MapReduce in MPI for Large-scale Graph Algorithms

Steven J. Plimpton and Karen D. Devine
Sandia National Laboratories
Albuquerque, NM
sjplimp@sandia.gov

Keywords: MapReduce, message-passing, MPI, graph algorithms, R-MAT matrices

Abstract

We describe a parallel library written with message-passing (MPI) calls that allows algorithms to be expressed in the MapReduce paradigm. This means the calling program does not need to include explicit parallel code, but instead provides "map" and "reduce" functions that operate independently on elements of a data set distributed across processors. The library performs needed data movement between processors. We describe how typical MapReduce funtionality can be implemented in an MPI context, and also in an out-of-core manner for data sets that do not fit within the aggregate memory of a parallel machine. Our motivation for creating this library was to enable graph algorithms to be written as MapReduce operations, allowing processing of terabyte-scale data sets. We outline MapReduce versions of several such algorithms: vertex ranking via PageRank, triangle finding, connected component identification, Luby's algorithm for maximally independent sets, and single-source shortest-path calculation. To test the algorithms on arbitrarily large artificial graphs we generate randomized R-MAT matrices in parallel; a MapReduce version of this operation is also described. Performance and scalability results for the various algorithms are presented for varying size graphs on a distributed-memory cluster. For some cases, we compare the results with non-MapReduce algorithms, different machines, and different MapReduce software, namely Hadoop. Our open-source library is written in C++, is callable from C++, C, Fortran, or scripting languages such as Python, and can run on any parallel platform that supports MPI.

1 Introduction

MapReduce is the programming paradigm popularized by Google researchers Dean and Ghemawat [9]. Their motivation was to enable rapid development and deployment of analysis programs to operate on massive data sets residing on Google's large distributed clusters. They introduced a novel way of thinking about certain kinds of large-scale computations as a "map" operation followed by a "reduce" operation. The power of the paradigm is that when cast in this way, a nominally serial algorithm now becomes two highly parallel operations working on data local to each processor, sandwiched around an intermediate data-shuffling operation that requires inter-processor communication. The user need only write serial code for the application-specific map and reduce functions; the parallel data shuffle can be encapsulated in a library since its operation is independent of the application.

The Google implementation of MapReduce is a C++ library with communication between networked machines via remote procedure calls. It allows for fault tolerance when large numbers of machines are used, and can use disks as out-of-core memory to process petabyte-scale data sets. Tens of thousands of MapReduce programs have since been written by Google researchers and are a significant part of the daily compute tasks run by the company [10].

Similarly, the open-source Hadoop implementation of MapReduce [1], has become widely popular in the past few years for parallel analysis of large-scale data sets at Yahoo and other data-centric companies, as well as in university and laboratory research groups, due to its free availability. MapReduce programs in Hadoop are typically written in Java, although it also supports use of stand-alone map and reduce kernels, which can be written as shell scripts or in other languages.

More recently, MapReduce formulations of traditional number-crunching kinds of scientific computational tasks have been described, such as post-processing analysis of simulation data [20], graph algorithmics [8], and linear algebra operations [11]. The paper by Tu et al. [20] was particularly insightful to us, because it described how MapReduce could be implemented on top of the ubiquitous distributed-memory message-passing interface (MPI), and how the intermediate data-shuffle operation is conceptually identical to the familiar MPI_Alltoall operation. Their implementation of MapReduce was within a Python wrapper to simplify the writing of user programs. The paper motivated us to develop our own C++ library built on top of MPI for use in graph analytics, which we initially released as open-source software in mid-2009. We have since worked to optimize several of the library's underlying algorithms and to enable its operation in out-of-core mode for larger data sets. These algorithmic improvements are described in this paper and are part of the current downloadable version; see Section 7 for details.

The MapReduce-MPI (MR-MPI) library described in this paper is a simple, lightweight implementation of basic MapReduce functionality, with the following features and limitations:

- *C++ library using MPI for inter-processor communication:* The user writes a (typically) simple main program which runs on each processor of a parallel machine, making calls to the MR-MPI library. For map and reduce operations, the library calls back to user-provided *map()* and *reduce()* functions. The use of C++ allows precise control over the memory and format of data allocated by each processor during a MapReduce. Library calls for performing a map, reduce, or data shuffle, are synchronous, meaning all the processors participate and finish the operation before proceeding. Similarly, the use of MPI within the library is the traditional mode of MPI_Send and MPI_Recv calls between pairs of processors using large aggregated messages to improve bandwidth performance and reduce latency costs. A recent paper [15] also outlines the MapReduce formalism from an MPI perspective, although they advocate a more asynchronous approach, using one-way communication of small messages.

- *Small, portable:* The entire MR-MPI library is a few thousand lines of standard C++ code. For parallel operation, the program is linked with MPI, a standard message passing library available on all distributed-memory machines and many shared-memory parallel machines. For serial operation, a dummy MPI library (provided) can be substituted. As a library, MR-MPI can be embedded in other codes [2] to enable them to perform MapReduce operations.

- *In-core or out-of-core operation:* Each MapReduce object that a processor creates allocates "pages" of memory, where the page size is determined by the user. MapReduce operations can be performed using a few such pages (per processor). If the data set fits in a single page (per processor), the library performs its operations in-core. If the data set exceeds the page size, processors each write to temporary disk files (on local disks or a parallel file system) as needed and subsequently read from them. This out-of-core mode of operation allows processing of data sets larger than the aggregate memory of all the processors, i.e. up to the aggregate available disk space.

- *Flexible programmability:* An advantage of writing a MapReduce program on top of MPI, is that the user program can also invoke MPI calls directly, if desired. For example, one-line calls to MPI_Allreduce are often useful in determining the status of an iterative graph algorithm, as described in Section 4. The library interface also provides a user data pointer as an argument passed to all callback functions, so it is easy for the user program to store "state" on each processor, accessible during the map and reduce operations. For example, various flags can be stored that alter the operation of a map or reduce operation, as can richer data structures that accumulate results.

- *C++, C, and Python interfaces:* The C++ interface to the MR-MPI library allows user programs to instantiate and then invoke methods in one or more MapReduce objects. The C interface allows the same operations to be called from C or other high-level languages such as Fortran. The C interface also allows the library to be easily wrapped by Python via the Python "ctypes" module. The library can then be called from a Python script, allowing the user to write *map()* and *reduce()* callback functions in Python. If a machine supports running Python in parallel, a parallel MapReduce can also be run in this mode.

- *No fault tolerance or data redundancy:* Current MPI implementations do not enable easy detection of a dead processor or retrieval of the data it was working on. So like most MPI programs, a parallel program calling the MR-MPI library will hang or crash if a processor goes away. Unlike Hadoop, and its file system (HDFS) which provides data redundancy, the MR-MPI library reads and writes simple, flat files. It can use local per-processor disks, or a parallel file system, if available, but these typically provide no data redundancy.

