Distributed Machine Learning and Graph Processing with Sparse Matrices

Paper #83

Abstract

It is cumbersome to write machine learning and graph algorithms in data-parallel models such as MapReduce and Dryad. We observe that these algorithms are based on matrix computations and, hence, are inefficient to implement with the restrictive programming and communication interface of such frameworks.

In this paper we show that array-based languages such as R [2] are suitable for implementing complex algorithms and can outperform current data parallel solutions. Since R is single threaded and does not scale to large datasets, we have built Pronto, a distributed system that extends R and addresses many of its limitations. Pronto efficiently shares sparse structured data, can leverage multi-cores, and dynamically partitions data to mitigate load imbalance. Our results show the promise of this approach: many important machine learning and graph algorithms can be expressed in a single framework and are substantially faster than those in Hadoop and Spark.

1. A matrix based approach

Many real-world applications require sophisticated analysis on massive datasets. Most of these applications use machine learning, graph algorithms, and statistical analyses that are easily expressed as matrix operations.

For example, PageRank corresponds to the dominant eigenvector of a matrix `G` that represents the Web graph. It can be calculated by starting with an initial vector `x` and repeatedly performing `x=G*x` until convergence [7]. Similarly, recommendation systems in companies like Netflix are implemented using matrix decomposition [34]. Even graph algorithms, such as shortest path, centrality measures, strongly connected components, etc., can be expressed using operations on the matrix representation of a graph [18].

Array-based languages such as R and MATLAB provide an appropriate programming model to express such machine learning and graph algorithms. The core construct of arrays makes these languages suitable to represent vectors and matrices, and perform matrix computations. R has thousands of freely available packages and is widely used by data miners and statisticians, albeit for problems with relatively small amounts of data. It has serious limitations when applied to very large datasets: limited support for distributed processing, no strategy for load balancing, no fault tolerance, and is constrained by a server's DRAM capacity.

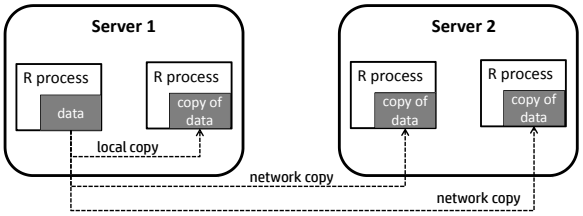


Figure 1. R's poor multi-core support: multiple copies of data on the same server and high communication overhead across servers.

1.1 Towards an efficient distributed R

We validate our hypothesis that R can be used to efficiently execute machine learning and graph algorithms on large scale datasets. Specifically, we tackle the following challenges:

Effective use of multi-cores. R is single threaded. The easiest way to incorporate parallelism is to execute programs across multiple R processes. Existing solutions for parallelizing R [23] use message passing techniques, including network communication, to communicate among processes. This multi-process approach is also used in commercial systems like parallel MATLAB and has two limitations. First, it results in redundant data copies. Figure 1 shows that two R instances on a single physical server would have two copies of the same data, hindering scalability to larger datasets. Second, the network communication overhead becomes proportional to the number of cores utilized instead of the number of distinct servers, again limiting scalability.

Existing efforts for parallelizing R have another limitation. They do not support point-to-point communication. Instead data has to be moved from worker processes to a designated master process after each phase. Thus, it is inefficient to execute anything that is not embarrassingly parallel [23]. Even simple iterative algorithms are costly due to the communication overhead via the master.

Imbalance in sparse computations. Most real-world datasets are *sparse*. For example, the Netflix prize dataset

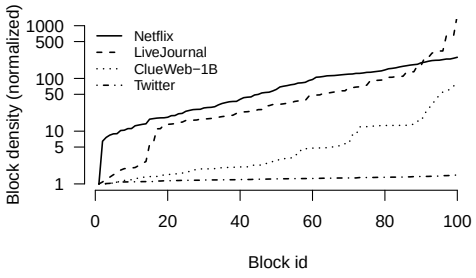


Figure 2. Variance in block density. Y-axis shows density of a block normalized by that of the sparsest block. Lower is better.

is a matrix with 480K users (rows) and 17K movies (cols) but only 100 million of the total possible 8 billion ratings are available. Similarly, very few of the possible edges are present in Web graphs. It is important to store and manipulate such data as sparse matrices and retain only non-zero entries. These datasets also exhibit skew due to the power-law distribution [13], resulting in severe computation and communication imbalance when data is partitioned for parallel execution. Figure 2 illustrates the result of naïve partitioning of various sparse data sets: LiveJournal (68M edges) [3], Twitter (280M edges), pre-processed ClueWeb sample[1] (1.2B edges), and the ratings from Netflix prize (100M ratings). The y-axis represents the block density relative to the sparsest block, when each input matrix is partitioned into 100 blocks. The plot shows that a dense block may have 1000× more elements than a sparse block. Depending upon the algorithm, variance in block density can have a substantial impact on performance (Section 6).

1.2 Limitations of current data parallel approaches

Existing distributed data processing frameworks, such as MapReduce and DryadLINQ, simplify large-scale data processing [11, 16]. Unfortunately, the simplicity of the programming model (as in MapReduce) or reliance on relational algebra (as in DryadLINQ) makes these systems unsuitable for implementing complex algorithms based on matrix operations. Current systems either do not support stateful computations, or do not retain the structure of global shared data (e.g., mapping of data to matrices), or do not allow point to point communication (e.g., restrictive MapReduce communication pattern). Such shortcomings in the programming model have led to inefficient implementations of algorithms or the development of domain specific systems. For example, Pregel was created for graph algorithms because MapReduce passes the entire state of the graph between steps [22].

There have been recent efforts to better support large-scale matrix operations. Ricardo [10] and HAMA [28] convert matrix operations to MapReduce functions but end up inheriting the inefficiencies of the MapReduce interface. MadLINQ provides a linear algebra platform on Dryad but with a focus on dense matrix computations [27]. PowerGraph [13] uses a vertex-centric programming model (non matrix approach) to implement data mining and graph algorithms. Unlike MadLINQ and PowerGraph, our aim is to address the issues in scaling R, a system which already has a large user community. Additionally, our techniques for handling load imbalance in sparse matrices may be applicable to MadLINQ.

[1] http://lemurproject.org/clueweb09.php

1.3 Our Contribution

We present Pronto, an R prototype to efficiently process large, sparse datasets. Pronto introduces the distributed array, `darray`, as the abstraction to process both dense and sparse datasets in parallel. Distributed arrays store data across multiple machines. Programmers can execute parallel functions that communicate with each other and share state using arrays, thus making it efficient to express complex algorithms.

Pronto programs are executed by a set of worker processes which are controlled by a master. For efficient multi-core support each worker on a server encapsulates multiple R instances that read share data. To achieve zero copying overhead, we modify R's memory allocator to directly map data from the worker into the R objects. This mapping preserves the meta-data in the object headers and ensures that the allocation is garbage collection safe.

To mitigate load imbalance, the runtime tracks the execution time and the number of elements in each array partition. In case of imbalance, the runtime dynamically merges or sub-divides array partitions between iterations and assigns them to a new task, thus varying the parallelism and load in the system.

