Kylix: A Sparse Allreduce for Commodity Clusters

Huasha Zhao
Computer Science Division
University of California
Berkeley, CA 94720
hzhao@cs.berkeley.edu

John Canny
Computer Science Division
University of California
Berkeley, CA 94720
jfc@cs.berkeley.edu

***Abstract*—Allreduce is a basic building block for parallel computing. Our target here is "Big Data" processing on commodity clusters (mostly sparse power-law data). Allreduce can be used to synchronize models, to maintain distributed datasets, and to perform operations on distributed data such as sparse matrix multiply. We first review a key constraint on cluster communication, the minimum efficient packet size, which hampers the use of direct all-to-all protocols on large networks. Our allreduce network is a nested, heterogeneous-degree butterfly. We show that communication volume in lower layers is typically much less than the top layer, and total communication across all layers a small constant larger than the top layer, which is close to optimal. A chart of network communication volume across layers has a characteristic "Kylix" shape, which gives the method its name. For optimum performance, the butterfly degrees also decrease down the layers. Furthermore, to efficiently route sparse updates to the nodes that need them, the network must be nested. While the approach is amenable to various kinds of sparse data, almost all "Big Data" sets show power-law statistics, and from the properties of these, we derive methods for optimal network design. Finally, we present experiments showing with Kylix on Amazon EC2 and demonstrating significant improvements over existing systems such as PowerGraph and Hadoop.**

***Keywords*—Allreduce; butterfly network; power law; big data;**

I. Introduction

Much of the world's "Big Data" is sparse: web graphs, social networks, text, clicks logs etc. Furthermore, these datasets are well fit by power-law models. By power-law, we mean that the frequency of elements in one or both (row and column) dimensions of these matrices follow a function of the form

$$F \propto r^{-\alpha} \tag{1}$$

where r is the rank of that feature in a frequency-sorted list of features [1]. These datasets are large: 40 billion vertices for the web graph, terabytes for social media logs and news archives, and petabytes for large portal logs. Many groups are developing tools to analyze these datasets on clusters [2]–[5], [5]–[8]. While cluster approaches have produced useful speedups, they have generally not leveraged single-machine performance either through CPU-accelerated libraries (such as Intel MKL) or using GPUs. Recent work has shown that very large speedups are possible on single nodes [9]. In fact, for many common machine learning problems single node benchmarks now dominate the cluster benchmarks that have appeared in the literature [9].

It is natural to ask if we can further scale single-node performance on clusters of fully-accelerated nodes. However, this requires proportional improvements in network primitives if the network is not to be a bottleneck. In this work we are looking to obtain an order of magnitude improvement in the throughput of the allreduce operation.

Allreduce is a general primitive that is integral to distributed graph mining and machine learning. In an Allreduce, data from each node, which can be represented as a vector (or matrix) v_i for node i, is reduced in some fashion (say via a sum) to produce an aggregate

$$v = \sum_{i=1,\ldots,m} v_i$$

and this aggregate is then shared across all the nodes.

In many applications, and in particular when the shared data is large, the vectors v_i are sparse. And furthermore, each cluster node may not require all of the sum v but only a sparse subset of it. We call a primitive which provides this capability a *Sparse Allreduce*. By communicating only those values that are needed by the nodes Sparse Allreduce can achieve orders-of-magnitude speedups over dense approaches.

Many Big Data analysis toolkits: GraphLab/Powergraph, Hadoop, Spark etc., all use direct all-to-all communication for the Allreduce operation. That is, every feature has a home node and all updates to that features are forwarded to the home node. The updates are accumulated and then sent to all the nodes who request the new value of that feature. Home nodes are distributed in balanced fashion across the network. Unfortunately this approach is not scalable: as the number of nodes m increases, the packet size for each message decreases as $1/m$ assuming fixed data per node. If instead the total dataset size is fixed, then message size decreases as $1/m^2$. Eventually, the time to send each message hits a floor value determined by overhead in the TCP stack and switch latencies. We show later that this limit is easily hit in practice, and is present in published benchmarks on these systems.

The aim of this paper is to develop a general efficient sparse allreduce primitive for computing on commodity clusters. The solution, called "Kylix", is a nested, heterogeneous-degree butterfly network. By heterogeneous we mean that the butterfly degree d differs from one layer of the network to another. By nested, we mean that values pass "down" through the network to implement an scatter-reduce, and then back up through the same nodes to implement an allgather. Nesting allows the return routes to be collapsed down the network, so that communication is greatly reduced in the lower layers. Because reduction happens in stages, the total data and communication volume in each layer almost always decreases (caused by

collapsing of sparse elements with the same indices). This network works particularly well with power-law data which have high collision rates among the high-frequency head terms. From an analysis of power-law data, we give a method to design the optimal network (layer degrees and number of layers) for a given problem.

We next show how sparse allreduce primitives are used in algorithms such as PageRank, Spectral Clustering, Diameter Estimation, and machine learning algorithms that train on blocks (mini-batches) of data, e.g. those that use Stochastic Gradient Descent(SGD) or Gibbs samplers.

A. Applications

1) MiniBatch Machine Learning Algortihms: Recently there has been considerable progress in sub-gradient algorithms [10], [11] which partition a large dataset into mini-batches and update the model using sub-gradients. Such models achieve many model updates in a single pass over the dataset, and several benchmarks on large datasets show convergence in a single pass [10]. Most machine learning models are amenable to subgradient optimization, and in fact it is often the most efficient method for moderate accuracy. Finally, MCMC algorithms such as Gibbs samplers involve updates to a model on every sample. To improve performance, the sample updates are batched in very similar fashion to sub-gradient updates [12].

All these algorithms have a common property in terms of the input mini-batch: if the mini-batch involves a subset of features , then a gradient update commonly uses input only from, and only makes updates to, the subset of the model that is projected onto those features. This is easily seen for factor and regression models whose loss function has the form

$$l = f(X_i v)$$

where X_i is the input mini-batch of the entire data matrix X, v is a vector or matrix which partly parametrizes the model, and f is in general a non-linear function. The derivative of loss, which defines the SGD update, has the form

$$dl/dv = f'(X_i v)X_i^T$$

from which we can see that the update is a scaled copy of X, and therefore involves the same non-zero features.

2) Graph Mining Algorithms: Many graph mining algorithms use repeated matrix/vector multiplication, which can be implemented using sparse allreduce: Each node i holds a subgraph whose adjacency matrix is X_i, and the input and output vectors are distributed using the allreduce. Each node requests vector elements for non-zero columns of X_i, and outputs elements corresponding to non-zero rows of X_i. Connected components, breadth-first search, and eigenvalues can be computed from such matrix-vector products. Diameter estimation algorithm in [13], the probabilistic bit-string vector is updated using matrix-vector multiplications.