The remainder of the paper is organized as follows. Sections 2 and 3 describe how in-core and out-of-core MapReduce primitives are formulated as MPI-based operations in the MR-MPI library. Section 4 briefly describes the formulation of several common graph algorithms as MapReduce operations. Section 5 gives performance results for these algorithms running on a parallel cluster for graphs ranging in size from 8 million to 2 billion edges; we highlight the performance and complexity trade-offs of a MapReduce approach versus other more special-purpose algorithms. The latter generally perform better but are harder to implement efficiently on distributed memory machines, due to the required explicit management of parallelism, particularly for large out-of-core data sets. Section 6 summarizes some lessons learned from the implementation and use of our library.

2 MapReduce in MPI

The basic datums stored and operated on by any MapReduce framweork are key/value (KV) pairs. In the MR-MPI library, individual keys or values can be of any data type or length, or combinations of multiple types (one integer, a string of characters, two integers and a double, etc); they are simply treated as byte strings by the library. A KV pair always has a key; its value may be NULL. KV pairs are stored within a MapReduce (MR) object. A user program may create one or more MR objects to implement an algorithm. Various MapReduce operations (map, reduce, etc) are invoked on an object and its KV pairs; KV pairs can also be passed between and combined across objects.

A related data type in the MR-MPI library is the key/multivalue (KMV) pair, where all values associated with the same key are collected and stored contiguously as a multivalue, which is just a longer byte string with an associated vector of lengths, one integer length per value. In this section, we assume the KV and KVM datums operated on by each processor fit in local memory.

A typical MR-MPI program makes at least three calls to the MR-MPI library, to perform *map()*, *collate()*, and *reduce()* operations on a MR object it creates. In a *map()*, zero or more key/value pairs are generated by each processor. Often this is done using data read from files, but a *map()* may generate data itself or process existing KV pairs to create new ones. The KV pairs produced are stored locally by each processor; a *map()* thus requires no inter-processor communication. Users call the library with a count of tasks to perform and a pointer to a user function; the MR-MPI *map()* operation invokes the user function multiple times as a callback. Depending on which variant of *map()* is called, the user function may be passed a file name, a chunk of bytes from a large file, a task ID, or a KV pair. Options for assigning map tasks to processors are specified by the user and include assigning consecutive chunks of tasks to processors, striding the tasks across processors, or using a master-slave model that is useful when tasks have widely varying workloads.

The *collate()* operation (or data shuffle in Hadoop) identifies unique keys and collects all the values associated with those keys to create KMV pairs. This is done in two stages, the first of which requires communication, since KV pairs with the same key may be owned by any processor. Each processor hashes each of its keys to determine which processor will "own" it. The k-byte length key is hashed into a 32-bit value whose remainder modulo P processors is the owning processor rank. Alternatively, the user can provide a hash function which converts the key into a processor rank. Each processor then sends each of its KV pairs to the owning processor.

After receiving new KV pairs, the second stage is an on-processor computation, requiring no further communication. Each processor reorganizes its KV pairs into KMV pairs, one for each unique key it owns. This is done using a hash table, rather than a sort. A sort scales as $N \log_2(N)$, i.e. it requires $\log_2(N)$ passes through the N KV pairs. With hashing, the list of KMV pairs can be created in two passes. The first pass populates the hash table with needed count and length information; the second pass copies the key and value datums into the appropriate location in a new KMV data structure. Since the cost to lookup a key in a well-formed hash table is a constant-time $O(1)$ operation, the cost of the data reorganization is also $O(N)$. This methodology has the added benefit that the number of values in each KMV pair is known and can be passed to the user function during a reduce. For some reduce operations, this count is all the information a reduce requires; the values need not be looped over. The count is not available in Hadoop-style data shuffles, which sort the values; counting the values associated with the same key requires iterating over the values.

Note that the first portion of the *collate()* operation involves all-to-all communication since each processor exchanges KV pairs with every other processor. The communication can either be done via a MPI_Alltoall() library call, or by a custom routine that aggregates datums into large messages and invokes point-to-point MPI_Send() and MPI_IRecv() calls.

The *reduce()* operation processes KMV pairs and can produce new KV pairs for continued com-

putation. Each processor operates only on the KMV pairs it owns; no communication is required. As with the *map()*, users call the library with a pointer to a user function. The MR-MPI *reduce()* operation invokes the user function, once for each KMV pair.

Several related MapReduce operations are provided by the library. For example, the *collate()* function described above calls two other functions: *aggregate()*, which performs the all-to-all communication, and *convert()*, which reorganizes a set of KV pairs into KMV pairs. Both functions can be called directly. The *compress()* function allows on-processor operations to be performed; it is equivalent to a *convert()* with on-processor KVs as input, followed by a *reduce()*. The *clone()* function turns a list of KV pairs into KMV pairs, with one value per key. The *collapse()* function turns N KV pairs into one KMV pair, with the keys and values of the KV pairs becoming $2N$ values of a single multivalue assigned to a new key. The *gather()* function collects KV pairs from all processors to a subset of processors; it is useful for doing output from one or a few processors. Library calls for sorting datums by key or value or for sorting the values within each multivalue are also provided. These routines invoke the C-library *quicksort()* function to compute the sorted ordering of the KV pairs (or values within a multivalue), using a user-provided comparison function. The KV pairs (or values in a multivalue) are then copied into a new data structure in sorted order.

The interface to various low-level operations is provided so that a user's program can string them together in various ways to construct usefule MapReduce algorithms. For example, output from a *reduce()* can serve as input to a subsequent *map()* or *collate()*. Likewise, data can be partitioned across multiple MapReduce (MR) objects, each with their own set of KV pairs. For example, as will be illustrated in Section 4, one MR object can store the edges of a graph, and another its vertices. KV pairs from multiple MR objects can be combined to perform new sequences of *map()*, *collate()*, and *reduce()* operations.

The above discussion assumed that the KV or KMV pairs stored by a processor fit in its physical memory. In this case, the MR-MPI library performs only "in-core" processing and no disk files are written or read by any of the processors, aside from initial input data (if it exists) or final output data (if it is generated). Note that the aggregate physical memory of large parallel machines can be multiple terabytes, which allows for large data sets to be processed in-core, assuming the KV and KMV pairs remain evenly distributed across processors throughout the sequence of MapReduce operations. The use of hashing to assign keys to processors typically provides for such load-balancing. In the next section, we discuss what happens when data sets do not fit in available memory.

3 Out-of-core Issues

Since the MapReduce paradigm was designed to enable processing of extremely large data sets, the MR-MPI library also allows for "out-of-core" processing, which is triggered when the KV or KMV pairs owned by one or more processors do not fit in local memory. When this occurs, a processor writes one or more temporary files to disk, containing KV or KMV pairs, and reads them back in when required. Depending on the parallel machine's hardware configuration, these files can reside on disks local to each processor, on the front end (typically a NFS-mounted file system for a parallel machine), or on a parallel file system and its associated disk array.

When a user program creates a MapReduce object, a "pagesize" can be specified, which defaults to 64 Mbytes. As described below, each MR-MPI operation is constrained to use no more than a handful of these pages. The *pagesize* setting can be as small as 1 Mbyte or as large as desired, though the user should ensure the allocated pages fit in physical memory; otherwise a processor may allocate slow virtual memory. The *pagesize* is also the typical size of individual reads and writes to the temporary disk files; hence a reasonable *pagesize* ensures good I/O performance.

