23**	0.80	0.64	**18.35**	0.99	0.66

Table 2. Ratio of *TotalVolume* with 2D checkerboard partitioning to the *TotalVolume* with 1D row-wise partitioning (less than 1 means 2D improves over 1D).

For 1D row-wise partitioning, random relabeling increases the total communication volume (i.e., the edge cut), by a factor of up to 10× for low-diameter graphs (realized in `coPaperCiteseer` with 64 processors) and up to 250× for high-diameter graphs (realized in `rgg_n_2_20_s0` with 16 processors), compared to the natural ordering. Random relabeling never decreases the communication volume. PaToH can sometimes drastically reduce the total communication volume, as seen by the case `delaunay_n20` graph (15× reduction compared to natural ordering and 45× reduction compared to random relabeling for 64 processors) in Table 1. However, it is of little use with synthetic Kronecker graphs.

Graph	$p = 4 \times 1$			$p = 16 \times 1$			$p = 64 \times 1$		
	N	R	P	N	R	P	N	R	P
kron-simple-logn18	1.04	1.06	1.56	1.22	1.16	1.57	1.63	1.42	1.92
coAuthorsDBLP	1.84	1.01	1.39	2.58	1.05	1.85	3.27	1.13	2.43
eu-2005	1.37	1.10	1.05	3.22	1.28	3.77	**7.35**	1.73	**9.36**
coPapersCiteseer	1.46	1.01	1.23	1.81	1.02	1.76	2.36	1.07	2.44
delaunay_n20	2.36	1.03	1.71	3.72	1.13	3.90	**6.72**	1.36	**8.42**
rgg_n_2_20_s0	1.94	1.09	2.11	4.89	1.21	6.00	**9.04**	1.49	**13.34**

Table 3. Ratio of $P \cdot MaxVolume$ to *TotalVolume* for 1D row-wise partitioning (lower is better).

Table 2 shows that 2D checkerboard partitioning generally decreases total communication volume for random and PaToH orderings. However, when applied to natural ordering, 2D generally increases the communication volume, with the exception of `kron-simple-logn18` graph. For synthetic Kronecker graphs,

the communication reduction of using a 2D partitioning is proportional to the number of processors, but for other graphs a clear trend is not visible.

Graph	$p = 2 \times 2$			$p = 4 \times 4$			$p = 8 \times 8$		
	N	R	P	N	R	P	N	R	P
kron-simple-logn18	1.31	1.31	2.08	1.14	1.12	1.90	1.12	1.09	1.93
coAuthorsDBLP	1.51	1.28	1.28	2.51	1.08	1.81	4.57	1.12	1.97
eu-2005	1.70	1.32	1.78	3.38	1.15	3.25	8.55	1.19	8.58
coPapersCiteseer	1.38	1.29	1.20	2.66	1.07	1.59	4.82	1.04	2.12
delaunay_n20	1.40	1.30	1.77	3.22	1.12	4.64	**8.80**	1.18	**11.15**
rgg_n_2_20_s0	3.44	1.31	2.38	8.25	1.13	6.83	**53.73**	1.18	**17.07**

Table 4. Ratio of $P \cdot MaxVolume$ to $TotalVolume$ for 2D checkerboard partitioning (lower is better).

$(P \cdot MaxVolume)/\ TotalVolume$ metric shown in Tables 3 and 4 show the expected slowdown due to load imbalance in per-processor communication. This is an understudied metric that is not directly optimized by partitioning tools. Random relabeling of the vertices result in partitions that are load-balanced per iteration; no processor does too much communication than the average. The maximum happens with the `eu-2005` matrix on 64 processors with 1D partitioning, but even there maximum is less than twice (1.73×) the average. By contrast, both natural and PaToH orderings suffer from imbalances, especially for higher processor counts.

Graph	$p = 4 \times 1$			$p = 16 \times 1$			$p = 64 \times 1$		
	N	R	P	N	R	P	N	R	P
kron-simple-logn18	1.03	1.02	1.01	1.10	1.08	1.02	1.29	1.21	1.02
coAuthorsDBLP	1.90	1.00	1.00	2.60	1.03	1.00	**3.40**	1.04	1.00
eu-2005	1.05	1.01	1.01	1.50	1.05	1.02	2.40	1.06	1.02
coPapersCiteseer	2.11	1.00	1.00	2.72	1.02	1.00	3.14	1.06	1.00
delaunay_n20	1.00	1.00	1.02	1.00	1.00	1.02	1.00	1.00	1.02
rgg_n_2_20_s0	1.01	1.00	1.03	1.02	1.00	1.02	1.02	1.00	1.02

Table 5. Edge count imbalance: $max_{i \in P}(m_i)/\ average_{i \in P}(m_i)$ with 1D row-wise partitioning (lower is better, 1 is perfect balance).