To present one of these examples in a bit more detail: PageRank provides an ideal motivation for Sparse Allreduce. The PageRank iteration in matrix form is:

$$v' = \frac{1}{n} + \frac{n-1}{n} Xv \tag{2}$$

The dominant step is computing the matrix-vector product Xv. We assume that edges of adjacency matrix X are distributed across machines with X_i being the share on machine i, and that vertices v are also distributed (usually redundantly) across machines as v_i on machine i. At each iteration, every machine first acquires a sparse *input* subset v_i corresponding to non-zero columns of its share X_i - for a sparse graph such as a web graph this will be a small fraction of all the columns. It then computes the product $u_i = X_i v_i$. This product vector is also sparse, and its nonzeros correspond to non-zero *rows* of X_i. The input vertices v_i and the output vertices u_i are passed to a sparse (sum) Allreduce, and the result loaded into the vectors v_i' on the next iteration will be the appropriate share of the matrix product Xv. Thus a requirement for Sparse Allreduce is that we be able to specify a vertex subset going in, and a different vertex set going out (i.e. whose values are to be computed and returned).

B. Main Contributions

The key contributions of this paper are the following:

- **Kylix**, a general and scalable sparse allreduce primitive that supports big data analysis on commodity clusters.
- A design workflow for selecting optimal parameters for the network for a particular problem.
- A replication scheme that provides a high-degree of fault-tolerance with modest overhead.
- Several experiments on large datasets. Experimental results suggest that Kylix (heterogeneous butterfly) is 3-5x faster than direct all-to-all communication (our implementation) on typical datasets. The gains for Kylix over other systems for this benchmark are somewhat higher: 3-7x.

Kylix is modular and can be run self-contained using shell scripting (it does not require an underlying distributed middleware like Hadoop or MPI). Our current implementation is in pure Java, making it easy to integrate with Java-based cluster systems like Hadoop, HDFS, Spark, etc.

The rest of the paper is organized as follows. Section II reviews existing allreduce primitives for commodity data clusters, and highlights their scalability difficulties. Section III introduces Kylix, its essential features, and an example network. A design workflow for choosing its optimal parameters is discussed in Section IV. Section V and VI describe fault tolerance and optimized implementation of Kylix respectively. Experimental results are presented in Section VII. We summarize related works in Section VIII, and finally Section IX concludes the paper.

II. Background: Allreduce on Clusters

Cloud computing is a cost-effective, scalable technology with many applications in business and the sciences. Cloud computing is built on commodity hardware - inexpensive computer nodes and moderate performance interconnects. Cloud hardware may be private or third party as typified by Amazone EC2, Microsoft Azure and Google Cloud Platform. Cloud

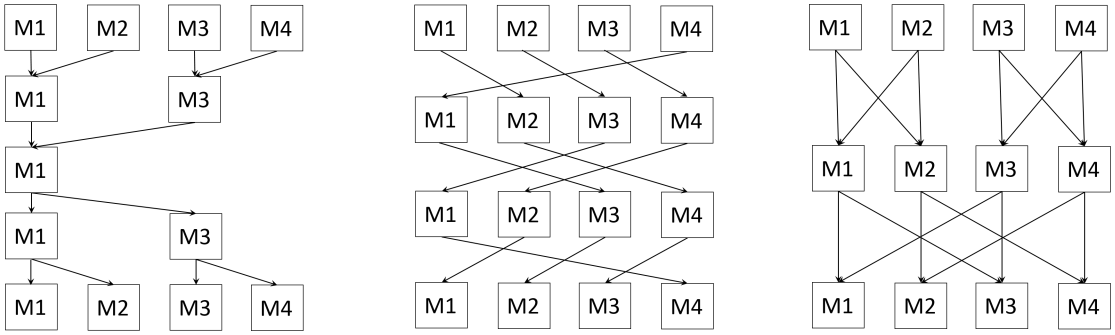


Fig. 1: Allreduce Topologies

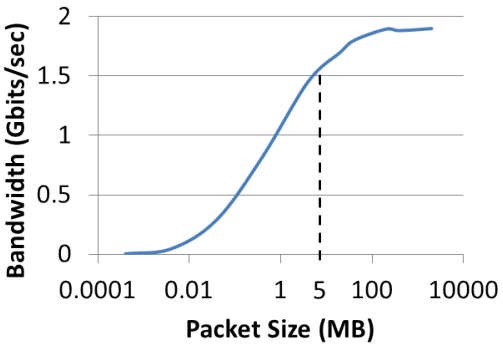


Fig. 2: Bandwith vs. packets size

computers can be configured as parallel or distributed computing services but systems that use them must deal with the realities of cloud computing: (i) variable compute node performance and external loads (ii) networks with modest bandwidth and high (and variable) latency and (iii) faulty nodes and sometimes faulty communication and (iv) power-law data for most applications. Kylix addresses these challenges. Like many cloud systems, Kylix is "elastic" in the sense that its size and topology can be adapted to the characteristics of particular sparse workloads. It is fault-tolerant and highly scalable. We next discuss the tradeoffs in its design.

A. AllReduce

Allreduce is commonly implemented with 1) tree structure [14], 2) direct communications [15] or 3) butterfly topologies [16].

1) Tree Allreduce: The tree reduce topology is illustrated in Figure 1(a). The topology uses the lowest overall bandwidth for atomic messages, although it effectively maximizes latency since the delay is set by the slowest path in the tree. It is a reasonable solution for small, dense (fixed-size) messages. A serious limitation for sparse allreduce applications is that intermediate reductions grow in size - while some sparse terms collapse, others do not. The middle (full reduction) node will have complete (fully dense) data which will often be intractably large. It also has no fault tolerance and extreme sensitivity to node latencies anywhere in the network. It is not practical for the problems of interest to us, and we will not discuss it further.

2) Direct All-to-All Allreduce: In direct all-to-all allreduce (from now on we will use the term "direct allreduce" for this method), each processor communicates with all other processors. Every feature is first sent to a unique home node where it is aggregated, and the aggregate is then broadcast to all the nodes. Random partitioning of the feature set is commonly used to balance communication. Messages are typically scheduled in a circular order, as presented in Figure 1(b). Direct allreduce achieves asymptotically optimal bandwidth, and optimal latency *when packets are sufficiently large* to mask setup-teardown times. But there is a minimum efficient packet size that must be used, or network throughput drops (see figure 2). It is easy to miss this target size on a large network, and there is no way to tune the network to avoid this problem. Also, the very large (quadratic in m) number of messages make this network more prone to failures due to packet corruption, and sensitive to latency outliers.