We now explain how the MapReduce operations described in the previous section work in out-of-core mode. The *map()* and *reduce()* operations are relatively simple. As a *map()* generates KV pairs via the user callback function, a page of memory fills up, one KV pair at a time. When the page is full, it is written to disk. If the source of data for the *map()* operation is an existing set of KV pairs, those datums are read, one page at a time, and a pointer to each KV pair is given to the user function. Similarly, a *reduce()* reads one page of KMV pairs from disk, and passes a pointer to each pair, one at a time, to the user callback function that typically generates new KV pairs. The generated pairs fill up a new page, which is written to disk when full, just as with the *map()* operation. Thus for both a *map()* and *reduce()*, out-of-core disk files are read and written sequentially, one page at a time, which requires at most two pages of memory.

A special case is when a single KMV pair is larger than a single page. This can happen, for example, in a connected component finding algorithm if the graph collapses into one or a few giant components. In this case, the set of values (graph vertices and edges) associated with a unique key (the component ID), may not fit in one page of memory. The individual values in the multivalue are then spread across as many pages as needed. The user function that processes the KMV pair is passed a flag indicating it received only a portion of the values. Once it has processed them, it can request a new set of values from the MR-MPI library, which reads in a new page from disk.

Performing an our-of-core *collate()* operation is more complex. Recall that the operation occurs in two stages. First, keys are hashed to "owning" processors and the KV pairs are communicated to new processors in an all-to-all fashion. In out-of-core mode, this operation is performed on one page of KV data (per processor) at a time. Each processor reads in a page of KV pairs, the communication pattern is determined (which processors receive which datums in that page), and all-to-all communication is performed. Each processor allocates a two-page chunk of memory to receive incoming KV pairs. On average each processor should receive one page of KV pairs; the two-page allocation allows for some load imbalance. If the KV pairs are distributed unevenly so that this limit is exceeded, the all-to-all communication is performed in smaller, multiple passes until the full page contributed by every processor has been communicated. Performing an MPI_Alltoall(), or using the custom all-to-all routines provided by the MR-MPI library, requires allocation of auxiliary arrays that store processor indices, datum pointers, and datum lengths. For the case of many tiny KV pairs, the total number of memory pages required to perform the all-to-all communication, including the source and target KV pages is at most seven.

When the communication stage of the *collate()* operation is complete, each processor has a set of KV pages, stored in an out-of-core KV file, that need to be reorganized into a set of KMV pages, likewise stored in a new out-of-core KMV file. In principle, creating one page of KMV pairs would require all KV pages be scanned to find all the keys that contribute values to that page. Doing this for each output page of KMV pairs could be prohibitively expensive. A related issue is that, as described in the previous section, a hash table is needed to match new keys with previously encountered keys. But there is no guarantee that the hash table itself will not grow arbitrarily large for data sets with many unique keys. We need an algorithm for generating KMV pairs that operates within a small number of memory pages and performs a minimal number of passes through the KV disk file. An algorithm that meets these goals is diagrammed in Figure 1; it reads the KV pages at most four times, and writes out new KV or KMV pages at most three times. It also uses a finite-size in-memory hash table, that stores unique keys as they are encountered while looping over the KV pages, as well as auxiliary information needed to construct the output pages of KMV pairs.

- (Pass 1) The KV pairs are read, one page at a time, and split into "partition" files, represented by KVp in the figure. Each partition file contains KV pairs whose unique keys (likely) fit in the hash table (HT). Initially, all KV pairs are assigned to the first partition file as the

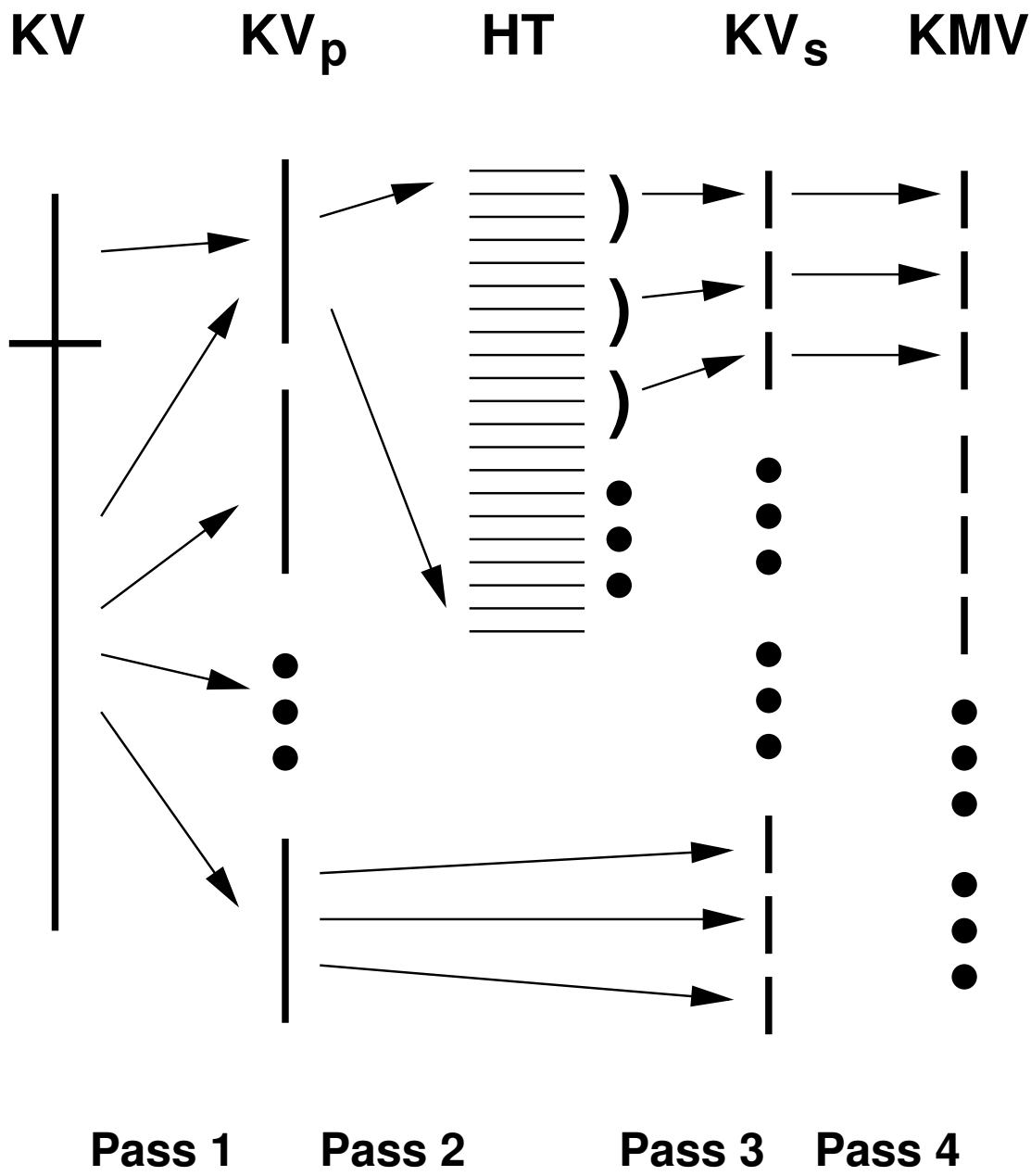

Figure 1: *Multi-pass algorithm for converting KV data to KMV data. The vertical lines represent out-of-core data sets. KV is key/value pairs; HT is an in-memory hash table; KMV is key/multivalue pairs.*

HT is populated. When the HT becomes full, e.g. at the point represented by the horizontal line shown on the leftmost vertical line as a downward scan is performed, the fraction of KV pairs read thus far is used to estimate the number of additional partition files needed. As subsequent KV pairs are read, they are assigned to the original partition file if the KV pair's key is already in the current HT. If not, a subset of bits in the hash value of the key is used to assign the KV pair to one of the new partition files. The number of needed partition files is estimated conservatively, rounding up to the next power-of-two, so that extra passes through the KV pairs are almost never needed, and so that the bit masking can be done quickly. This first pass entails both a read and write of all the KV pairs.