We have implemented seven different applications in Pronto, ranging from a recommendation system to a graph centrality measure. Our experience shows that Pronto programs are easy to write and can be used to express a wide variety of complex algorithms. Compared to published results of Hadoop and Spark [33], Pronto achieves equally good execution times with only a handful of multi-core servers. For the PageRank algorithm, Pronto is more than 40× faster than Hadoop and 15× faster than Spark.

2. Background

Matrix computation is heavily used in data mining, image processing, graph analysis, and elsewhere [30]. Our focus is to analyze sparse datasets that are found as web graphs, social networks, product ratings in Amazon, and so on. Many of these analyses can be expressed using matrix formulations that are difficult to write in data parallel models such as MapReduce.

Example: graph algorithms. Many common graph algorithms can be implemented by operating on the adjacency matrix [18]. To perform breadth-first search (BFS) from a

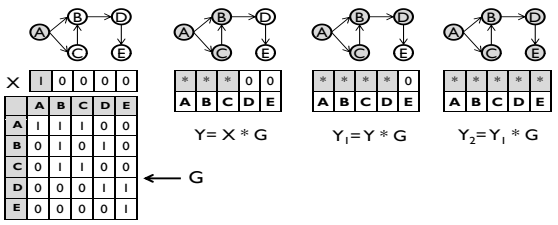


Figure 3. Breadth-first search using matrix operations. The k^{th} multiplication uncovers vertices up to k hop distance away.

vertex i we start with a $1 \times N$ vector `x` which has all zeroes except the i^{th} element. Using the multiplication `y=x*G` we extract the i^{th} row in `G`, and hence the neighbors of vertex i. Multiplying `y` with `G` gives vertices two steps away and so on. Figure 3 illustrates BFS from source vertex `A` in a five vertex graph. After each multiplication step the non-zero entries in Y_i (starred) correspond to visited vertices. If we use a sparse matrix representation for `G` and `x`, then the performance of this algorithm is similar to traditional BFS implementations on sparse graphs.

The Bellman-Ford single-source shortest path algorithm (SSSP) finds the shortest distance to all vertices from a source vertex. SSSP can be implemented by starting with a distance vector `d` and repeatedly performing a modified matrix multiplication, `d=d⊗G`. In the modified multiplication `d(j)=`$min_k\{$`d(k)+G(k,j)`$\}$ instead of the usual `d(j)=`$\sum_k\{$`d(k)*G(k,j)`$\}$. In essence, each multiplication step updates the vertex distances by choosing the minimum of the current distance, and that of reaching the vertex using one more edge.

2.1 R: An array-based environment

R provides an interactive environment to analyze data. It has interpreted conditional execution (`if`), loops (`for, while, repeat`), and uses array operators written in C, C++ and FORTRAN for better performance. Line 1 in Figure 4 shows how a 3×3 matrix can be created. The argument `dim` specifies the shape of the matrix and the sequence $10:18$ is used to fill the matrix. One can refer to entire subarrays by omitting an index along a dimension. For example, in line 3 the first row of the matrix is obtained by $A[1,]$, where the column index is left blank to fetch the entire first row. Subsections of a matrix can be easily extracted using *index vectors*. Index vectors are an ordered vector of integers. To extract the diagonal of `A` we create an index matrix `idx` in line 4 whose elements are $(1,1)$,$(2,2)$ and $(3,3)$. In line 6, `A[idx]` returns the diagonal elements of `A`. In a single machine environment, R has native support for matrix multiplication, linear equation solvers, matrix decomposition and others. For example, $\%*\%$ is an R operator for matrix multiplication (line 7).

```
1: > A<-array(10:18,dim=c(3,3))       #3x3 matrix
2: > A
        [,1] [,2] [,3]
   [1,]   10   13   16
   [2,]   11   14   17
   [3,]   12   15   18
3: > A[1,]                            #First row
   [1] 10 13 16
4: > idx<-array(1:3,dim=c(3,2))       #Index vector
5: > idx
        [,1] [,2]
   [1,]    1    1
   [2,]    2    2
   [3,]    3    3
6: > A[idx]                           #Diagonal of A
   [1] 10 14 18
7: > A%*%idx                          #Matrix multiply
        [,1] [,2]
   [1,]   84   84
   [2,]   90   90
   [3,]   96   96
```

Figure 4. Example array use in R.

3. Programming model

Pronto is R with new language extensions and a runtime to manage distributed execution. The extensions add distributed and parallel execution. The runtime takes care of memory management, scheduling, dynamic data partitioning, and fault tolerance. As shown in Figure 5, programmers write a Pronto program and submit it to a *master* process. The runtime at the master is in charge of the overall execution. It executes the program as distributed tasks across multiple *worker* processes. Table 1 depicts the Pronto language constructs which we discuss in this section.

3.1 Distributed arrays

Pronto solves the problem of structure and scalability by introducing distributed arrays. Distributed array (`darray`) provides a shared, in-memory view of multi-dimensional data stored across multiple servers. Distributed arrays have the following characteristics:

Partitioned. Distributed arrays can be partitioned into chunks of rows, columns or blocks. Users specify the size of the initial partitions. Pronto workers store partitions of the distributed array in the compressed sparse column format unless the array is defined as dense. Programmers use partitions to specify coarse grained parallelism by writing functions that execute in parallel and operate on partitions. Partitions can be referred to by the `splits` function. The `splits` function automatically fetches remote partitions and combines them to form a local array. For example, if `splits(A)` is an argument to a function executing on a worker then the whole array `A` would be reconstructed by the runtime, from local and remote partitions, and passed to that worker. The i^{th} partition can be referenced by `splits(A,i)`.

Shared. Distributed arrays can be read-shared by multiple concurrent tasks. The user simply passes the array partitions as arguments to many concurrent tasks. Arrays can be mod-

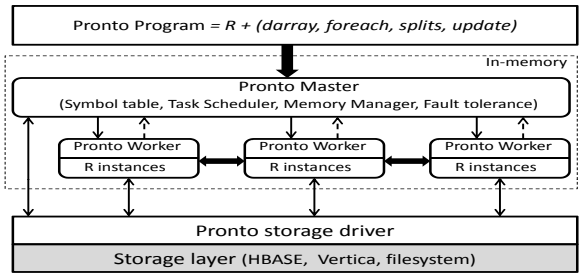


Figure 5. Pronto architecture

ified inside tasks and the changes are visible globally when `update` is called. Pronto supports only a single writer per partition.

Dynamic. Partitions of a distributed array can be loaded in parallel from data stores such as HBase, Vertica, or from files. Once loaded, arrays can be dynamically re-partitioned to reduce load imbalance and to mitigate stragglers.

3.2 Distributed parallelism

Pronto provides programmers with a `foreach` construct to execute deterministic functions in parallel. The functions do not return data. Instead, programmers call `update` inside the function to publish changes. The Pronto runtime starts tasks on worker nodes for parallel execution of the loop body. By default, there is a barrier at the end of the loop to ensure all tasks finish before statements after the loop are executed.

3.3 Repartition and invariants

At runtime programmers can use the `repartition` function to trigger Pronto's dynamic repartitioning method. Repartitioning can be used to subdivide an array into a specified number of parts. Two or more partitions can be combined using the merge flag. Repartitioning is a performance optimization which helps when there is an imbalance in the system.