In our experiment setup of a 64-node (`cc2.8xlarge`) Amazon EC2 cluster with 10Gb/s inter-connect, the smallest efficient packets size is 5M to mask message sending overhead (Figure 2). For smaller packets, latency dominates and throughput will drop. Similar observations have also been discussed in [17], [18]. In fact, scaling the cluster much beyond this limit actually *increases* the total communication time because of the increasing number of messages, reversing the advantages of parallelism.

3) Butterfly Network: In a butterfly network, every node computes a reduction of values from its *in* neighbours (including its own) and outputs to its *out* neighbours. In the binary case, the neighbours at layer i lie on the edges of hypercube in dimension i with nodes as vertices. The cardinality of the neighbor set is called the degree of that layer. Figure 1 demonstrates a 2×2 butterfly network and Figure 3 shows a 3×2 network. A binary butterfly gives the lowest latency for allreduce operations *when messages have fixed cost*.

In a generalized butterfly, nodes in layer i are partitioned into groups of size d_i and allreduce is performed within each group using direct allreduce. Equivalently, the $\prod d_i$ nodes can be laid out on a unit grid within a hyper-rectangle of length

d_i along dimension i. Then a reduction in layer i is reduction along dimension i.

B. Partitions of Power-Law Data

It is known that most big data (power-law or "natural graph" data [19], [20]) is difficult to partition. Thus distributed algorithms on these graphs are communication-intensive, and it is very important to have an efficient allreduce. For matrix multiply, it was shown in [3], [21], edge partitioning is more effective for power-law datasets than vertex partitioning (vertices represent row/column indices and edges represent nonzeros of the matrix) . Paper [3] describes two edge partitioning schemes, one random and one greedy. Here we will only use random edge partitioning - the precomputation needed to partition is quite significant compared to the application running time (e.g. PowerGraph takes 300s for configuration and 3.6s for runtime per iteration in [3]).

III. Sparse Allreduce

A sparse allreduce operation for an n-vector on a network of m nodes should have the following properties:

1) Each network node $i \in \{1, \ldots, m\}$ specifies a set of input indices $in_i \subseteq \{1, \ldots, n\}$ that it would like to receive, and a set of output indices $out_i \subseteq \{1, \ldots, n\}$ that it would like to reduce data into.
2) Node i has a vector of values $vout_i$ (typically in $\mathbb{R}^{|out_i|}$) that correspond to the indices out_i. It pushes these values into the network and receives reduced values vin_i corresponding to the indices in_i that it has asked for.

In general, in_i and out_i will be different. It must be the case that $\cup_i in_i \subseteq \cup_i out_i$ or there will be some input nodes with no data to draw from. This is typically ensured by the type of calculation: for pagerank the indices are fixed and the union of row indices should cover $\{1, \ldots, n\}$. For distributed models, every model feature should have a "home machine" which always sends and receives that feature.

Steps 1 and 2 can be performed separately or together. We call step 1 *configuration* and step 2 *reduction*. For pagerank, step 1 is done just once (in and out vertex sets are fixed throughout the calculation), with step 2 performed on every iteration. For minibatch updates, the in and out vertices change on every allreduce. In that case, it is more efficient to do configuration and reduction concurrently with combined network messages. For our nested butterfly method, we define in addition:

- l is the number of layers of the network (layers of communication), and d_i the degree of the network at layer $i \in \{1, \ldots, l\}$. We will also slightly abuse notation to refer to "node layer" i, which is used to denote the contents of a node which are the results of communication layer i. Node layers are numbered $\{0, \ldots, l\}$ with 0 at the top.
- $m_j^i \subseteq \{1, \ldots, m\}$ is the set of neighbors of machine j at layer i, then $|m_j^i| = d_i$.
- in_k^i and out_k^i are the set of in (resp. out) indices hosted by node layer i. We initialize $in_k^0 = in_k$ and $out_k^0 = out_k$.
- in_{jk}^i and out_{jk}^i are the set of in (resp. out) indices sent from node j to node k at layer i during configuration.

A. Configuration

Configuration has a downward pass only (Figure 4 is helful in this discussion). In this pass, indices are partitioned in node layer $i-1$, transmitted in communication layer i, and merged (by union) in node layer i. More precisely:

$$(in^i_{jm^i_j(1)}, \ldots, in^i_{jm^i_j(k_i)}) = \text{partition}(in_j^{i-1}) \text{ and}$$

$$(out^i_{jm^i_j(1)}, \ldots, out^i_{jm^i_j(k_i)}) = \text{partition}(out_j^{i-1})$$

Partitioning is done into equal-size ranges of indices (this is unbalanced in general but we ensure that the original indices are hashed to the values used for partitioning). This ensures that all the indices merged in the node layer below lie in the same range, to maximize overlap.

Then the in_{jk}^i and out_{jk}^i index sets are sent to node k. Node k receives data from all its neighbors and computes the unions:

$$in_k^i = \bigcup_{j \in m_k^i} in_{jk}^i \text{ and } out_k^i = \bigcup_{j \in m_k^i} out_{jk}^i.$$

For efficiency, these union operations also generate maps $f_{jk}^i : \mathbb{N} \to \mathbb{N}$ from positions in the in-feature index sets in_{jk}^i into their union in_k^i, and $g_{jk}^i : \mathbb{N} \to \mathbb{N}$ from positions in the out-feature sets into their unions. These are used during reduction to add and project vectors in constant time per element.

B. Reduction

Reduction involves both a downward pass and an upward pass. The downward pass proceeds from layer $i = 1, \ldots, l$.

In the downward pass, values $vout_{jk}^i$ are sent from node j to node k in layer i. These values correspond to the indices out_{jk}^i.

As values from its layer-i neighbors are received by node k, they are summed into the total v_k^i for node k using the index map f_{jk}^i.

After l such steps, layer l contains a single copy of fully-reduced data distributed across all the nodes.

The upward pass proceeds from layer $i = l, \ldots, 1$. We define $vin_k^l = vout_k^l$ for $k = 1, \ldots, m$ to start the upward pass.

The map g_{jk}^i is used to extract the vector of values vin_{jk}^i to be sent to node j by node k from vin_k^i.

When node j receives these vectors from all of its layer-i neighbors, it simply concatenates them to form vin_j^{i-1}.

Finally vin_j^0 is the desired reduced data for node j.