- (Pass 2) The partition files are read, one at a time. The key for each KV pair is hashed into the HT, accumulating data about the number and size of values associated with each unique key. Before the next partition file is read, passes 3 and 4 are completed for the current partition. Eventually this pass entails a read of all KV pairs.

- (Pass 3) The unique keys in the HT for one partition are scanned to determine the total size of KMV for keys in the HT. The associated values create KMV pairs that span M memory pages. If $M > 1$, the associated partition file of KV pairs is read again, and each KV pair is assigned to one of M smaller "set" files, represented by KVs in the figure. Each set file contains the KV pairs that contribute to the KMV pairs that populate one page of KMV output. If a single KMV pair spans multiple pages because it contains a very large number of values, the corresponding KV pairs are still written to a single set file. Eventually, this pass reads all partition files, entailing both a read and write of all KV pairs.

- (Pass 4) A set file is read and the key/value data for each of its KV pairs is copied into the appropriate location in the page of KMV pairs, using information stored in the HT. When complete, the KMV page is written to disk. This pass eventually reads all set files, entailing a final read of all KV pairs. It also writes all KMV pairs to disk. If each KV pair has a unique key, the volume of KMV output is roughly the same as that of the KV input.

In summary, this data reorganization for the *collate()* operation is somewhat complex, but requires only a small, constant number of passes through the data. The KV datums are read from disk at most four times, and written to disk three times. The first two write passes reorganize KV pairs into partition and set files; the final write pass creates the KMV data set. Depending on the size and characteristics of the KV datums (e.g., the number of unique keys), some of these passes may not be needed. By contrast, a full out-of-core sort of the KV pairs, performed via a merge sort as discussed below is still an $O(N \log_2 N)$ operation. The number of read/write passes through the KV data files depends on the amount of KV data that can be held in memory, but can be a large number for big data sets.

The memory page requirements for the out-of-core *collate()* operation are as follows. Two contiguous pages of memory are used for the hash table. Intermediate passes 2 and 3 can require numerous partition or set files to be opened simultaneously and written to. To avoid small writes of individual KV datums, a buffer of at least 16K bytes is allocated for each file. The precise number of simultaneously open files is difficult to bound, but typically one or two additional pages of memory suffice for all needed buffers. Thus the total number of pages required for the data reorganization stage of the *collate()* operation is less than the seven used by the all-to-all communication stage.

Other MR-MPI library calls discussed in the previous section, such as *clone()*, *collapse()*, and *gather()*, can also operate in out-of-core mode, typically with one pass through the KV or KMV data and the use of one or two memory pages. Sorting KV pairs, by key or value, is an exception.

An out-of-core merge sort is performed as follows. Two pages of KV datums are read from disk. Each is sorted in local memory using the C-library quicksort() function, as discussed in the previous section. The two pages are then scanned and merged into a new file which is written to disk as its associated memory page fills up. This process requires five memory pages, which includes vectors of KV datum pointers and lengths needed by the in-memory quicksort() operation. Once all pairs of pages have been merged, the sort continues in a recursive fashion, merging pairs of files into a new third file, without the need to perform additional in-memory sorts. Thus the overall memory requirement is five pages. The number of read/write passes through the KV data set for the merge sort is $O(\log_2 M)$, where M is the number of pages of KV pairs. The number of passes could be reduced at the cost of allocating more in-memory pages.

4 Graph Algorithms in MapReduce

We begin with a MapReduce procedure for creating large, sparse, randomized graphs, since they are the input for the algorithms discussed below. R-MAT matrices [7] are recursively generated graphs with power-law degree distributions. They are commonly used to represent web and social networks. The user specifices six parameters that define the graph: the number of vertices N and edges M, and four parameters a, b, c, d that sum to 1.0 and are discussed below. The algorithm in Figure 2 uses a single MapReduce object to generate M unique non-zero entries in a sparse $N \times N$ matrix G, where each entry G_{ij} represents an edge between graph vertices (V_i, V_j), with V_i an integer from 1 to N.

Input:
  MapReduce object G = edges of graph, initially empty

$M_{\text{remain}} = M$
while $M_{\text{remain}} > 0$:
  Map G: Generate M_{remain}/P random edges (V_i, V_j) on each processor
    Output: Key = (V_i, V_j), Value = NULL
  Collate G
  Reduce G: Remove duplicate edges
    Input: Key = (V_i, V_j), MultiValue = one or more NULLs
    Output: Key = (V_i, V_j), Value = NULL
  $M_{\text{remain}} = M - N_{kv}$

Output:
  MapReduce object G = edges of graph: Key = (V_i, V_j), Value = NULL

Figure 2: *MapReduce algorithm for R-MAT graph generation on P processors, using a single MapReduce object G.*

In the *map()*, each of P processors generates a $1/P$ fraction of the desired edges. A single random edge (V_i, V_j) is computed recursively as follows. Pick a random quadrant of the G matrix with relative probabilities a, b, c, and d. Treat the chosen quadrant as a sub-matrix and select a random quadrant within it, in the same manner. Repeat this process n times where $N = 2^n$. At the end of the recursion, the final "quadrant" is a single non-zero matrix element G_{ij}.

Though G is very sparse, the *map()* typically generates some small number of duplicate edges. The *collate()* and *reduce()* remove the duplicates. The entire map-collate-reduce sequence is re-

peated until the number of resulting key/value (KV) pairs $N_{kv} = M$. For reasonably sparse graphs this typically takes only a few iterations.

Note that the degree distribution of vertices in the graph depends on the choice of parameters a, b, c, d. If one of the four values is larger than the other three, a skewed distribution results. Variants of the algorithm can be used when N is not a power-of-two, to generate graphs with weighted edges (assign a numeric value to the edge), graphs without self edges (require $i \neq j$), or graphs with undirected edges (require $i < j$). Or the general R-MAT matrix can be further processed by MapReduce operations to meet these requirements. For some algorithms below (e.g., triangle finding and maximal independent set generation), we post-process R-MAT matrices to create upper-triangular R-MAT matrices representing undirected graphs. For each edge (V_i, V_j) with $i > j$, we add edge (V_j, V_i), essentially symmetrizing the matrix; we then remove duplicate edges and self edges, leaving only edges with $i < j$.