One needs be careful while repartitioning structured data, otherwise program correctness may be affected. For example, when multiplying two matrices, the number of rows and columns in partitions of both the matrices should conform. If we repartition only one of the matrices then this invariant may be violated. Therefore, Pronto allows programmers to optionally specify the array invariants in the program. We show in Section 4.3 how the runtime can use the `invariant` and `repartition` functions to automatically detect and reduce imbalance without any user assistance.

Note that for programs with general data structures (e.g., trees) writing invariants is difficult. However, for matrix computation, arrays are the only data structure and the relevant invariant is the compatibility in array sizes. The `invariant` in Pronto is similar in spirit to the alignment directives used in High Performance Fortran (HPF [20]). The HPF directives align elements of multiple arrays to ensure the arrays are distributed in the same manner. Unlike HPF, in Pronto the invariants are used to maintain correctness during repartitioning.

Functionality	Description
`darray(dim=, blocks=, sparse=)`	Create a distributed array with dimensions specified by `dim`, and partitioned by blocks of size `blocks`.
`splits(A,i)`	Return i^{th} partition of the distributed array `A` or the whole array if i is not specified.
`foreach(v, A, f())`	Execute function `f` as distributed tasks for each element `v` of `A`. Implicit barrier at the end of the loop.
`update(A)`	Publish the changes to A.
`repartition(A, n=, merge=)`	Repartition `A` into `n` parts.
`invariant(A, B, type=)`	Declare compatibility between arrays `A` and `B` by rows or columns or both. Used by the runtime to maintain invariants while repartitioning.

Table 1. Main programming language constructs in Pronto

3.4 Example: PageRank

Figure 6 shows the Pronto code for PageRank. `M` is the modified adjacency matrix of the Web graph. PageRank is calculated in parallel (lines 6–13) using the power method [7]. In line 1, `M` is declared as an NxN array. `M` is loaded in parallel from an HBase table using the Pronto driver, and is partitioned by rows. In line 3 the number of columns of `M` is used to define the size of a dense vector `pgr` which acts as the initial PageRank vector. This vector is partitioned such that each partition of `pgr` has the same number of rows as the corresponding partition of `M`. The accompanying illustration points out that each partition of the vector `pgr` requires the corresponding (shaded) partitions of `M`, `Z`, and the whole array `xold`. The Pronto runtime passes these partitions and completely reconstructs `xold` from its partitions before executing `prFunc` at each worker.

Line 5 (Figure 6) is an example invariant for the PageRank code. Each of `pgr`, `M`, `xold`, and `Z` should have the same number of rows in each partition. By specifying this invariant, the programmer constrains the runtime to adhere to this compatibility between arrays even during automatic repartitioning.

4. System design

The Pronto master acts as the control thread. The workers execute the loop body in parallel whenever `foreach` loops are encountered. The master keeps a symbol table which maps variables to their physical location. This map is used by workers to exchange information using pairwise communication. In this paper we describe only the main mechanisms related to multi-core support and optimizations for sparsity and caching.

```
#Load data in parallel from adjacency matrix in HBase
1 : M<- darray(dim=c(N,N),blocks=c(s,N), sparse=T)
2 : load(M, table='Web-graph', drv='HBase')
3 : pgr<- darray(dim=c(ncol(M),1),blocks=c(s,1),sparse=F)
4 : ...
5 : invariant(pgr,M,xold,Z, type=ROW)
#Calculate PageRank (pgr)
6 :  repeat{
      #Distributed matrix operations
7 :    foreach(i, 1:numsplits(M),
             prFunc(p=splits(pgr,i), m=splits(M,i),
                    x=splits(xold), z=splits(Z,i)) {
8 :        p<-(m%*%x)+ z
9 :       update(p)
10:    })
11:    if(norm(pgr-xold)<1e-9) break
12:    xold<-pgr
13:  }
```

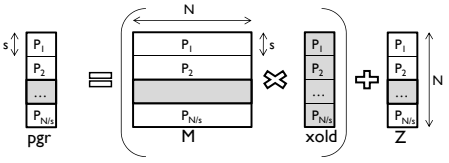


Figure 6. PageRank on a Web graph.

4.1 Versioning arrays

Pronto uses versioning to ensure correctness when arrays are updated across loop iterations. For example, write conflicts may arise if tasks read share an array which is also written to within that iteration. To avoid such conflicts, each partition of a distributed array has a version. The version of a distributed array is a concatenation of the versions of its partitions, similar in spirit to vector clocks. Updates to array partitions occur on a new version of the partition. By updating the version, concurrent readers of previous versions are guaranteed to have access to data. For example, the PageRank vector `pgr` in Figure 6 starts with version $\langle 0,0,..,0\rangle$. After the first iteration the new version is $\langle 1,1,..,1\rangle$. Pronto uses reference counting to garbage collect older versions of arrays not used by any task. Pronto workers periodically inform the master about what partitions they are still using. The master uses this information to track live references.

4.2 Efficient multi-core support

Since R is not thread safe, a simple approach to utilize multi-cores is to start multiple worker processes on the same server. There are three major drawbacks: (1) on the server multiple copies of the same array will be created, thus inhibiting scalability, (2) copying the data across processes, using pipes or network, takes time, and (3) the network communication increases as we increase the number of cores being utilized.

Instead, Pronto allows a worker to encapsulate multiple R processes that can communicate through shared memory with zero copying overhead (Figure 7). The key idea in Pronto is to efficiently initialize R objects by mapping data using mmap or shared memory constructs. However, there are some important safety challenges that need to be addressed.

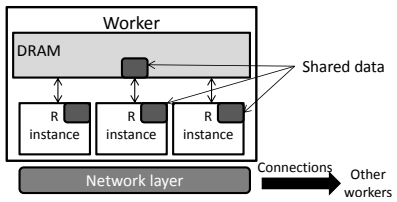


Figure 7. Multiple R instances share data hosted in a worker.

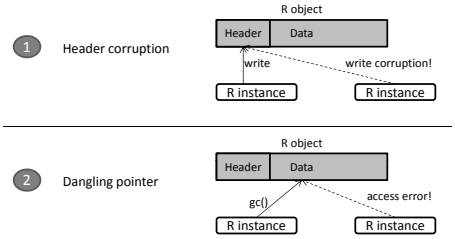


Figure 8. Simply sharing R objects can lead to errors.

Issues with data sharing. Each R object consists of a fixed-size header, and an array of data immediately following the header. The header (among other things) has information about the type and size of the corresponding data part. Simply pointing an R variable to an external data source leads to data corruption. As shown in Figure 8, if we were to share an R object across different R instances two problems can arise. First, both the instances may try to write instance specific values to the object header. This conflict will lead to header corruption. Second, R is a garbage-collected language. If one of the instances garbage collects the object then the other instance will be left with a dangling pointer.

Safe data sharing. We solve the data sharing challenge by entrusting each worker with management of data shared by multiple R processes. We only share read-only data since only one process may write to a partition during a loop iteration and writes always create a new version of a partition. Pronto first allocates process local objects in each R instance and then maps the shared data on the data part of the object. Since the headers are local to each R instance, write conflicts do not occur on the header.