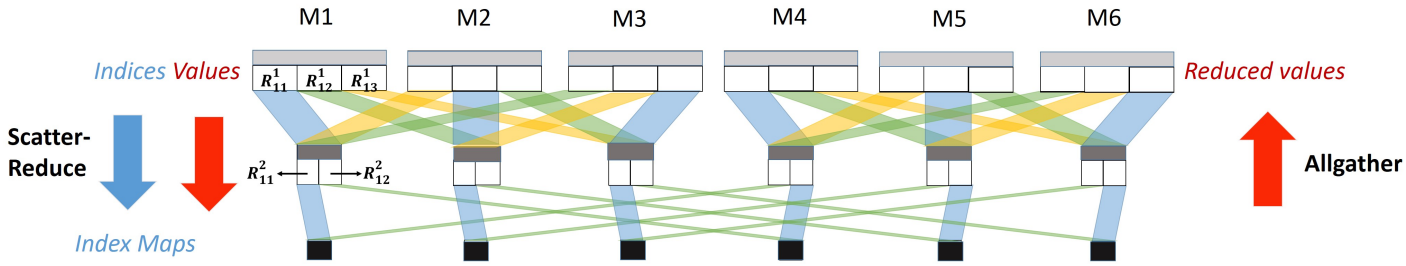


Fig. 3: Nested Sparse Allreduce within a heterogeneous-degree (3×2) butterfly network for 6 machines. Widths of the rectangles are proportional to the range lengths for that layer; R^i_{jk} is the range of vertices sent from machine j to k at layer i. The densities (proportion of non-zeros) of the ranges are color-coded; darker indicates higher density.

C. *Configreduce*

Configuration and Reduction can be performed in a single down-up pass through the network. During the downward pass from $i = 1, \ldots, l$, configuration is done on layer i, and then the downward reduction step for layer i. When we reach the bottom layer, we perform the upward reduction steps as usual. Combined configure/reduce are used for communication in algorithms where in and out vertices change on every allreduce, such as minibatch updates.

D. *Heterogeneous Degree Butterfly*

The goal of network design is to make the allreduce operation as efficient as possible. The first goal is to minimize the number of layers since layers increase latency. The total network size is the product of the layer degrees, so the larger the degrees, the fewer layers we will need. We adjust d_i for each layer to the largest value that avoids saturation (packet sizes below the minimum efficient size discussed earlier). Figure 3 illustrates a 3×2 network, where each processor talks to 3 neighbors in layer 1 and 2 neighbors in layer 2. In direct allreduce, packet size in each round of communication is constrained to be E_m/m where E_m is the data size on each machine. This may be too small - smaller than the minimum efficient packet. For example, in the Twitter followers' graph, the packet size is around 0.4 MB in a 64 node direct allreduce network. Figure 3 suggests that the smallest efficient packet is around 5MB for EC2, and that the B/W for 0.4MB packets is only about 30% of the efficient packet B/W.

The heterogeneity of layer degrees allows us to tailor packet size layer-by-layer. For the first layer, we know the amount of data at each node (total input data divided by m), and we can chose d_1 accordingly. It should be the largest integer such that (node MB)/d is larger than 5MB. Later layers involve intervals of merged sparse data. The total amount of data decreases layer-by-layer because of sparse index set overlap during merges. But to determine these sizes analytically, we need to do be able to model the expected number of merged indices. We do this for power-law data next.

IV. Tuning Layer Degrees for Power-Law Data

In this section, we describe a systematic method to determine the degrees of the network for optimal performance on power-law data.

Assume the frequency of rank-ordered features in the data vector follows a Poisson distribution with power-law rate. That is,

$$f_r \sim Poisson(\lambda r^{-\alpha}), \tag{3}$$

where r is the feature rank in descending order of frequency, α is the exponent of the power-law rate and λ is a scaling factor. Larger λ gives a denser vector. Let λ_0 be the scaling factor for the initial random partition of data across m machines. The expected message size for communication at each layer of butterfly network can be determined by the following proposition.

Proposition 4.1: Given a dataset of n features, and a sparse allreduce butterfly network of degrees $d_1 \times d_2 \cdots \times d_l$. Define $d_0 = 1$ and $K_i = \prod_{j=0}^{i} d_i$. The vector density, i.e. the proportion of non-zero features in the vector to be reduced for each machine at layer i is

$$D_i = \frac{1}{n} \sum_{r=1}^{n} \left(1 - \exp\left(-K_i \lambda_0 r^{-\alpha}\right)\right). \tag{4}$$

And the message size is

$$P_i = \frac{1}{K_i} \sum_{r=1}^{n} \left(1 - \exp\left(-K_i \lambda_0 r^{-\alpha}\right)\right). \tag{5}$$

Proof: Denote X_r the indicator function of the event the r^{th} feature occurs at least once in the vector at the initial partition, then

$$Pr(X_r = 1) = 1 - \exp\left(-\lambda_0 r^{-\alpha}\right) \tag{6}$$

according to the Poisson distribution. Then the scaling factor λ_0 is implicitly determined by the density of the initial partition at each node which is measurable:

$$\begin{aligned} D_0 &= \frac{1}{n} \mathbb{E}\left(\sum_{r=1}^{n} X_r\right) \\ &= \frac{1}{n} \sum_{r=1}^{n} 1 \cdot Pr(X_r = 1) + 0 \cdot Pr(X_r = 0) \\ &= \frac{1}{n} \sum_{r=1}^{n} \left(1 - \exp\left(-\lambda_0 r^{-\alpha}\right)\right) := f(\lambda_0). \end{aligned} \tag{7}$$

Notice that the density D is a function of only the scaling factor λ given the number of features n. So we could define $D := f(\lambda)$ according to Equation 7.

At the i^{th} layer of the network, the Poisson rate of each feature becomes $K_i\lambda_0 r^{-\alpha}$ and the scaling factor becomes $K_i\lambda_0$ since data there is a sum of data from K_i nodes. The density at layer i is $D_i = f(K_i\lambda_0)$.

At layer i the range of the vector to be reduced is $\frac{n}{K_i}$ according to our index partition. Then $P_i = D_i\frac{n}{K_i}$, which gives Equation 5. ∎

Figure 4 plots the density function f with regards to normalized scaling factor $\hat{\lambda}$, for different α. The scaling factor in the figure is normalized by $\lambda_{0.9}$, where $f(\lambda_{0.9}) = 0.9$ for the purpose of better presentation. We also zoom in the figure to show the relationship between density and λ for very low densities. Notice that the shape of the curve has only a modest dependence on α (α concentrates from 0.5 to 2 for most real word datasets).

To use Figure 4:

- Measure the density of the input data (could be either in our out features), i.e. the fraction of non-zeros over the number of features n. Draw a horizontal line through this density on the y-axis of the curve.
- Read off the λ value from the x-axis for this density.
- Multiply this x-value by the layer degree, to give a new x-value.
- Read off the new density from the y-axis.

This gives the density of the next layer. To compute the optimal degree for that layer, we need to know the amount of data per node. That will be the length of the partition range at that node times the density. The partition range at each layer is the total data vector length divided by all the layer degrees above that layer. i.e. the expected data size at layer i is $P = nD/\prod_{j=1,\ldots,i-1} d_j$ where D is the density we just computed (Proposition 4.1). Given this data size, we find the largest d such that P/d is at least 5 MB. Given this new d we can repeat the process one layer below etc.