By using an additional MapReduce object, we can improve the performance of the R-MAT generation algorithm, particularly when out-of-core operations are needed. In the enhanced algorithm (Figure 3), we emit newly created edges into MapReduce object G_{new} and *aggregate()* the edges to processors. We then add those edges to the MapReduce object G, containing all previously generated unique edges; the edges of G are already aggregated to processors using the same mapping as G_{new}. We can use a *compress()* to reduce duplicates locally on each processor. The enhanced algorithm runs faster due to the smaller number of KV pairs communicated in the *aggregate()*, compared to the *collate()* of the original algorithm, particularly in later iterations of the algorithm where few new edges are generated.

Input:
    MapReduce object G = edges of graph, initially empty
    MapReduce object G_{new} = new edges of graph, initially empty

$M_{\text{remain}} = M$
while $M_{\text{remain}} > 0$:
    Map G_{new}: Generate M_{remain}/P random edges (V_i, V_j) on each processor
        Output: Key = (V_i, V_j), Value = NULL
    Aggregate G_{new}
    Add G_{new} to G, G_{new} = empty
    Compress G: Remove duplicate edges
        Input: Key = (V_i, V_j), MultiValue = one or more NULLs
        Output: Key = (V_i, V_j), Value = NULL
    $M_{\text{remain}} = M - N_{kv}$

Output:
    MapReduce object G = edges of graph: Key = (V_i, V_j), Value = NULL

Figure 3: *Enhanced MapReduce algorithm for R-MAT graph generation on P processors, using two MapReduce objects G and G_{new}.*

4.1 PageRank

The PageRank algorithm assigns a relative numeric rank to each vertex in a graph. In an Internet context, it models the web as a directed graph $G(V, E)$, with each vertex $V_i \in V$ representing a web page and each edge $(V_i, V_j) \in E$ representing a hyperlink from V_i to V_j. The probability of

moving from V_i to another vertex V_j is $\alpha/d_{out}(V_i)+(1-\alpha)/|V|$, where α is a user-defined parameter (usually 0.8-0.9), $d_{out}(V_i)$ is the outdegree of vertex V_i, and $|V|$ is the cardinality of V. The first term represents the probability of following a given link on page V_i; the second represents the probability of moving to a random page. For pages with no outlinks, the first term is changed to $\alpha/|V|$, indicating equal likelihood to move to any other page. Equivalently, the graph can be represented by a matrix A [16], with matrix entries $A_{ij} = \alpha/d_{out}(V_i)$ if vertex V_i links to V_j. To keep A sparse, terms for random jumps and zero out-degree vertices are not explicitly stored as part of the matrix. Rather, they are computed separately as adjustments to the PageRank vector. The kernel of the PageRank algorithm is thus a power-method iteration consisting of matrix-vector multiplications $A^T x = y$, where x is the PageRank vector from the previous iteration. A few additional dot product and norm computations per iteration are also required.

The algorithm in Figure 4 performs these iterations, using linear-algebra concepts implemented in a MapReduce framework, with data stored in 4 MapReduce objects: A, MT, x, and y. A stores the matrix, MT its "empty" rows, while x and y are vectors of length $|V|$. The nonzeros of matrix A are initialized as described above; the PageRank vector x is initialized uniformly. A MapReduce object MT contains the indices of all-zero rows of A, corresponding to vertices with no outlinks. The PageRank algorithm is made more efficient by invoking the *aggregate()* operation once on each of these MapReduce objects before the PageRank iterations begin. Since the 3 objects are all keyed by vertex, pre-aggregating them moves all KV pairs with a given key to the same processor. Many of the subsequent PageRank operations then become local operations, replacing a *collate()* (which is equivalent to an *aggregate()* followed by *convert()*) with simply a *convert()*. In Figure 4, steps where *convert()* is substituted for *collate()* are marked with an asterisk. This strategy of pre-aggregating to improve data locality is useful in many MapReduce algorithms, but assumes that identical keys in different MapReduce objects can be mapped to the same processor.

In Step 1 of a PageRank interation, MT is effectively dotted with x to compute an adjustment c_{dot} to the PageRank vector that represents the uniform probability of going from a leaf page to any other page. MT is unchanged by the operation. Adjustments for random jumps from any page are also computed. In Step 2, the matrix-vector product $A^T x$ is computed and the result stored in y. This is the most expensive part of the computation, requiring two passes over the nonzero structure of A. In the first pass, a *convert()* gathers all local row entries A_{ij} with their associated x_i entry, and a *reduce()* computes $A_{ij}x_i$. In the second pass, a *collate()* gathers, for each j, all contributions to the column sum $\sum_i A_{ij}x_i$, which is computed by a second *reduce()*. Steps 3 and 4 adjust and scale the product vector; the work is proportional to $|V|$. Step 5 computes the residual as an infinity norm to determine whether the PageRank vector has converged; work is again proportional to $|V|$. Note that y and x are both keyed by vertex at this point, so they can be joined via an *add()* and *convert()* without need for the communication required by a full *collate()*. In Step 6, the PageRank vector is overwritten prior to the next iteration. Throughout the algorithm, we exploit the ability to call MPI_Allreduce to compute global dot products and norms.

This linear-algebra approach is attractive because it allows construction of the MapReduce PageRank algorithm from simple kernels (matrix-vector multiplication, dot products) on commonly used data types (matrices, vectors). One can build a variety of algorithms based on these primitives; indeed, linear-algebra toolkits such as Trilinos [14] and PETSc [3] naturally exploit this idea. In parallel, these toolkits share pre-computed data layouts and communication maps among distributed objects to enable efficient operations on the objects. In a MapReduce context, however, separating the data into vectors and matrices results in additional overhead to gather KV pairs with the same key into a key-multivalue (KMV) pair. This overhead could be reduced in the PageRank algorithm, for example, by storing the x and y vectors in a single MapReduce

object, with key i and value (x_i, y_i). Doing this would eliminate the *copy()* in Step 2 of Figure 4, as well as the *add()* and *convert()* in Step 5. A further performance improvement would store an entire row of A along with x and y in a single MapReduce object, with key i and value $(x_i, y_i, d_{out}, [j_1, A_{ij_1}], [j_2, A_{ij_2}], \ldots, [j_{d_{out}}, A_i j_{d_{out}}])$ (that is, storing together, for each vertex, its old and new PageRank value along with all out-going edges incident to the vertex). This data layout would eliminate an additional it convert(), but at the cost of code reusability. This data layout would be efficient for PageRank, but is more complex and less likely to be useful for additional algorithms. In addition, since each KV pair must fit within a single page of memory (in MR-MPI), storing entire matrix rows in a single KV pair limits the size of matrices to which the algorithm can be applied. The more general approach used in Figure 4 of storing non-zeros separately, enables arbitrarily large problems to be solved. The detailed timing break-downs in section 5.2 allow estimation of the benefit of these possible enhancements.