There is another issue that has to be solved: the mmap call locates data only to an address at a page boundary. However, R's internal allocator does not guarantee that the data part of an object will start at a page boundary. To solve this issue, Pronto overrides the behavior of the internal allocator of R. We use `malloc_hook` to intercept R's malloc() calls. Whenever we want to allocate a shared R object we use our custom malloc to return a set of pages rounded to the nearest multiple of the page size. Once the object has been allocated the shared data can be mapped using mmap.

Figure 9 shows that R objects are allocated through the default malloc for local objects and through Pronto's malloc function for shared objects. The shared objects consist of a set of pages with the data part aligned to the page bound-

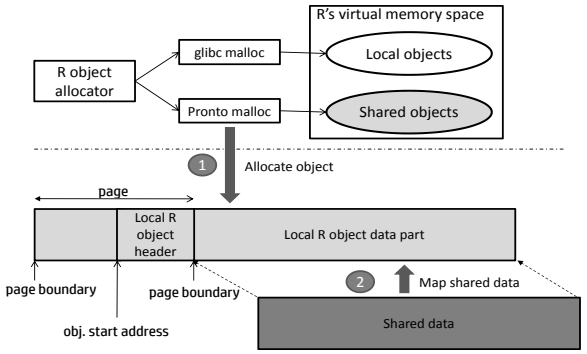


Figure 9. Shared object allocation in an R instance.

ary. The first page starts with an unused region because the header is smaller than a full page.

When the objects are no longer needed, these specially allocated regions need to be unmapped. Pronto uses `free_hook` to intercept the calls to the glibc free() function. Pronto also maintains a list of objects that were specially allocated. The list contains the starting address and allocation size of the shared objects. Whenever free is called, the runtime checks if the object to be freed is present in the list. If it is then munmap is called. Otherwise, the glibc free function is called. Note that while the malloc hook is used only when allocating shared R objects, the free hook is active throughout the lifetime of the program, because we do not know when R may garbage collect objects.

4.3 Dynamic partitioning for sparse data

While shared memory constructs help in reducing the network overhead, the overall time taken for a distributed computation also depends on the execution time. Partitioning a sparse matrix may lead to uneven distribution of nonzero elements and cause a skew in task execution times. Moreover, the number of tasks in the system is tied to the number of partitions which makes it difficult to effectively use additional workers at runtime. Pronto uses dynamic partitioning to mitigate load imbalance, and to increase or decrease the amount of parallelism in the program at runtime. One can determine optimal partitions statically to solve load imbalance but it is an expensive solution. Such partitions may not remain optimal as data is updated and static partitioning does not adjust to change in number of workers.

Pronto uses two observations to dynamically adjust partitions. First, since our target algorithms are iterative, we refine the partitions based on the execution of the first few iterations. Second, if we know the invariants for the program we can automatically re-partition data without affecting correctness.

The Pronto runtime tracks both the number of elements in a partition (e_i) and the execution time of the tasks (t_i). It uses these metrics to decide when to repartition data to reduce load imbalance. The runtime starts with an initial partitioning (generally user specified), and in subsequent iterations may either merge or sub-divide partitions to create new ones. The aim of dynamic partitioning is to keep the partition sizes and the execution time of each task close to the median [4]. The runtime tracks the median partition size (e_m) and task execution time (t_m). After each iteration, the runtime checks if a partition has more (less) elements than the median by a given constant (partition threshold e.g. $e_i/e_m \geq \delta$) and sub-divides (merges) them. In the PageRank program (Figure 6), after repartitioning the runtime simply invokes the loop function (`pgFunc`) for a different number of partitions and passes the corresponding data. No other changes are required.

For dynamic partitioning the programmer needs to specify the invariants and annotate functions as *safe* under repartitioning. For example, a function that assigns the first element of each partition is unsafe. Such a function is closely tied to each partition, and if we sub-divide an existing partition then two cells will be updated instead of one. In our applications, the only unsafe functions are related to initialization such as setting `A[i]=1` in breadth-first search.

4.4 Co-location, scheduling, and caching

Pronto workers execute functions which generally require multiple array partitions, including remote ones. Pronto uses three mechanisms to reduce communication: locality based scheduling, partition co-location, and caching.

The Pronto master schedules tasks on workers. The master uses the symbol table to calculate the amount of remote data copy required when assigning a task to a worker. It then schedules tasks to minimize data movement. Partitions that are accessed and modified in the same function can be co-located on the same worker. As matrix computations are structured, in most cases co-locating different array partitions simply requires placing the i^{th} partition of the corresponding arrays together. For example, in PageRank, the i^{th} partition of vectors `pgr`, `M`, and `Z` should be co-located. Instead of another explicit placement directive, Pronto reuses information provided by the programmer in the `invariant` function to determine which arrays are related and attempts to put the corresponding partitions on same workers. This strategy of co-location works well for our applications. In future, we plan to consider work-stealing schedulers [5, 26].

Pronto automatically caches and reuses arrays whose versions have not changed. For example, in the PageRank code `Z` is never modified. After the first iteration, workers always reuse `Z` as its version never changes. The runtime simply keeps the reference to partitions of `Z` alive and is informed by the master when a new version is available. Due to automatic caching, Pronto does not need to provide explicit directives such as broadcast variables [33].

4.5 Fault tolerance

Pronto uses primary-backup replication to withstand failures of the master node. Only the meta-data information like the

symbol table, program execution state, and worker information is replicated at the backup. The state of the master is reliably updated at the backup before a statement of the program is considered complete. R programs are generally a couple of hundred lines of code, but most lines perform a compute intensive task. The overhead of check-pointing the master state after each statement is low compared to the time spent to execute the statement.

We use existing techniques in literature for worker fault tolerance. The master sends periodic heartbeat messages to determine the progress of worker nodes. When workers fail they are restarted and the corresponding functions are re-executed. Like MapReduce and Dryad we assume that tasks are deterministic, which removes checkpointing as data can be recreated using task re-execution. The matrix computation focus of Pronto simplifies worker fault-tolerance. Arrays undergo coarse grained transformations and hence it is sufficient to just store the transformations reliably instead of the actual content of the arrays. Therefore, Pronto recursively recreates the corresponding versions of the data after a failure. The information on how to recreate the input is stored in a table which keeps track of what input data versions and functions result in specific output versions. In practice, arrays should periodically be made durable for faster recovery.

5. Implementation

Pronto is implemented as an R add-on package and provides support for the new language features described in Section 3. Dense and sparse matrices are stored using R's Matrix library. Our current prototype has native support for a limited set of distributed array operators such as load, save, matrix multiplication, addition, and so on. Other operators and algorithms can be written by programmers using functions inside `foreach`. The implementation of both Pronto master and workers use Zero MQ servers [15]. Control messages, like starting the loop body in a worker or calls to garbage collect arrays, are serialized and sent using Google's protocol buffers. Data transfers, such as copying remote arrays, occur directly via TCP connections. The Pronto package contains 800 lines of R code and 10,000 lines of C++ code.