The same method can be used for other sparse datasets without power-law structure. It will be necessary to construct an approximate density curve similar to figure 4. This involves drawing p samples from the sparse set for various p, and measuring the density. A scaled version of p should be plotted on the x-axis, with the density on y.

V. Fault Tolerance

Machine failure is a reality of large clusters. We next describe a simple but relatively efficient fault tolerance mechanism using replication to deal with multiple node failures [22].

A. Data Replication

Our approach is to replicate by a replication factor s, the data on each node, and all messages. Thus data on machine i also appears on the replicas $m + i$ through $i + (s-1) * m$. Similarly every config and reduce message targeted at node j is also sent to replicas $m + j$ through $j + (s-1) * m$. When receiving a message expected from node j, the other replicas are also listened to. The first message received is used, and the other listeners are canceled. This protocol completes unless all the replicas in a group are dead. The expected number of node failures before this occurs is about $\sqrt{m}$ by the birthday paradox [23] for a system of replication factor 2.

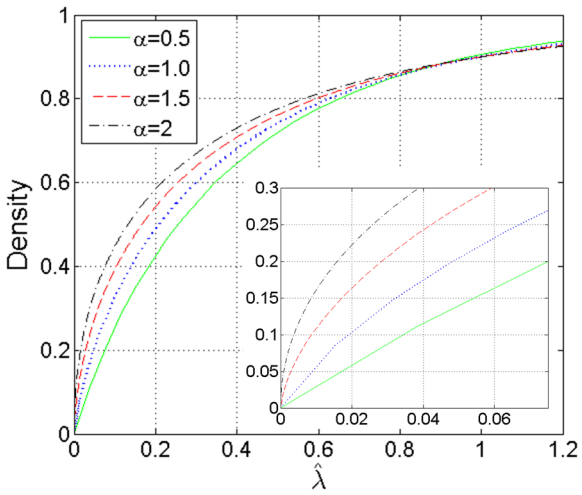


Fig. 4: Density curve for different α

B. Packets Racing

Replication by s increases per-node communication by s in the worst case (cancellations will reduce it somewhat). There is some performance loss because of this, as shown in the next section. On the other hand, replication offers potential gains on networks with high latency or throughput variance, because they create a race for the fastest response (in contrast to the non-replicate network which is instead driven by the *slowest* path in the network.

VI. Implementation

On 10 Gbit networks, the overhead of computing and memory access can dominate communication. It is important for all operations to be as fast as possible. We describe below some techniques to remove potential bottlenecks.

A. Tree Merging

The dominant step in Kylix is merging (union of) the index sets during configuration. This is a linear time operation using hash tables, but in reality the constants involved in random memory access are too high. Instead we maintain each index set in sorted order and use merging to combine them. The merged sets must be approximately equal in length or this will not be efficient (cost of a merge is the length of the longer sequence). So we use a tree-merge. Each sequence is assigned to a leaf of a full binary tree. Nodes are recursively merged with their siblings. This was 5x faster than a hash implementation.

B. Multi-Threading and Latency Hiding

Scientific computing systems typically maintain a high degree of synchrony between nodes running a calculation. In cluster environments, we have to allow for many sources of variability in node timing, latency and throughput. While our network conceptually uses synchronized messages to different destinations to avoid congestion, in practice this does not give the best performance. Instead we use multi-threading and communicate opportunistically. i.e. we start threads to send all messages concurrently, and spawn a thread to process each message that is received. In the case of replicated messages, once the first message of a replicate group is received, the other threads listening for duplicates are terminated and those copies discarded. Still, the network interface itself is a shared resource, so we have to be careful that excessive threading does not hurt performance through switching of the active message thread. The effects of thread count is shown in Figure 7.

C. Language and Networking Libraries

Kylix is currently implemented using standard Java sockets. We explored several other options including OpenMPI-Java, MPJexpress, and Java NIO. Unfortunately the MPI implementations lacked key features that we needed to support multi-threading, asynchronous messaging, cancellation etc., or these features did not work through the Java interface. Java NIO was simply more complex to use without a performance advantage. All of the features we needed were easily implemented with sockets, and ultimately they were a better match for the level of abstraction and configurability that we needed.

We acknowledge that the network interface could be considerably improved. The ideal choice would be RDMA over Ethernet (RoCE), and even better RoCE directly from GPUs when they are available. This feature in fact already exists (as GPUdirect for NVIDIA CUDA GPUs), but is currently only available for infiniband networks.

VII. Experiments

In this section, we evaluate Kylix and compare its performance with two other systems: Hadoop and PowerGraph. Two datasets are used:

1) Twitter Followers Graph. The graph consists of 60 million vertices and 1.5 billion edges.
2) Yahoo! Altavista web graph. This is one of the largest publicly available web graphs with 1.4 billion vertices and 6 billion edges.

The densities of the 64-way partitioned datasets are 0.21 and 0.035 respectively. All the experiments with Kylix are performed on the Amazon EC2 commodity cluster which comprises 64 `cc2.8xlarge nodes`. Each node has an eight-core Intel Xeon E5-2670 processor and all nodes are interconnected by 10Gb/s Ethernet.

A. Optimal Degrees

The optimal degrees are $8 \times 4 \times 2$, and 16×4 for the Twitter followers' graph and Yahoo web graph respectively. They are determined using the method discussed in Section IV. Figure 5 shows the total communication volume across the network (including packets to its own) for different layers. The last layer in the figure is the total volume of fully reduced values at the bottom layer of scatter-reduce (it is the communication volume if there were an additional layer for reduce). The total communication volume is decreasing from layer to layer. The Twitter graph shrinks very fast at lower layers, because vectors communicated are dense and the collision rate is close to a hundred percent. For the Yahoo's graph, the collision rate is much lower because it is more sparse, and the volume shrinking is less significant. The shape of the total communication volume by layer looks like a Kylix, which coins the name of our system.

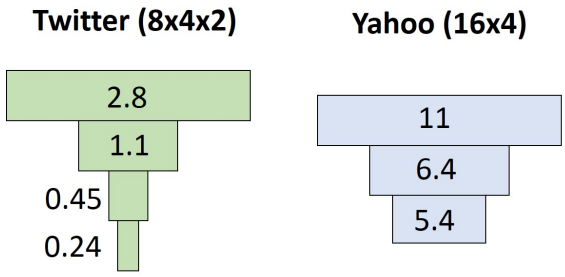


Fig. 5: Data volumes (GB) at each layer of the protocol, resembling a Kylix. These are also the total communication volumes for all but the last layer.