Tables 5 and 6 show the per-processor edge count (non-zero count in sparse matrix) load imbalance for 1D and 2D checkerboard partitionings, respectively. This measure affects memory footprint and local computation load balance. 1D row-wise partitioning gives very good edge balance for high-diameter graphs, which is understandable due to their local structure. This locality is not affected by any ordering either. For low-diameter graphs that lack locality, natural ordering can result in up to a 3.4× higher edge count on a single processor than the average. Both the random ordering and PaToH orderings seem to take care

Graph	$p = 2 \times 2$			$p = 4 \times 4$			$p = 8 \times 8$		
	N	R	P	N	R	P	N	R	P
kron-simple-logn18	1.03	1.01	1.01	1.06	1.04	1.03	1.15	1.11	1.03
coAuthorsDBLP	2.46	1.00	1.03	5.17	1.02	1.01	**10.33**	1.02	1.02
eu-2005	1.91	1.03	1.03	3.73	1.06	1.03	**9.20**	1.13	1.05
coPapersCiteseer	3.03	1.01	1.02	**7.43**	1.00	1.03	**15.90**	1.02	1.02
delaunay_n20	1.50	1.00	1.04	2.99	1.00	1.03	**5.99**	1.01	1.04
rgg_n_2_20_s0	2.00	1.00	1.04	4.01	1.00	1.04	8.05	1.01	1.03

Table 6. Edge count imbalance: $max_{i \in P}(m_i) / \ average_{i \in P}(m_i)$ with 2D checkerboard partitioning (lower is better, 1 is perfect balance).

of this issue, though. On the other hand, 2D checkerboard partitioning exacerbates load imbalance in the natural ordering. For both low and high diameter graphs, a high imbalance, up to $10 - 16\times$, may result with natural ordering. Random ordering lowers it to the 11% envelope and PaToH further reduces it to approximately $3 - 5\%$.

6.2 Impact of Partitioning on parallel execution time

We next study parallel performance on Hopper for some of the DIMACS graphs. To understand the relative contribution of intra-node computation and inter-node communication to the overall execution time, consider the Hopper microbenchmark data illustrated in Figure 3. The figure plots the aggregate bandwidth (in GB/s) with multi-node parallel execution (and four MPI processes per node) and a fixed data/message size. The collective communication performance rates are given by the total number of bytes received divided by the total execution time. We also generate a 'random memory references' throughput rate (to be representative of the local computational steps discussed in Section 2), and this assumes that we use only four bytes of every cache line fetched. This rate scales linearly with the number of sockets. Assigning appropriate weights to these throughput rates (based on the the communication costs reported in the previous section) would give us a lower bound on execution time, as this assumes perfect load balance.

We report parallel execution time on Hopper for two different parallel concurrencies, $p = 16$ and $p = 256$. Tables 7 and 8 give the serial performance rates (with natural ordering) as well as the relative speedup with different reorderings, for several benchmark graphs. There is a $3.5\times$ variation in serial performance rates, with the skewed-degree graphs showing the highest performance and the high diameter graphs `road_central` and `hugebubbles-00020` the least performance. For the parallel runs, we report speedup over the serial code with the natural ordering. Interestingly, the random-ordering variants perform best in all of the low-diameter graph cases. Comparing the relative speedups across low-diameter graphs, we see that the Metis-partitioned variant is comparable to the performance of the random variant only for `coPapersCiteseer`. For the rest of the graphs, the random variant is faster than Metis ordering by a large margin. The second half of the table gives the impact of checkerboard partitioning on the

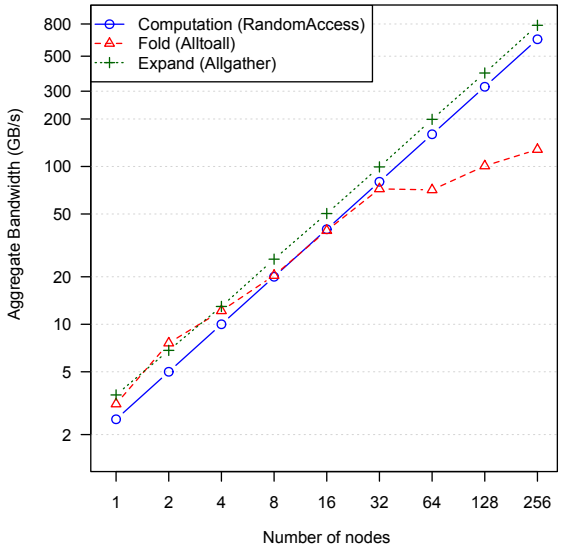


Fig. 3. Strong-scaling performance of collective communication with large messages and intra-node random memory accesses on Hopper.