6. Evaluation

Programmers can express various algorithms in Pronto that are otherwise difficult or inefficient to implement in current systems. Table 2 lists seven applications that we implement in Pronto. These applications span graph algorithms, matrix decomposition, and dense linear algebra. The sequential version of each of these algorithms can be written in fewer than 80 lines in R. In Pronto, the distributed versions of the same applications take at most 135 lines. Therefore, only a modest effort is required to convert these sequential algorithms to run in Pronto.

Application	Algorithm Characteristic	R LOC	Pronto LOC
PageRank	Eigenvector calculation	20	41
Vertex centrality	Graph Algorithm	40	128
Edge centrality	Graph Algorithm	48	132
SSSP	Graph Algorithm	30	62
Netflix recommender [34]	Matrix decomposition	78	130
Triangle count [17]	Top-k eigenvalues	65	121
k-Means clustering	Dense linear algebra	35	71

Input data	Size	Application
Twitter	V=41M, E=1.4B	PageRank, Centrality, SSSP
ClueWeb-S	V=100M, E=1.2B	PageRank
ClueWeb	V=2B, E=6B	PageRank

Table 2. Pronto applications and their input data.

In this paper we focus on PageRank, vertex centrality, and single-source shortest path (SSSP). We compare the performance of Pronto to Spark [33], which is a recent in-memory system for cluster computing, and Hadoop-mem, which is Hadoop-0.20 but run entirely on ramfs to avoid disk latencies. Spark performs in-memory computations, caches data, and is known to be 20× faster than Hadoop on certain applications. In all the experiments we disregard the initial time spent in loading data from disk. Subsequent references to Hadoop in our experiments refers to Hadoop-mem.

Our evaluation shows that:

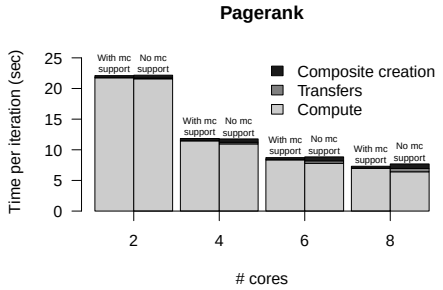
- Pronto is the first R extension to efficiently leverage multi-cores by reducing memory and network overheads.
- Pronto can handle load imbalance due to sparsity by dynamic partitioning.
- Pronto is much faster than current systems. On PageRank Pronto is 40× faster than Hadoop, 15× faster than Spark, and comparable to MPI implementations.

Our experiments use a cluster of 50 HP SL390 servers with Ubuntu 11.04. Each server has two 2.67GHz (12-core) Intel Xeon X5650 processors, 96GB of RAM, 120GB SSD, and a 10Gbps network interface. Pronto, Hadoop, and Spark are run with the same number of workers or mappers. Hadoop algorithms are part of Apache Mahout [1].

6.1 Application description

Since we have discussed PageRank and SSSP in Section 2, we briefly describe the centrality measure algorithm.

Vertex or edge betweenness centrality determines the importance of a vertex or edge in a network (e.g., social graph) based on the number of shortest paths that include the vertex or edge. We implement Brandes' algorithm for unweighted graphs [6]. Each betweenness algorithm consists of two phases: first the shortest paths from each vertex to all other vertices are determined (using BFS) and then these paths are used to update the centrality measure using scalar transformations. In our experiments we show the results of starting from a vertex whose BFS has 13 levels.



Input data	Vertices	#Cores	Additional memory used (no MC)
Twitter	41M	8	2.1G
ClueWeb-S	100M	8	5.3G

Figure 10. Multi-core (MC) support lowers total execution time and memory usage on a single server. Lower is better.

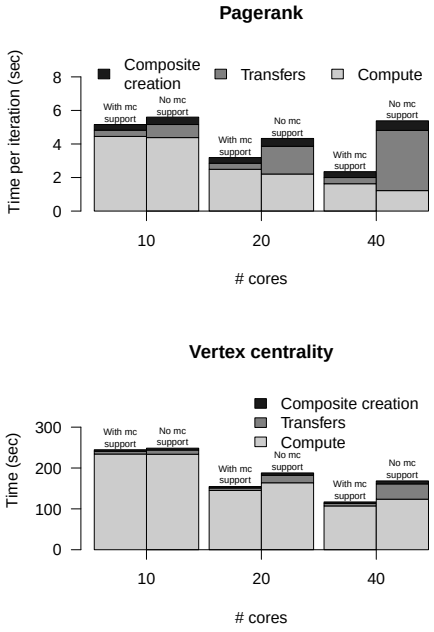


Figure 11. Multi-core support reduces communication overhead in (a) PageRank (b) Centrality. Lower is better.

6.2 Advantages of multi-core support

Pronto's multi-core support lowers the memory and communication overhead over simply using multiple (single core) worker instances. In this section we vary the number of cores utilized and show the time spent during computation, composite creation (constructing a distributed array from its partitions), and data transfer. We use Pronto-NoMC to denote the system which does not have multi-core support and has single core workers.

Single server: low memory overhead. The first advantage of multi-core support is that there is no need to copy data between two R instances that are running on the same server. Unlike other R packages, Pronto can safely share data across processes through shared memory. Figure 10 shows the average iteration time of PageRank on the 1.5B edge Twitter graph when executed on a single server. The data transferred in this algorithm is the PageRank vector. In Pronto there is no transfer overhead as all the R instances are on the same server and can share data. At 8 cores Pronto-NoMC spends 7% of the time in data transfers and takes 5% longer to complete than Pronto. The difference in execution time is not much as communication over `localhost` is very efficient even with multiple workers per server. However, the real win for multi-core support in a single server is the reduction in memory footprint. The table in Figure 10 shows that at 8 cores the redundant copies of the PageRank vector in Pronto-NoMC increase the memory footprint by 2 GB, which is 10% of the total memory usage. For the Clueweb-S dataset Pronto-NoMC uses up to 5.3 GB of extra memory.

Multiple servers: low communication overhead. The second advantage of Pronto is that in algorithms with all-to-all communication (broadcast), the amount of data transferred is proportional only to the number of servers, not the number of R instances. Figure 11 shows the significance of this improvement for experiments on the Twitter graph. In these experiments we fix the number of servers to 5 and vary the total number of cores utilized. Figure 11(a) shows that the network transfer overhead for Pronto-NoMC is $2.1\times$ to $9.7\times$ higher than Pronto as we vary the total cores utilized from 10 to 40. Worse still, at 40 cores the PageRank code on Pronto-NoMC not only stops scaling rather it takes more time to complete than with 20 cores due to higher transfer overhead. In comparison, Pronto can complete an iteration of PageRank in about 3 seconds, though there is only marginal benefit of adding more than 20 cores for this dataset. Figure 11(b) shows similar behavior for the centrality measure algorithm. Using Pronto the execution time for a single vertex decreases from 244 seconds at 10 cores to 116 seconds at 40 cores. In comparison, with no multi-core support Pronto-NoMC incurs very high transfer overhead at 40 cores and the execution time is worse by 43% and takes 168 seconds.