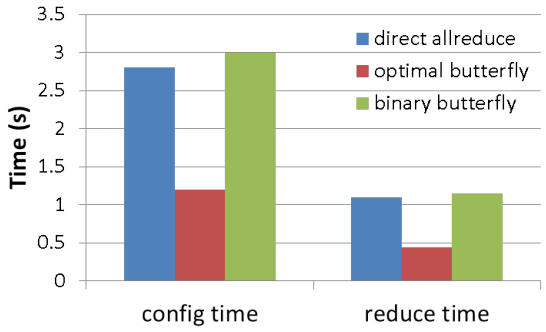
Figure 6 plots the average config time and reduce time per iteration for different network configurations, including, direct all-to-all communication, optimal butterfly and binary butterfly, for both graphs. From the figure, we can see that optimal butterfly is 3-5x faster than the other configurations. Optimal butterfly fully utilized the network bandwidth with tuned packet sizes, whereas, the direct allreduce topology sends 0.4MB of packets each round (for the Twitter graph) which is below the optimal packet size; this packet size utilizes about 30% of the full bandwidth as we can tell from Figure 2. Optimal butterfly is also faster than binary butterfly since it has fewer layers and both latency and size of replicated messages to be routed are reduced.

B. Effect of Multi-Threading

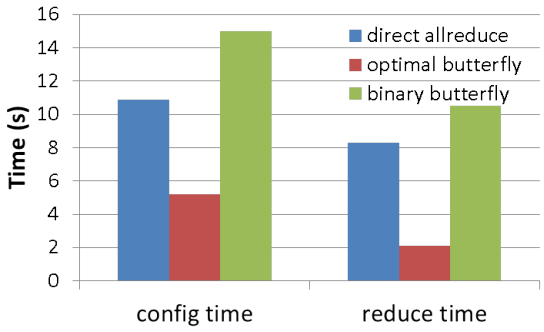
We compare the Allreduce runtime for different thread levels in Figure 7. All the results are run under the $8 \times 4 \times 2$ configuration. Significant performance improvement can be observed by increasing from single thread up to 4 threads, and it is also clear from the figure that the benefit of adding thread level is marginal beyond 16 threads (each machine has 16 CPU threads).

C. Performance with Fault Tolerance

Table I demonstrates the performance with data replication. As shown in the table, the impact of data replication on runtime is moderate. The first colum shows the performance of the optimal, unreplicated network on 64 nodes. The last four columns show the optimal replicated network on 64 nodes (which has data partition into 32 parts) with 0,1,2,3 failures. The second column shows an unreplicated 32-nnode network for reference purposes. Notice first that the runtime with



(a) Twitter followers' graph



(b) Yahoo web graph

Fig. 6: Allreduce time per iteration

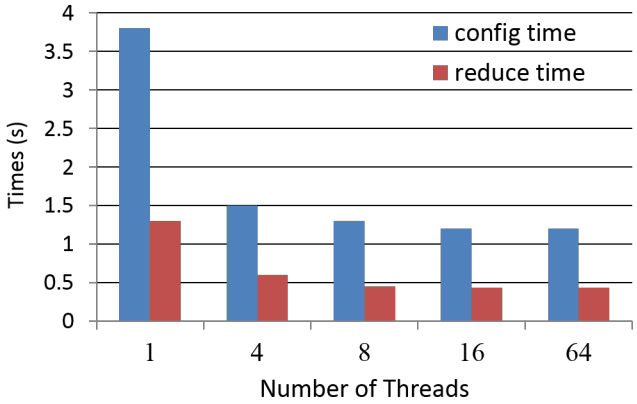


Fig. 7: Runtime comparison between different thread levels.

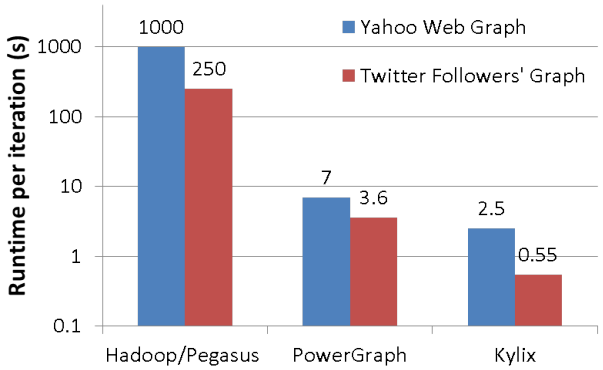


Fig. 8: PageRank runtime comparison (log scale).

failures is apparently independent of the number of failures. Notice second that replication increases the configuration time by only about 25%, and reduction time by about 60%. While the replicated network potentially does twice the work of the unreplicated network, it benefits from packet racing and early termination. For the network to fail completely, it would require about $\sqrt{m} = 8$ failures by our earlier analysis. This is very unlikely to happen for a 64-node cluster in real production settings.

We also compare the runtime for different number of machine failures in Table I. Runtimes with replication are the same regardless of the number of dead nodes up to 3 tested.

D. Performance and Scalability

We next compared Kylix with two other systems on the PageRank algorithm. PageRank is a widely benchmarked problem for Power-law graphs. Our version of PageRank is implemented using BIDMat+Kylix. BIDMat [9] is an interactive matrix library written in Scala that includes hardware acceleration (Intel MKL).

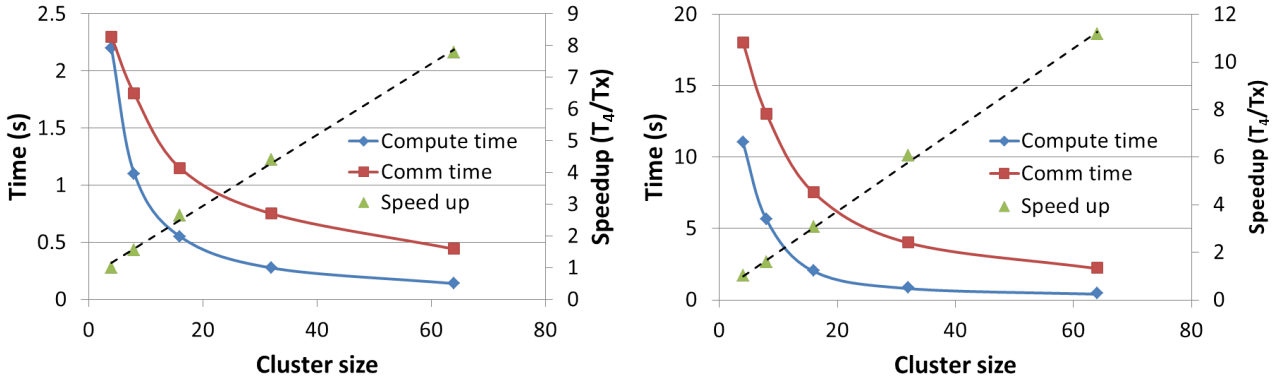
The scalability of Kylix is illustrated in Figure 9. The figure plots the runtimes (broken down into computation and communication time per iteration) against cluster sizes. It also plots the speed up over runtime of 4 nodes (4 is the minimal cluster size such that number of edges in each partition of Yahoo graph fits into the range of `int`), which is defined as $speedup = T_4/T_x$ for cluster of size x. The butterfly degrees are optimally tuned individually for different cluster sizes.