4.2 Triangle enumeration

A triangle in a graph is any triplet of vertices (V_i, V_j, V_k) where the edges $(V_i, V_j), (V_j, V_k), (V_i, V_k)$ exist. Figure 5 outlines a MapReduce algorithm that enumerates all triangles, assuming an input graph G of undirected edges (V_i, V_j) where $V_i < V_j$ for every edge, i.e. an upper-triangular R-MAT matrix. This exposition follows the triangle-finding algorithm presented in [8].

The initial step is to store a copy of the graph edges as KV pairs in an auxiliary MapReduce object G_0, for use later in the algorithm. The first *map()* on G converts edge keys to vertex keys with edge values. After a *collate()*, each vertex has the list of vertices it is connected to; the first *reduce()* can thus flag one vertex V_i in each edge with a degree count D_i. The second *collate()* and *reduce()* assign a degree count D_j to the other vertex in each edge. In the third *map()*, only the lower-degree vertex in each edge emits its edges as KV pairs. The task of the third *reduce()* is to emit "angles" for each of these low-degree vertices. An "angle" is a root vertex V_i, with two edges to vertices V_1 and V_2, i.e. a triangle without the third edge (V_1, V_2). The *reduce()* emits a list of all angles of vertex V_i, by a double loop over the edges of V_i. Note that the aggregate number of KV pairs emitted at this stage is minimized by having only the low-degree vertex in each edge generate angles.

In stage 4, KV pairs from the original graph G_0 are added to the current working set of KV pairs in MapReduce object G. The KV pairs in G_0 contain edges that complete triangles for the angle KV pairs just generated. After the fourth *collate()*, a pair of vertices (V_i, V_j) is the key, and the multivalue is the list of all root vertices in angles that contain V_i and V_j. If the multivalue also contains a NULL, contributed by G_0, then there is a (V_i, V_j) edge in the graph. Thus all vertices in the multivalue are roots of angles that are complete triangles and can be emitted as a triplet key.

4.3 Connected component labeling

A connected component of a graph is a set of vertices where all pairs of vertices in the set are connected by a path of edges. A sparse graph may contain many such components. Figure 6 outlines a MapReduce algorithm that labels each vertex in a graph with a component ID. All vertices in the same component are labelled with the same ID, which is the ID of some vertex in the component. We assume an input graph E of undirected edges (V_i, V_j). This exposition also follows the connected-component algorithm presented in [8], with the addition of logic that load-balances data across processors, which is important when one or a few giant components exist in the graph.

The algorithm begins (before the iteration loop) by assigning each vertex to its own component or

"zone," so that $Z_i = V_i$ and storing this assignment in MapReduce object V. Each iteration grows the zones, one layer of neighbors at a time. As zones collide due to shared edges, a winner is chosen (the smaller zone ID), and vertices in the losing zone are reassigned to the winning zone. When the iterations complete, each zone has become a fully connected component and the MR object V holds the zone assignment for every vertex. The algorithm thus finds all connected components in the graph simultaneously. The number of iterations required depends on the largest diameter of any component in the graph.

The first *map()* uses MapReduce object E as input to the empty MapReduce object W to emit the vertices in each edge as keys, with the edge as a value. The current zone assignment of each vertex in V is added to the set of KV pairs in W. The first *collate()* collects all the edges of a vertex and its zone assignment together in one multivalue. The first *reduce()* re-emits each edge, tagged by the zone assignment of one of its vertices.

Since each edge was emitted twice, the second *collate()* on W collects the zone assignments for its two vertices together. If the two zone IDs are different, the second *reduce()* chooses a winner (the smaller of the two IDs), and emits the loser ID as a key, with the winning ID as a value. If no zone ID changes are emitted, the algorithm is finished, and the iterations cease.

The third *map()* inverts the vertex/zone KV pairs to become zone/vertex KV pairs. The partitioning logic described in Figure 6 for this step (and subsequent ones) is described below. The fourth *map()* adds the changed zone assignments in W to the set of KV pairs in V. The fourth *collate()* can now collect all the vertices in each zone along with any reassignments of that zone ID. Since a zone could collide with multiple other zones on the same iteration due to shared edges, the new zone ID for a zone becomes the minimum ID of any of its neighboring zones. If no zone reassignment values appear in the multivalue resulting from the fourth *collate()*, the zone ID is unchanged. The final *reduce()* emits a KV pair for each vertex in the zone, with the vertex as a key and the new zone ID as a value. This is an updated version of MapReduce object V, ready to be used as input for the next iteration.

Note that if a graph has only a few components, the fourth *collate()*, which keys on the zone ID, may generate a few very large key/multivalue (KMV) pairs. For example, if the entire graph is fully connected, in the last iteration, a single KMV pair assigned to one processor will result, containing all vertices in the graph. This imbalance in memory and computational work can lead to poor parallel performance of the overall algorithm. To counter this effect, various operations in stages 3 and 4 include extra logic. The idea is to partition zones whose vertex count exceeds a user-defined threshhold into P sub-zones, where P is the number of processors. The 64-bit integer that stores the zone ID also stores a bit flag indicating the zone has been partitioned and a set of bits that encode the processor ID.

During the third *map()*, if the zone has been partitioned, the vertex is assigned to a random processor and the processor ID bits are added to the zone ID, as indicated by the Z_i^+ notation in Figure 6. Likewise, if Z_i has been partitioned, the fourth *map()* emits the KV pair for the zone ID as (Z_i^+, Z_{winner}) instead of as (Z_i, Z_{winner}). In this case (Z_i, Z_{winner}), is emitted not once, but $P+1$ times, once for each processor, and once as if Z_i had not been partitioned (for a reason discussed below).

With this additional partitioning logic, the fourth *collate()* (which keys on zone IDs, some of which now include processor bits) collects only $1/P$ of the vertices in partitioned zones onto each processor. Yet the multivalue on each processor still contains all the zone-reassignments relevant to the unpartitioned zone. Thus, the fourth *reduce()* can change the zone ID (if necessary) in a consistent manner across all P multivalues that contain the zone's vertices. When the fourth *reduce()* emits new zone assignments for each vertex, the zone retains its partitioned status; the partition bit is also explicitly set if the vertex count exceeds the threshhold for the first time.

Note that this logic does not guarantee that the partition bits of the zone IDs for all the vertices in a single zone are set consistently on a given iteration. For example, an unpartitioned zone with a small ID may consume a partitioned zone. The vertices from the partitioned zone retain their partitioned status, but the original vertices in the small zone may not set the partition bit of their zone IDs. On subsequent iterations, the fourth *map()* emits $P+1$ copies of new zone reassignments for both partitioned and unpartioned zone IDs, to ensure all vertices in the zone will know the correct reassignment information.

4.4 Maximal independent set identification

An "independent" set of vertices from a graph is one where no pair of vertices in the set is connected by an edge. The set is "maximal" if no vertex can be added to it [1]. Finding a maximal independent set (MIS) is useful in several contexts, such as identifying large numbers of independent starting vertices for graph traversals. Luby's algorithm [17] is a well-known parallel method for finding a MIS. Figure 7 outlines a MapReduce version of Luby's algorithm, assuming an input graph of undirected edges (V_i, V_j) where $V_i < V_j$ for every edge, i.e. an upper-triangular R-MAT matrix.