6.3 Advantages of dynamic partitioning

While multi-core support lowers the memory and communication overhead, dynamic repartitioning of matrices reduces imbalance due to data sparsity. We evaluate the effectiveness of dynamic partitioning for PageRank on the ClueWeb graph with 2B vertices and 6B edges. These experiments use 25 servers each with 8 R instances. Even though we use 200 cores in this experiment, we initially partition the graph into 1000 parts. This allows the scheduler to intelligently overlap computations and attempts to improve the balance. In this section we show that dynamic repartitioning improves performance even in such a case.

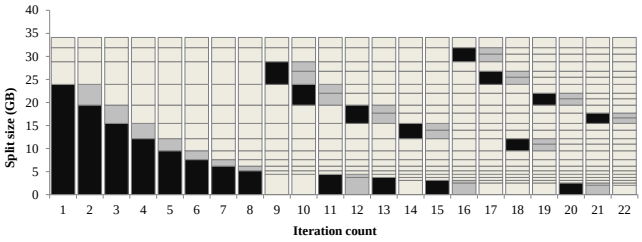


Figure 12. We trace the repartitioning seen in the initial four matrix blocks. Black boxes represents *heavy* blocks chosen for repartitioning and gray boxes indicate newly created blocks.

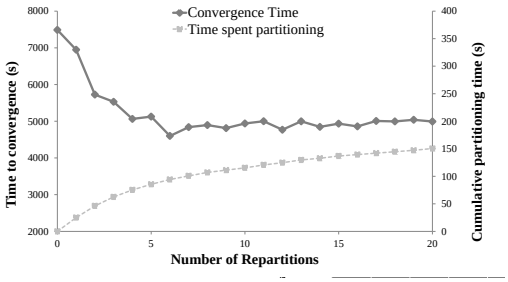


Figure 13. Convergence time decreases with repartitioning. The cumulative partitioning time is the time spent in repartitioning.

Effects of repeated partitioning Figure 12 shows how the repartitioning algorithm proceeds on the ClueWeb dataset. A black colored partition indicates that the particular block was *heavy* and chosen for repartitioning. The newly created array partitions are shown in gray. In Figure 12 the first block (also the densest) is continuously repartitioned from iteration 1 to iteration 7 and then again at iterations 11, 13, 15, and 20. Overall, repartitioning reduces the size of this partition from 23GB to 2.2GB.

However there is a cost associated with repartitioning as the PageRank iterations have to be paused while the graph is being repartitioned. To quantify the cost-benefit trade-off, we estimate the total running time of PageRank as we increase the number of repartitions. Figure 13 shows the estimated running time for 50 iterations (till convergence) of PageRank. We calculate the total execution time after a certain number of re-partitions by assuming no more repartitions will occur. For example, at x-axis value of 5, Pronto has performed five repartitions and the convergence time is 5,126 seconds if no further repartitions occur. The convergence time reduces by 32% (40 minutes) after the first four repartitions, but the benefits diminish beyond that. Note that the cumulative time spent in partitioning is a small fraction of the total execution time (between 0.3% and 3%).

Benefits of reducing imbalance Reducing the imbalance among partitions helps decrease the PageRank iteration time. Figure 14 shows the time taken by each worker during one iteration of PageRank. The horizontal bars depict

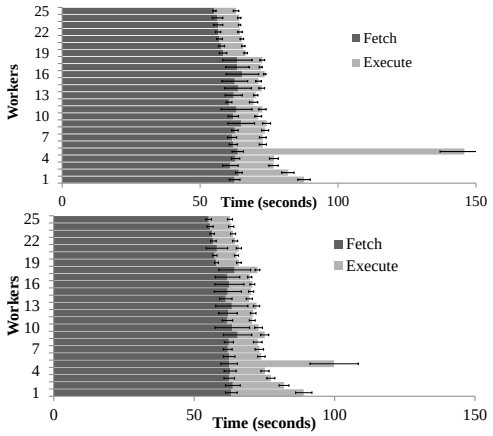


Figure 14. Per worker execution time for PageRank (a) before repartitioning (b) after four repartitions. Shorter bar is better.

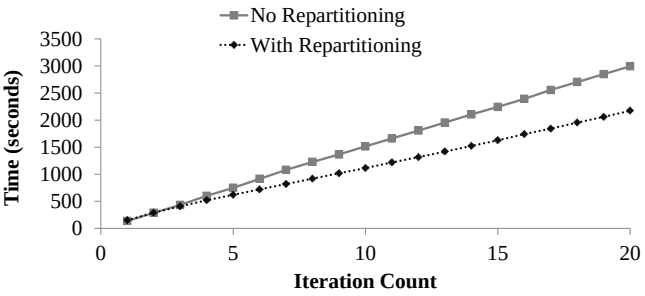


Figure 15. Comparison of overall execution time with and without repartitioning. Lower is better.

what part of the total time was spent in transferring data versus the time taken to perform the computation. Since there is a barrier at the end of an iteration, the iteration time is determined by the maximum execution time among the workers. Figure 14(a) shows that the slowest worker takes 147 seconds initially but after four repartitions (Figure 14(b)) it finishes in 95 seconds thus reducing the per-iteration time.

Reducing imbalance is especially important for iterative algorithms as the overall execution time can be significantly high due to the skew among workers. As seen in Figure 15, re-partitioning reduces the completion time by around 822 seconds (13.7 minutes) when the PageRank algorithm is run for 20 iterations.

6.4 Scalability

Figure 16 uses SSSP on the 1.5B edge Twitter dataset to show the performance scaling of Pronto. The experiments were done using 8 cores per server. While Pronto can scale to hundreds of cores, and the execution time continues to decrease, the scaling factor is less than the ideal. For example, when increasing the cores from 16 to 128 (8×), the execution time drops from 125 seconds to 41 seconds (3×). The less than ideal scaling is a result of the communication over-

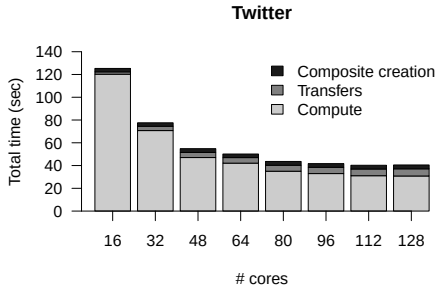


Figure 16. SSSP scalability on the Twitter dataset.

head involved in SSSP, which is proportional to the number of vertices in the graph. In future we plan to rewrite the SSSP algorithm to use block partitions of the matrix (instead of row partitions) so that no single R instance requires the full shortest path vector.

6.5 Comparison with MPI, Spark, and Hadoop

PageRank experiments on the 1.2B edge ClueWeb-S graph shows that Pronto is more than 40× faster than Hadoop, more than 15× faster than Spark, and can outperform simple MPI implementations.

MPI. We implemented PageRank using sparse matrix and vector multiplication in MPI. The communication phase in the code uses `MPI_Allgather` to gather the partitions of the PageRank vector from processes and distribute it to all. Figure 17(a) shows that Pronto outperforms the MPI code sometimes by 2×. There are two reasons for this performance difference. First, the MPI code does not handle compute imbalance. For example, at 64 cores one MPI process finishes in just 0.6 seconds while another process takes 4.4 seconds. Since processes wait for each other before the next iteration, the compute time is determined by the slowest process. Second, while MPI's network overhead is very low at 8 processes, it increases with the increase in the number of cores. However, for Pronto the network overhead is proportional to the number of multi-core servers used, and hence does not increase at the same rate. With more effort one can implement multi-threaded programs executing at each MPI process. Such an implementation will reduce the network overhead but not the compute imbalance.