As shown in the figure, the speed up ratio is 7-11 on 64 nodes. The optimal speedup is 16, since we are comparing with runtime on 4 nodes as baseline. The system achieves roughly linear scaling with the number of machines in the cluster. Scaling beyond this should be possible, but from the graph it can be seen that communication starts to dominate the runtime for both datasets after 32 nodes. Particularly, for our 64-node experiments, communication takes up to 75-90% of overall runtime. However, there is no computational explanation for this and we believe it is actually due to lack of synchronization (which is absorbed in the communication time measurements) of the protocol across larger networks. We are currently exploring improvements on larger networks.

Finally, we compare our system with Hadoop/Pegasus and Powergraph. We chose Hadoop because it is a widely-known system, and Powergraph because it has demonstrated the highest performance to date for the Pagerank problem [3]. Figure 8 plots runtime per iteration for different systems. PowerGraph was run on a 64-node EC2 cluster with 10Gb/s interconnect - the same as this paper. Each Powergraph node has a two quad core Intel Xeon X5570 for the Twitter benchmark [3] and dual

TABLE I: Cost of Fault Tolerance

System Configuration	$8 \times 4 \times 2$ network replication=1 (64 nodes)	8×4 network replication=1 (32 nodes)	8×4 network replication=2 (64 nodes)	8×4 network replication=2 (64 nodes)	8×4 network replication=2 (64 nodes)	8×4 network replication=2 (64 nodes)
Number of dead nodes	0	0	0	1	2	3
Config time (s)	1.2	1.3	1.51	1.49	1.52	1.51
Reduce time (s)	0.44	0.60	0.75	0.73	0.76	0.74



(a) Twitter followers' graph

(b) Yahoo web graph

Fig. 9: Compute/Comm time break-down and speedup on a 64-node EC2 commodity cluster (512 cores in total)

8-core CPUs for the Yahoo benchmark [24]. Pegasus runs on a 90-node Yahoo M45 cluster. We estimate Pegasus runtime for Twitter and Yahoo graph by using their runtime result [6] on a power-law graph with 0.3 billion edges and assuming linear scaling in number of edges. We believe that the estimate is sufficient since we are only interested in the runtime in terms of order of magnitude for Hadoop-based system. The y-axis of the plot is log-scale. Kylix spends 0.55 second for on PageRank iterations on the Twitter followers' graph and 2.5 seconds for the Yahoo graph. From the graph it can be seen that Kylix runs 3-7x faster than PowerGraph, and about 500x times faster than Hadoop. We point out that the benchmark quoted for Pagerank on PowerGraph uses greedy partitioning which its authors note saves 50% runtime compared to the random partition we used [24]. Thus one would expect the performance ratio between Kylix and Powergraph to be closer to 6-14x when run on the same partioned dataset. The gap for Twitter (7x) is larger than for the Yahoo graph (3x). We expect this is because Twitter is a smaller dataset and direct messaging in Powergraph falls significantly below the efficient message minimum size. This suggests that the gap is likely to widen if calculations are run on larger clusters where message sizes will once again be small in direct allreduce.

It is worth pointing out that the overall achieved bandwidth for these experiments is around 3 Gb/s per node on EC2 which is somewhat lower than the rated 10Gb/s of the network. It is known that standard TCP/IP socket software has many memory-to-memory copy operations, whose overhead is significant at 10Gb/s. There are several technologies available which would better this figure, however at this time there are barriers to their use on commodity clusters. RDMA over Converged Ethernet would be an ideal technology. This technology bypasses copies in several layers of the TCP/IP stack and uses direct memory-to-memeory copies to deliver throughputs much closer to the limits of the network itself. It is available currently for GPU as GPUdirect (which communicates directly from on GPU's memory to another over a network), and in Java as Sockets Direct. However, at this time both these technologies are only available for Infiniband networks. We will monitor these technologies, and we also plan to experiment with some open source projects like RoCE (RDMA over Converged Ethernet) which offer more modest gains.

VIII. Related Work

Many other distributed learning and graph mining systems are under active development at this time [2]–[6], [8]. Our work is closest to the GraphLab [2] project. GraphLab improves upon the Hadoop MapReduce infrastructure by expressing asynchronous iterative algorithms with sparse computational dependencies. PowerGraph is a improved version of GraphLab, which particularly tackles the problem of power-law data in the context of graph mining and machine learning. We focus on optimizing the communication layer of the distributed learning systems, isolating the Sparse Allreduce primitive from other matrix and machine learning modules. There are a variety of other similar distributed data mining systems [13], [25], [26] built on top of Hadoop. However, the disk-caching and disk-buffering philosophy of Hadoop, along with heavy reliance on reflection and serialization, cause such approaches to fall orders of magnitude behind the other approaches discussed here. Finally, we also distinguish our work with other dense Allreduce systems such as [27].

Our work is also related with research in distributed SpMV [28], [29], distributed graph mining [21], [29] and optimized all-to-all communications [27], [30] in the scientific computing community. However, they usually deal with matrices with

regular shapes (tri-diagonal), matrices desirable partition properties such as small surface-to-volume ratio, or dense matrices. We are more focused on communicating intensive algorithms posed by sparse power law data which is hard to partition. We also distinguish our work by concentrating on studying the performance trade-off on commodity hardwares, such as on Amazon EC2, as opposed to scientific clusters featuring extremely fast network connections, high synchronization and exclusive (non-virtual) machine use.

IX. Conclusion

In this paper, we described Kylix, a Sparse Allreduce primitive for efficient and scalable distributed machine learning. The primitive is particularly well-adapted to the power-law data common in machine learning and graph analysis. We showed that the best approach is a hybrid between butterfly and direct all-to-all topologies, using a nested communication pattern and non-homogeneous layer degrees. We added a replication layer to the network which provides a high degree of fault tolerance with modest overhead. Finally, we presented a number of experiments exploring the performance of Kylix. We showed that it is significantly faster than other primitives. In the future we hope to achieve further gains by using more advanced network layers that use RDMA over Converged Ethernet (RoCE), and more attention to potential clock skew across the network. Our code is open-source and freely-available, and is currently in pure Java. It is distributed as part of the BIDMat suite, but can be run standalone without other BIDMat features.