running time. There is a moderate for the random variant, but the checkerboard scheme is slower for the rest of the schemes. The variation in relative speedup across graphs is also surprising. The synthetic low-diameter graphs demonstrate the best speedup overall. However, the speedups for the real-world low-diameter graphs are 1.5× lower, and the relative speedups for the high-diameter graphs are extremely low.

Figure 4 gives a breakdown of the average parallel BFS execution and inter-node communication times for 16-processor parallel runs, and provides insight into the reason behind varying relative speedup numbers. We also report the execution times with PaToH partitions in this case. For all the low-diameter graphs, at this parallel concurrency, execution time is dominated by local computation. The local discovery and local update steps account for up to 95% of the total time, and communication times are negligible. This is expected and in line with the microbenchmark rates in Figure 3. Comparing the computational time of random ordering vs. Metis reordering, we find that BFS on the Metis-reordered graph is significantly slower. The first reason is that Metis partitions are highly unbalanced in terms of the number of edges per partition for this graph, and so we can expect a certain amount of imbalance in local computation. The second reason is a bit more subtle. Partitioning the graph to minimize edge cut does not guarantee that the local computation steps will be balanced, even if the number of edges per process are balanced. The per-iteration work is dependent on the number of vertices in the current frontier and their distribution among processes. Randomizing vertex identifiers destroys any inherent locality, but also improves local computation load balance. The partitioning tools reduce

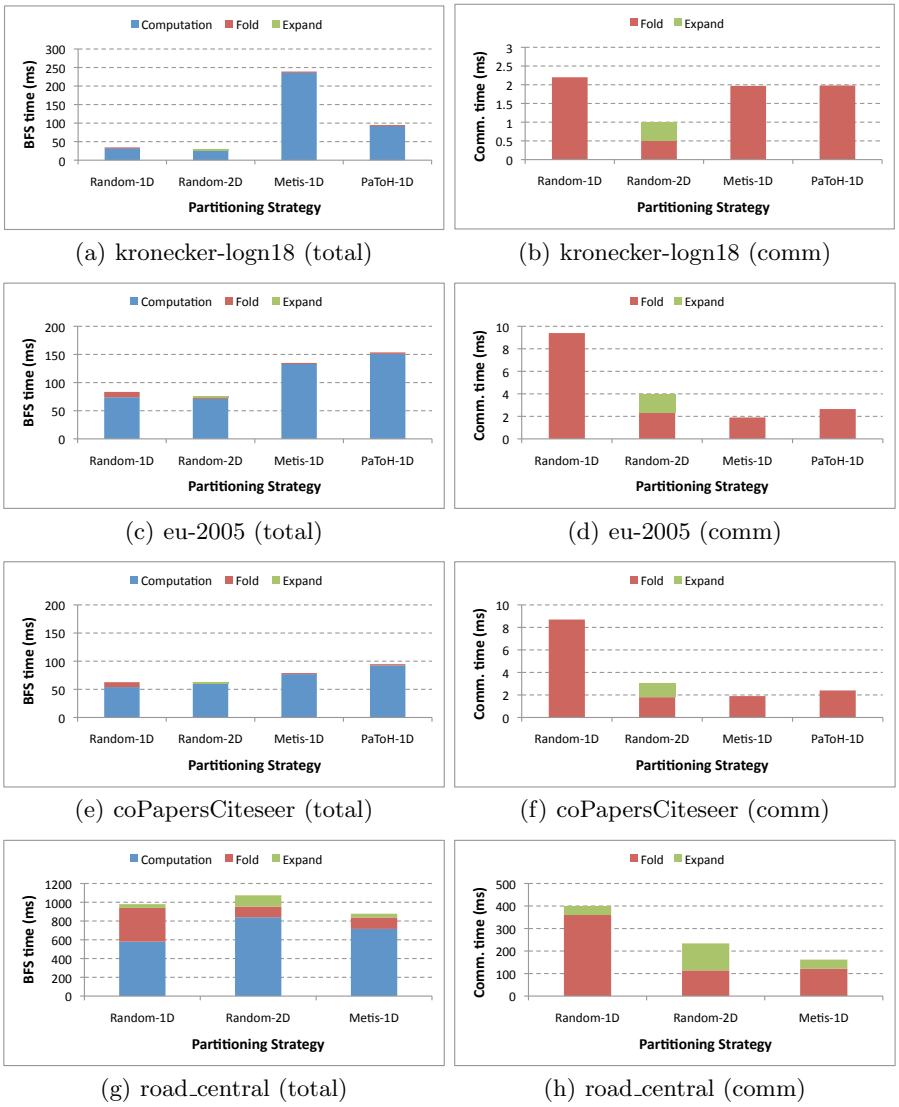
Graph	Perf Rate $p = 1 \times 1$	Relative Speedup $p = 16 \times 1$			Rel. Speedup over 1D $p = 4 \times 4$		
	N	N	R	M	N	R	M
coPapersCiteseer	24.9	5.6×	9.7×	8.0×	0.4×	**1.0**×	0.4×
eu-2005	23.5	6.1×	7.9×	5.0×	0.5×	**1.1**×	0.5×
kron-simple-logn18	24.5	12.6×	12.6×	1.8×	1.1×	**1.1**×	1.4×
er-fact1.5-scale20	14.1	11.2×	11.2×	11.5×	1.1×	**1.2**×	0.8×
road_central	7.2	3.5×	2.2×	3.5×	0.6×	0.9×	0.5×
hugebubbles-00020	7.1	3.8×	2.7×	3.9×	0.7×	0.9×	0.6×
rgg_n_2_20_s0	14.1	2.5×	3.4×	2.6×	0.6×	1.2×	0.6×
delaunay_n18	15.0	1.9×	1.6×	1.9×	0.9×	1.4×	0.7×