Spark. We use Spark's PageRank implementation [33] to compare its performance with Pronto. Spark takes about 64.185 seconds per-iteration with 64 cores. The per-iteration time includes a map phase which computes the rank of vertices and then propagates them to reducers that sum the values. We found that the first phase was mostly compute intensive and took around 44.3 seconds while the second phase involved shuffling data across the network and took 19.77 seconds. At fewer cores, the compute time is as high as 267.26 seconds with 8 cores. The main reason why Spark is at least 15× slower than Pronto is because it generates a large amount of intermediate data and hence spends more time than Pronto during execution and network transfers. Note that the Y-axis in the plot is log scale.

Hadoop. In Mahout, an iteration of PageRank takes 161 seconds with 64 mappers (Figure 17(b)). In comparison each iteration of PageRank in Pronto takes less than 4 seconds. A portion of the 40× performance difference is due to the use of Java. However unlike Pronto, MapReduce has the additional overhead of the sort phase and the time spent in deserialization. Pronto preserves the matrix structure in between operations, and also eliminates the need to sort data between iterations.

Existing R packages. Unfortunately, current parallel R packages only allow side-effect free functions to be executed in parallel. It means that R objects in workers are deleted across iterations. Thus, to run more than one iteration of PageRank the whole graph needs to be reloaded in the next iteration making the measurements flawed. Instead, we ran a microbenchmark with 8 cores where only a dense vector of 100M entries was exchanged after each round (similar to the PageRank vector). By efficiently using mull-cores and worker-worker communication Pronto is more than 4× faster than the R parallel package (called `doMC`).

7. Discussion

Pronto makes it easy for users to algorithmically explore large datasets. It is a step towards a platform on which high level libraries can be implemented. We believe that Pronto packages that implement scalable machine learning and graph algorithms will help the large R user base reap the benefits of distributed computing.

However, certain challenges remain both in the current prototype and in the applicability of R to all problems. First, the current prototype is limited by main memory: datasets need to fit the aggregate memory of the cluster. While most pre-processed graphs are in the low terabyte size range, for larger datasets it may be economical to use an out-of-core system. Second, Pronto assumes that there is a single writer per partition. This is reasonable for matrix computations but is otherwise restrictive and the prototype does not detect incorrect programs where multiple writers update the same partition.

When applied to different datasets, array-based programming may require additional pre-processing. For example, Pronto is based on R and is very efficient at processing arrays. However, graphs may have attributes attached to each vertex. An algorithm which uses these attributes (e.g., search shortest path with attribute pattern) may incur the additional overhead of referencing attributes stored in R vectors separate from the adjacency matrix. In general, real world data is semi-structured and pre-processing may be required to extract relevant fields and convert them into arrays. Unlike

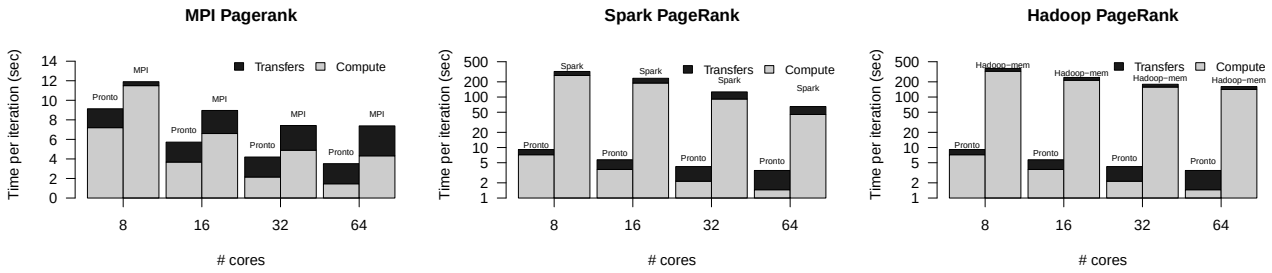


Figure 17. Performance advantage over (a) MPI (b) Spark and (c) Hadoop. Lower is better.

the Hadoop ecosystem which has both storage (HDFS) and computation (MapReduce), Pronto only has a efficient computation layer. In our experience, it's easier to load data into Pronto if the underlying store has tables (databases, HBase, etc.) and supports extraction mechanisms (e.g., SQL).

8. Related Work

Dataflow models. MapReduce and Dryad are popular dataflow systems for parallel data processing [11, 16]. To increase programmer productivity high-level programming models–DryadLINQ [32] and Pig [25]—are used on top of MapReduce and Dryad. These systems scale to hundreds of machines. However, they are best suited for batch processing, and because of their restrictive programming and communication interface make it difficult to implement matrix operations. Recent improvements, such as HaLoop [8], Twister [12], and Spark [33], do not change the programming model but improve iterative performance by caching data or using lineage for efficient fault tolerance. CIEL increases the expressibility of programs by allowing new data-dependent tasks during job execution [24]. However, none of these systems can efficiently express matrix operations.

Piccolo runs parallel applications that can share state using distributed, in-memory, key-value tables [26]. Compared to MapReduce, Piccolo is better suited for expressing matrix operations. However, Piccolo's key-value interface optimizes for low level reads and writes to keys instead of structured vector processing. Unlike Presto, Piccolo does not handle sparse datasets and the resulting load imbalance.

Pregel and GraphLab use bulk synchronous processing (BSP [31]) to execute parallel programs [21, 22]. With BSP, each vertex processes its local data and communicates with other vertices using messages. Both systems require an application to be (re)written in the BSP model. Pronto shows that the widely used R system can be extended to give similar performance without requiring any programming model changes. Pronto's execution time of PageRank on the Twitter graph (Figure 10, 8 cores, 7.3s) compares favorably to published results of PowerGraph (512 cores, 3.6s) [13].

Matrix computations. Ricardo [10] and HAMA [28] use MapReduce to implement matrix operations. While they solve the problem of scalability on large-scale data, the implementation is inefficient due to the restrictive MapReduce interface. In light of this observation, MadLINQ provides a platform on Dryad specifically for dense matrix computations [27]. Similar to Pronto, MadLINQ can reuse existing matrix libraries on local partitions, is fault tolerant and distributed. MadLINQ identifies the need to efficiently handle sparse datasets but, unlike Pronto, does not solve the problem, or support dynamic partitioning.

Popular high-performance computing (HPC) systems like ScaLAPACK do not support general sparse matrices. The few systems that do support sparse matrices (SLEPc [14], ARPACK [19]) typically provide only eigensolvers. To write a new algorithm, such as the betweenness centrality, one would have to implement it with their low level interfaces including FORTRAN code. None of these systems have load balancing techniques or fault tolerance.

MATLAB's parallel computing toolbox and existing efforts in parallelizing R can run single programs on multiple data. Unlike these systems, Pronto can safely share data across multiple processes, has fewer redundant copies of data, and can mitigate load imbalance due to sparse datasets.