References

[1] L. A. Adamic, "Zipf, power-laws, and pareto-a ranking tutorial," *Xerox Palo Alto Research Center, Palo Alto, CA, http://ginger. hpl. hp. com/shl/papers/ranking/ranking. html*, 2000.

[2] Y. Low, J. Gonzalez, A. Kyrola, D. Bickson, C. Guestrin, and J. M. Hellerstein, "Graphlab: A new framework for parallel machine learning," *arXiv preprint arXiv:1006.4990*, 2010.

[3] J. E. Gonzalez, Y. Low, H. Gu, D. Bickson, and C. Guestrin, "Powergraph: Distributed graph-parallel computation on natural graphs," in *Proc. of the 10th USENIX conference on Operating systems design and implementation, OSDI*, vol. 12.

[4] G. Malewicz, M. H. Austern, A. J. Bik, J. C. Dehnert, I. Horn, N. Leiser, and G. Czajkowski, "Pregel: a system for large-scale graph processing," in *Proceedings of the 2010 international conference on Management of data*. ACM, 2010, pp. 135–146.

[5] H. Zhao and J. Canny, "Butterfly mixing: Accelerating incremental-update algorithms on clusters," in *SIAM International Conference on Data Mining*, 2013.

[6] U. Kang, C. E. Tsourakakis, and C. Faloutsos, "Pegasus: A peta-scale graph mining system implementation and observations," in *Data Mining, 2009. ICDM'09. Ninth IEEE International Conference on*. IEEE, 2009, pp. 229–238.

[7] J. Ekanayake, H. Li, B. Zhang, T. Gunarathne, S.-H. Bae, J. Qiu, and G. Fox, "Twister: a runtime for iterative mapreduce," in *Proceedings of the 19th ACM International Symposium on High Performance Distributed Computing*. ACM, 2010, pp. 810–818.

[8] M. Isard, M. Budiu, Y. Yu, A. Birrell, and D. Fetterly, "Dryad: distributed data-parallel programs from sequential building blocks," *ACM SIGOPS Operating Systems Review*, vol. 41, no. 3, pp. 59–72, 2007.

[9] J. Canny and H. Zhao, "Big data analytics with small footprint: Squaring the cloud," in *ACM SIGKDD Conference on Knowledge Discovery and Data Mining (KDD)*, 2013.

[10] L. B. Y. Le Cun and L. Bottou, "Large scale online learning," *Advances in neural information processing systems*, vol. 16, p. 217, 2004.

[11] J. Duchi, E. Hazan, and Y. Singer, "Adaptive subgradient methods for online learning and stochastic optimization," *The Journal of Machine Learning Research*, vol. 12, pp. 2121–2159, 2011.

[12] A. Smola and S. Narayanamurthy, "An architecture for parallel topic models," *Proceedings of the VLDB Endowment*, vol. 3, no. 1-2, pp. 703–710, 2010.

[13] U. Kang, C. Tsourakakis, A. P. Appel, C. Faloutsos, and J. Leskovec, *Hadi: Fast diameter estimation and mining in massive graphs with hadoop*. Carnegie Mellon University, School of Computer Science, Machine Learning Department, 2008.

[14] J. Langford, L. Li, and A. Strehl, "Vowpal wabbit online learning project," 2007.

[15] J. Duato, S. Yalamanchili, and L. M. Ni, *Interconnection networks: An engineering approach*. Morgan Kaufmann, 2003.

[16] P. Patarasuk and X. Yuan, "Bandwidth optimal all-reduce algorithms for clusters of workstations," *Journal of Parallel and Distributed Computing*, vol. 69, no. 2, pp. 117–124, 2009.

[17] Z. Hill and M. Humphrey, "A quantitative analysis of high performance computing with amazon's ec2 infrastructure: The death of the local cluster?" in *IEEE Grid Computing*, 2009, pp. 26–33.

[18] E. Walker, "Benchmarking amazon ec2 for high-performance scientific computing," *Usenix Login*, vol. 33, no. 5, pp. 18–23, 2008.

[19] A. Abou-Rjeili and G. Karypis, "Multilevel algorithms for partitioning power-law graphs," in *Parallel and Distributed Processing Symposium, 2006. IPDPS 2006. 20th International*. IEEE, 2006, pp. 10–pp.

[20] J. Leskovec, K. J. Lang, A. Dasgupta, and M. W. Mahoney, "Community structure in large networks: Natural cluster sizes and the absence of large well-defined clusters," *Internet Mathematics*, vol. 6, no. 1, pp. 29–123, 2009.

[21] A. Yoo, E. Chow, K. Henderson, W. McLendon, B. Hendrickson, and U. Catalyurek, "A scalable distributed parallel breadth-first search algorithm on bluegene/l," in *Supercomputing, 2005. Proceedings of the ACM/IEEE SC 2005 Conference*. IEEE, 2005, pp. 25–25.

[22] R. Guerraoui and A. Schiper, "Fault-tolerance by replication in distributed systems," in *Reliable Software TechnologiesAda-Europe'96*. Springer, 1996, pp. 38–57.

[23] P. Flajolet, D. Gardy, and L. Thimonier, "Birthday paradox, coupon collectors, caching algorithms and self-organizing search," *Discrete Applied Mathematics*, vol. 39, no. 3, pp. 207–229, 1992.

[24] J. Gonzalez, "Thesis defense presentation," 2012. [Online]. Available: http://www.cs.berkeley.edu/~jegonzal/talks/jegonzal_thesis_defense.pptx

[25] "Mahout," *http://mahout.apache.org/*.

[26] Y. Bu, B. Howe, M. Balazinska, and M. D. Ernst, "Haloop: Efficient iterative data processing on large clusters," *Proceedings of the VLDB Endowment*, vol. 3, no. 1-2, pp. 285–296, 2010.

[27] A. R. Mamidala, J. Liu, and D. K. Panda, "Efficient barrier and allreduce on infiniband clusters using multicast and adaptive algorithms," in *Cluster Computing, 2004 IEEE International Conference on*. IEEE, 2004, pp. 135–144.

[28] J. Demmel, M. Hoemmen, M. Mohiyuddin, and K. Yelick, "Avoiding communication in sparse matrix computations," in *Parallel and Distributed Processing, 2008. IPDPS 2008. IEEE International Symposium on*. IEEE, 2008, pp. 1–12.

[29] A. Buluç and J. R. Gilbert, "The combinatorial blas: Design, implementation, and applications," *International Journal of High Performance Computing Applications*, vol. 25, no. 4, pp. 496–509, 2011.

[30] S. Kumar, Y. Sabharwal, R. Garg, and P. Heidelberger, "Optimization of all-to-all communication on the blue gene/l supercomputer," in *Parallel Processing, 2008. ICPP'08. 37th International Conference on*. IEEE, 2008, pp. 320–329.