Table 7. BFS performance (in millions of TEPS) for single-process execution, and observed relative speedup with 16 MPI processes (four nodes, four MPI processes per node, 1 OpenMP thread per process). The fastest variants are highlighted in each case.

edge cut and enhance locality, but also seems to worsen load balance, especially for skewed degree distribution graphs. The PaToH-generated 1D partitions are much more balanced in terms of number of edges per process (than the Metis partitions for Kronecker graphs), but the average BFS execution still suffers from local computation load imbalance. Next consider the web crawl `eu-2005`. The local computation balance even after randomization is not as good as the synthetic graphs. One reason might be that the graph diameter is larger than the Kronecker graphs. 2D partitioning after randomization only worsens the load balance. The communication time for the fold step is somewhat lower for Metis and PaToH partitions compared to random partitions, but the times are not proportional to the savings projected in Table 3. This deserves further investigation. `coPapersCiteseer` shows trends similar to `eu-2005`. Note that the communication time savings going from 1D to 2D partitioning are different in both cases.

As seen in Table 7, the level-synchronous approach performs extremely poorly on high-diameter graphs, and this is due to a combination of reasons. There is load imbalance in the local computation phase, and this is much more apparent after Metis and PaToH reorderings. For some of the level-synchronous phases, there may not be sufficient work per phase to keep all 16 processes busy. In addition, just the barrier synchronization overhead is extremely high (See, for instance, the cost of the expand step with 1D partitioning for `road_central`. This should ideally be zero, because there is no data exchanged in expand for 1D partitioning. However, multiple barrier synchronizations of a few microseconds turn out to be a significant cost). Both Metis and PaToH generate extremely good partitions (very low edge cuts, well-balanced partitions), and this is reflected to some extent in the fold phase cost.

Table 8 gives the parallel speedup achieved with different reorderings at 256-way parallel concurrency. The Erdos-Renyi graph gives the highest parallel speedup for all the partitioning schemes, and they serve as an indicator of the speedup achieved with good computational load balance. The speedup for



(a) kronecker-logn18 (total) (b) kronecker-logn18 (comm)

(c) eu-2005 (total) (d) eu-2005 (comm)

(e) coPapersCiteseer (total) (f) coPapersCiteseer (comm)

(g) road_central (total) (h) road_central (comm)

Fig. 4. Average BFS execution time for various test graphs with 16 MPI processes (4 nodes, four MPI processes per node, 1 OpenMP thread per process).

real-world graphs is up to 5× lower than this value, indicating the severity of the load imbalance problem. One more reason for the poor parallel speedup may be that these graphs are smaller than the Erdos-Renyi graph. The communication cost increases in comparison to the 16-node case, but the computational cost comprises 80% of the execution time. The gist of these performance results is that for level-synchronous BFS, partitioning has a considerable effect on the computational load balance, in addition to altering the communication cost. On

current supercomputers, the computational imbalance seems to be the bigger of the two costs to account for, particularly at low process concurrencies.