Before the iterations begin, a random value is assigned to each vertex (stored consistently for all edges in E containing V_i), which is used to compare pairs of vertices. The vertex ID itself could be used as the random value, but since the vertices eventually selected for the MIS depend on the randomization, this may introduce an unwanted bias. Instead, a random number generator can be used to assign a consistent random value R_i to each vertex in each edge (via a *map()*), which is then carried along with the vertex ID through each stage of the algorithm. In the notation of Figure 7, each V_i is then really two quantities, a vertex ID (1 to N) and R_i. Alternatively, the R_i can be used to relabel the vertices in a random manner, assigning each a new, unique vertex ID from 1 to N. In this case, V_i is simply the new vertex ID, and the vertices in the final MIS could be remapped to recover their original IDs. These operations can be performed with a few extra MapReduce steps, outside the iteration loop.

The pre-iteration *clone()* converts the initial list of edge key/value (KV) pairs stored in MapReduce object E to key/multivalue (KMV) pairs, with the edge as the key, and NULL as the value. Thus, the iterations can begin with a *reduce()*, without need for a *collate()*. A second empty MapReduce object V is also created, which will accumulate MIS vertices as the iterations proceed.

At each iteration, the current set of edges in E is examined to identify winning vertices. Vertices adjacent to winners are flagged as losers, and all edges which contain a loser vertex are marked for removal at the start of the next iteration. Vertices with all their edges marked as losers are also flagged as winners. At each iteration marked edges are removed from the graph, and winning vertices are added to the MIS. When the graph is empty, the MIS is complete.

The first *reduce()* flags the two vertices in each edge as a potential winner or loser. This is done by comparing the two random values for the vertices (or the randomized vertex IDs as explained above). The first *collate()* thus produces a multivalue for each vertex V_i which contains all the vertices it shares an edge with, and associated winner/loser flags.

In the second *reduce()*, if V_i won the comparison with all its neighbor vertices, then it becomes a "winner" vertex. All vertices connected to the winner are emitted with V_i tagged with a winner flag. If V_i is not an overall winner, its neighbor vertices are emitted without the winner flag on V_i.. The second *collate()* collects this new information for each vertex.

In the third *reduce()*, if any neighbor of V_i was a winner in the previoue *reduce()* (indicated by a winner flag), V_i becomes a "loser" vertex and emits all its neighbor vertices with V_i tagged with

[1] A maximal set is "maximum" if it is the largest maximal set. Finding a graph's maximum independent set is an NP-hard problem, which MapReduce is unlikely to help with.

a loser flag. Otherwise it emits all its neighbor vertices without the loser flag on V_i.

In the fourth *reduce()*, some vertices have all their neighbor vertices flagged as losers; these vertices become winners. They are either the vertices identified as winners in the second *reduce()*, or vertices who would become singletons (having no edges) when edges containing a loser vertex are removed (on the next iteration). These winning vertices are emitted to the MapReduce object V that stores the accumulating MIS.

All edges of each vertex are also emitted, retaining the loser flag if they have it; they are emitted with $E_{ij} = (V_i, V_j)$ as their key, with $V_i < V_j$. This is to ensure both copies of the edge come together via the fourth *collate()*. The multivalue for each edge now has two values, each of which is either NULL or a loser flag.

In the first *reduce()* of the next iteration, any edge with a loser flag in its multivalue does not compare its two vertices; it is effectively deleted from the graph. The iterations end when all edges have been removed from the graph. At this point, the MapReduce object V contains a complete MIS of vertices.

4.5 Single source shortest path calculation

For a directed graph with weighted edges, the single-source shortest path (SSSP) algorithm computes the shortest weighted distance from a chosen source vertex to all other vertices in the graph. This calculation is straightforward when global graph information is available. The source vertex is labeled with distance zero. Then in each iteration, edge adjacencies of labeled vertices are followed, and adjacent vertices are relabeled with updated distances. When no distances change, the iterations are complete. Most importantly, only edges adjacent to modified vertices need be visited in each iteration.

For a distributed graph, global information about edge adjacencies and labeled vertices is not available. Instead, following the Bellman-Ford-style algorithm of [6, 4, 12], every edge of the graph is visited in each iteration and vertex distancs are updated. A MapReduce formulation of this algorithm, described in Figure 8, begins with a graph E of directed edges (V_i, V_j). As generated by the R-MAT procedure disussed earlier, this may include self-edges (V_i, V_i) and loop edges (both (V_i, V_j) and (V_j, V_i)). These are handled as special cases within the algorithm to avoid spurious contributions to the shortest path distances. MapReduce object V stores the distance of each vertex V_i from the source vertex V_s.

To begin, the distance to all vertices is set to infinity, except a distance of 0 for V_s itself. At each iteration, a *map()* is performed using E as input to add edges and their weights to each vertex in V. After a *collate()*, each vertex V_i knows its outgoing edges and their weights, as well as its own current distance. It may also have one or more distances assigned to it from a previous iteration (from an inbound edge). In the *reduce()*, each vertex resets its distance to the minimum of these new distances and its current distance. If its distance changed, it emits new distance values d_j for the vertices associated with its outgoing edges, where $d_j = d_i + w_{ij}$. It also unsets a "done" flag, stored by each processor. When the *reduce()* is complete, an MPI_Allreduce is performed to check the "done" flag status on all processors. If any distance was changed, the iterations continue, and the distance updates propagate through the graph. Otherwise the algorithm is done.

As with PageRank and R-MAT generation, significant savings in communication and execution time can be achieved by pre-aggregating the data stored in MapReduce objects and by splitting the data appropriately across more than one MapReduce object. An enhanced SSSP algorithm is shown in Figure 9. The MapReduce objects V and E which store vertices and edges are aggregated across processors only once at the beginning of the computation, which means the communication portion of the *collate()* is performed to assign each key to a processor. The *collate()* of Figure 8

Data Set	# of vertices	# of edges	Maximum vertex degree
RMAT-20 (small)	$2^{20} \approx 1M$	$2^{23} \approx 8M$	$\approx 24K$
RMAT-24 (medium)	$2^{24} \approx 17M$	$2^{27} \approx 134M$	$\approx 147K$
RMAT-28 (large)	$2^{28} \approx 268M$	$2^{31} \approx 2B$	$\approx 880K$

Table 1: Characteristics of R-MAT input data for graph algorithm benchmarks.

can then be replaced with a *compress()* which requires no communication.

In each iteration of SSSP, typically only a small number of new distances are generated. The enhanced algorithm uses MapReduce object U to store these updates. Only the updates in U are communicated; which aggregates them to the same processor that permanently stores the corresponding vertex and its edges. Thus, the total amount of required communication is smaller than in the original algorithm, which communicated all graph edges at each iteration. Updated vertex distances are stored in MapReduce object V, as before.

5 Performance Results

In this section, we present performance results for the graph algorithms of the preceeding section, implemented as small C++ programs calling our MR-MPI library. The benchmarks were run on a medium-sized Linux cluster of 2 GHz dual-core AMD Opteron processors connected via a Myrinet network. Most importantly for MR-MPI, each node of the cluster has one local disk, which is used for out-of-core operations. To avoid contention for disk I/O, we ran all experiments with one MPI process per node. For comparisons with other implementations, we used either the same cluster or, where noted, Sandia's Cray XMT, a multi-threaded parallel computer with 500 MHz processors and a 3D-Torus network.