Parallel languages. HPC applications use explicit message passing models like MPI. MPI programmers have the flexibility to optimize the messaging layer but are difficult to write and maintain. New parallel programming languages like X10 [9] and Fortress [29] use the partitioned global address space model (PGAS). These languages are not optimized for matrix operations and the programmer has to deal with low level primitives like synchronization and explicit locations. For example, in X10 programmers specify on what processors computations should occur using *Place*. None of these languages are as popular as R, and users will have to rewrite hundreds of statistical algorithms that are already present in R.

9. Conclusion

Pronto advocates the use of sparse matrix operations to simplify the implementation of machine learning and graph algorithms in a cluster. Pronto uses distributed arrays for structured processing, efficiently uses multi-cores, and dynamically partitions data to reduce load imbalance. Our experience shows that Pronto is a flexible computation model that can be used to implement a variety of complex algorithms.

References

[1] Apache mahout. http://mahout.apache.org.

[2] The R project for statistical computing. http://www.r-project.org.

[3] Stanford network analysis package. http://snap.stanford.edu/snap.

[4] G. Ananthanarayanan, S. Kandula, A. Greenberg, I. Stoica, Y. Lu, B. Saha, and E. Harris. Reining in the outliers in map-reduce clusters using Mantri. In *In OSDI'10*, Vancouver, BC, Canada, 2010.

[5] R. D. Blumofe and C. E. Leiserson. Scheduling multithreaded computations by work stealing. In *Proc. of the 35th Annual Symposium on Foundations of Computer Science*, SFCS '94, pages 356–368, Washington, DC, USA, 1994.

[6] U. Brandes. A faster algorithm for betweenness centrality. *Journal of Mathematical Sociology*, 25:163–177, 2001.

[7] S. Brin and L. Page. The anatomy of a large-scale hypertextual Web search engine. In *WWW7*, pages 107–117, 1998.

[8] Y. Bu, B. Howe, M. Balazinska, and M. e. D. Ernst. HaLoop: Efficient iterative data processing on large clusters. *Proc. VLDB Endow.*, 3:285–296, September 2010.

[9] P. Charles, C. Grothoff, V. Saraswat, C. Donawa, A. Kielstra, K. Ebcioglu, C. von Praun, and V. Sarkar. X10: An object-oriented approach to non-uniform cluster computing. In *OOPSLA'05*, pages 519–538, 2005.

[10] S. Das, Y. Sismanis, K. S. Beyer, R. Gemulla, P. J. Haas, and J. McPherson. Ricardo: Integrating R and Hadoop. In *SIGMOD Conference'10*, pages 987–998, 2010.

[11] J. Dean and S. Ghemawat. MapReduce: Simplified data processing on large clusters. *Commun. ACM*, 51(1), 2008.

[12] J. Ekanayake, H. Li, B. Zhang, T. Gunarathne, S.-H. Bae, J. Qiu, and G. Fox. Twister: A runtime for iterative MapReduce. In *HPDC '10*, pages 810–818, 2010.

[13] J. E. Gonzalez, Y. Low, H. Gu, D. Bickson, and C. Guestrin. PowerGraph: Distributed Graph-Parallel Computation on Natural Graphs. In *OSDI'12*, Hollywood, CA, October 2012.

[14] V. Hernandez, J. E. Roman, and V. Vidal. Slepc: A scalable and flexible toolkit for the solution of eigenvalue problems. *ACM Trans. Math. Softw.*, 31(3):351–362, Sept. 2005.

[15] P. Hintjens. ZeroMQ: The Guide, 2010.

[16] M. Isard, M. Budiu, Y. Yu, A. Birrell, and D. Fetterly. Dryad: Distributed data-parallel programs from sequential building blocks. In *EuroSys '07*, pages 59–72, 2007.

[17] U. Kang, B. Meeder, and C. Faloutsos. Spectral Analysis for Billion-Scale Graphs: Discoveries and Implementation. In *PAKDD (2)*, pages 13–25, 2011.

[18] J. Kepner and J. Gilbert. *Graph Algorithms in the Language of Linear Algebra*. Fundamentals of Algorithms. SIAM, 2011.

[19] R. B. Lehoucq, D. C. Sorensen, and C. Yang. *ARPACK users' guide - solution of large-scale eigenvalue problems with implicitly restarted Arnoldi methods.* Software, environments, tools. SIAM, 1998.

[20] D. Loveman. High performance Fortran. *IEEE Parallel & Distributed Technology: Systems & Applications*, 1(1):25–42, 1993.

[21] Y. Low, J. Gonzalez, A. Kyrola, D. Bickson, C. Guestrin, and J. M. Hellerstein. GraphLab: A New Framework for Parallel Machine Learning. *CoRR*, pages 1–1, 2010.

[22] 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 *SIGMOD '10*, pages 135–146, 2010.

[23] Q. E. McCallum and S. Weston. *Parallel R*. O'Reilly Media, Oct. 2011.

[24] D. G. Murray and S. Hand. CIEL: A universal execution engine for distributed data-flow computing. In *NSDI '11*, Boston, MA, USA, 2011.

[25] C. Olston, B. Reed, U. Srivastava, R. Kumar, and A. Tomkins. Pig latin: A not-so-foreign language for data processing. In *SIGMOD'08*, pages 1099–1110, 2008.

[26] R. Power and J. Li. Piccolo: Building fast, distributed programs with partitioned tables. In *OSDI '10*, Vancouver, BC, Canada, 2010. USENIX Association.

[27] Z. Qian, X. Chen, N. Kang, M. Chen, Y. Yu, T. Moscibroda, and Z. Zhang. MadLINQ: large-scale distributed matrix computation for the cloud. In *EuroSys '12*, pages 197–210, 2012.

[28] S. Seo, E. J. Yoon, J. Kim, S. Jin, J.-S. Kim, and S. Maeng. Hama: An efficient matrix computation with the mapreduce framework. In *In CLOUDCOM'10*, pages 721–726, Washington, DC, USA, 2010. IEEE Computer Society.

[29] G. L. Steele, Jr. Parallel programming and code selection in fortress. In *PPoPP '06*, pages 1–1, 2006.

[30] G. Strang. *Introduction to Linear Algebra, Third Edition*. Wellesley Cambridge Pr, Mar. 2003.

[31] L. G. Valiant. A bridging model for parallel computation. *Commun. ACM*, 33:103–111, August 1990.

[32] Y. Yu, M. Isard, D. Fetterly, M. Budiu, U. Erlingsson, P. K. Gunda, and J. Currey. DryadLINQ: A system for general-purpose distributed data-parallel computing using a high-level language. In *OSDI '08*, pages 1–14, 2008.

[33] M. Zaharia, M. Chowdhury, T. Das, A. Dave, J. Ma, M. McCauley, M. J. Franklin, S. Shenker, and I. Stoica. Resilient distributed datasets: a fault-tolerant abstraction for in-memory cluster computing. In *NSDI'12*, San Jose, CA, 2012.

[34] Y. Zhou, D. Wilkinson, R. Schreiber, and R. Pan. Large-Scale Parallel Collaborative Filtering for the Netflix Prize. In *AAIM '08*, pages 337–348, Shanghai, China, 2008.