Graph	Perf Rate $p = 1 \times 1$	Relative Speedup $p = 256 \times 1$			Rel. Speedup over 1D $p = 16 \times 16$		
	N	N	R	M	N	R	M
coPapersCiteseer	24.9	10.8×	13.7×	12.9×	0.5×	**2.5**×	0.7×
eu-2005	23.5	12.9×	14.7×	8.8×	0.6×	**2.7**×	0.6×
kron-simple-logn18	24.5	42.3×	41.9×	16.3×	2.6×	**2.6**×	0.3×
er-fact1.5-scale20	14.1	57.1×	58.0×	50.1×	1.6×	**1.6**×	1.1×
road_central	7.2	1.2×	0.7×	1.3×	1.9×	2.1×	1.1×
hugebubbles-00020	7.1	1.6×	1.5×	1.6×	1.5×	2.2×	2.0×
rgg_n_2_20_s0	14.1	1.5×	1.3×	1.6×	1.2×	1.2×	1.3×
delaunay_n18	15.0	0.6×	0.5×	0.5×	1.8×	1.9×	2.1×

Table 8. BFS performance rate (in millions of traversed edges per second) for single-process execution, and observed relative speedup with 256 MPI processes (64 nodes, four MPI processes per node, 1 OpenMP thread per process).

7 Conclusions

Our study highlights limitations of current graph and hypergraph partitioners for the task of partitioning graphs for distributed computations. Below, we list some crucial ones:

1. The frequently-used partitioning objective function, total communication volume, is not representative of the execution time of graph problems such as breadth-first search, on current distributed memory systems.
2. Even well-balanced vertex and edge partitions do not guarantee load-balanced execution, particularly for real-world graphs. We observe a range of relative speedups, between 8.8× to 50×, for low-diameter DIMACS graph instances.
3. Although random vertex relabeling helps in terms of load-balanced parallel execution, it can dramatically reduce locality and increase the communication cost to worst-case bounds.
4. Weighting the fold phase by a factor of two is not possible with two-phase partitioning strategies employed in current checkerboard method in PaToH, but it is possible with the single-phase fine grained partitioning. However, fine grained partitioning arbitrarily assigns edges to processors, resulting in communication among all processors instead of one processor grid dimension.

In future work, we plan to extend this study to consider additional distributed-memory graph algorithms. Likely candidates are algorithms whose running time is not so heavily dependent on the graph diameter.

References

1. Buluç, A., Madduri, K.: Parallel breadth-first search on distributed memory systems. In: Proc. ACM/IEEE Conference on Supercomputing (2011)
2. Çatalyürek, Ü.V., Aykanat, C.: PaToH: Partitioning Tool for Hypergraphs (2011)
3. Çatalyürek, Ü.V., Aykanat, C., Uçar, B.: On two-dimensional sparse matrix partitioning: Models, methods, and a recipe. SIAM J. Scientific Computing 32(2), 656–683 (2010)
4. The Graph 500 List. `http://www.graph500.org`, last accessed April 2011
5. Gregor, D., Lumsdaine, A.: The Parallel BGL: A Generic Library for Distributed Graph Computations. In: Proc. Workshop on Parallel/High-Performance Object-Oriented Scientific Computing (POOSC'05) (2005)
6. Karypis, G., Kumar, V.: Multilevel k-way partitioning scheme for irregular graphs. Journal of Parallel and Distributed Computing 48(1), 96–129 (1998)
7. Lipton, R.J., Rose, D.J., Tarjan, R.E.: Generalized nested dissection. SIAM J. Numer. Analysis 16, 346–358 (1979)
8. McCalpin, J.: Memory bandwidth and machine balance in current high performance computers. IEEE Tech. Comm. Comput. Arch. Newslett (1995)
9. Yoo, A., Chow, E., Henderson, K., McLendon, W., Hendrickson, B., Çatalyürek, Ü.V.: A scalable distributed parallel breadth-first search algorithm on BlueGene/L. In: Proc. ACM/IEEE Conf. on High Performance Computing (SC2005) (Nov 2005)