We ran the algorithms on three R-MAT graphs of different sizes, each for a varying number of processors. Details of the input data are shown in Table 1. The *small* problem (around 8M edges) can typically be run on a single processor without incurring out-of-core operations. The *medium* problem (around 134M edges) can be run on a single processor with out-of-core operations; larger processor configurations, however, do not necessarily require out-of-core operations. The *large* problem (around 2B edges) requires out-of-core operations on our 64-node cluster. These are edge counts before pre-processing steps used in some of the algorithms, e.g. to eliminate self-edges or convert the matrix to upper-triangular form.

All data sets used R-MAT parameters $(a, b, c, d) = (0.57, 0.19, 0.19, 0.05)$ and generated 8 edges per vertex (on average). These parameters create a highly skewed degree distribution, as indicated by the maximumn vertex degree in Table 1.

The resulting timings give a sense of the inherent scalability of the MapReduce algorithms as graph size grows on a fixed number of processors, and of the parallel scalability for computing on a graph of fixed size on a growing number of processors. Where available, we compare the MapReduce algorithm with other parallel implementations, including more traditional distributed-memory algorithms and also multi-threaded algorithms in the Multi-Threaded Graph Library (MTGL) [5] on the Cray XMT. We compute parallel efficiency on P processors as $(time_M \times M)/(time_P \times P)$ where M is the smallest number of processors on which the same graph was run, and $time_I$ is the execution time required on I processors.

5.1 R-MAT generation results

In Figure 10, we show the scalability of the R-MAT generation algorithm (Figure 3) for RMAT-20, RMAT-24 and RMAT-28. For RMAT-20 and RMAT-24, the algorithm exhibits superlinear speed-up. This is due to the decreased amount of file I/O needed with greater numbers of processors; with a larger total memory, a fixed-size problem requires less I/O as processors are added. For RMAT-28, which requires significant out-of-core operations, parallel efficiency ranges from 52% to 97%. For these problems the enhanced algorithm of Figure 3 ran about 10% faster than the algorithm of Figure 2.

5.2 PageRank results

In Figure 11, we show the performance of the PageRank algorithm (Figure 4) compared to a distributed-memory matrix-based implementation using the linear algebra toolkit Trilinos [14]. The Trilinos implementation of PageRank uses Epetra matrix/vector classes to represent the graph and PageRank vectors and requires the matrix and associated vectors fit in the aggregate memory of the processors used. Rows of matrix A and the associated entries of the PageRank vector x are uniquely assigned to processors; a random permutation of the input matrix effectively load balances the non-zero matrix entries across processors. Interprocessor communication gathers x values for matrix-vector multiplication and sums partial products into the y vector. Most communication is point-to-point communication, but some global communication is needed for computing residuals and norms of x and y.

Figure 11 shows the execution time per PageRank iteration for R-MAT matrices RMAT-20, RMAT-24 and RMAT-28. Converging the PageRank iterations to tolerance 0.002 requires five or six iterations. Several R-MAT matrices of each size were generated for the experiments; the average time over the set of matrices is reported here. The Trilinos implementations show near-perfect strong scaling for RMAT-20 and RMAT-24. The MR-MPI implementations also demonstrate good strong scaling. However, MR-MPI's execution time is more than an order of magnitude slower than the Trilinos implementation. The benefit of MR-MPI's out-of-core implementation is seen, however, with the RMAT-28 data set, which could be run on smaller processor sets than the Trilinos implementation, which required 64 processors for RMAT-28.

A more detailed analysis of the performance of the MR-MPI PageRank algorithm is shown below. We perform the analysis using RMAT-20, which is small enough to fit in memory and thus does not need out-of-core operations, and RMAT-28, which requires out-of-core operation in MR-MPI. Table 2 shows the amount of time spent in each step of Figure 4 for the MR-MPI implementation and the equivalent Trilinos implementation. For both implementations, the matrix-vector multiplication (Step 2) is the most costly step. However, the MR-MPI algorithm is consistently slower than the Trilinos implementation in each step. This effect is more pronounced for RMAT-28, where MR-MPI performs out-of-core operations.

To understand where MR-MPI spends its effort during the PageRank computation, we break down the operation times further for the most expensive Step 2 of the PageRank computation; the results are in Table 3. In the table, we divide the *collate()* operation into its two parts: an *aggregate()* that involves parallel communication and a *convert()* that gathers local key/value (KV) pairs with matching keys into a key-multivalue (KMV) pair. All interprocessor communication occurs in the *aggregate()*. While the time spent in *aggregate()* is significant, it is small compared to the time spent in the two *convert()* operations. When out-of-core operations are required for RMAT-28, the communication cost in the *aggregate()* is a smaller percentage of total time for Step 2, as other data manipulation operations such as *convert()* and *add()* require a relatively larger percentage of the total time. Further experimentation is needed to determine whether reorganizing

	RMAT-20		RMAT-28	
	MR-MPI	Trilinos	MR-MPI	Trilinos
Step 1 (Compute adjustment)	0.0203	0.00030	17.53	0.0838
Step 2 (Multiply $A^T x$)	0.1377	0.00455	341.26	2.9239
Step 3 (Apply adjustment)	0.0015	0.00015	2.14	0.0494
Step 4 (Scale y)	0.0013	0.00014	1.74	0.0359
Step 5 (Compute residual)	0.0207	0.00014	21.20	0.0457

Table 2: Detailed execution times (in seconds) per PageRank iteration for each stage of the algorithm in Figure 4 using MR-MPI and Trilinos.

the MapReduce data, as described in Section 4.1. would reduce the execution time of Step 2. Organizing the data such that both the matrix A and vectors x and y are in the same MapReduce object would eliminate the *copy()* and *add()* functions, as well as the first *convert()*. However, to update the y values at the completion of Step 2, either an additional *convert()* or *aggregate()* with greater amounts of data would be needed.

PageRank Step 2	RMAT-20	RMAT-28
Copy x to y	1.1%	0.4%
Add A to y	3.0%	14.3%
Convert y	23.5%	43.6%
Reduce y	7.7%	5.8%
Aggregate y (first part of collate)	24.9%	6.8%
Convert y (second part of collate)	38.2%	27.4%
Reduce y	1.6%	1.7%

Table 3: Percentage of execution time in each operation of Step 2 of the MR-MPI PageRank algorithm in Figure 4.

Similarly, in Table 4, we show the execution times required for each operation in Step 5 of Figure 4, for the residual computation. Clearly the time to convert KV pairs to KMV pairs dominates this computation. Reorganization of the vectors x and y into a single MapReduce object (as described in Section 4.1) would benefit this step of the computation, as the *add()*, *convert()*, and *reduce()* could be replaced by a single *map()*, with execution time comparable to the vector-scaling operation of Step 4.

PageRank Step 5	RMAT-20	RMAT-28
Add y to x	2.2%	9.8%
Convert x	95.1%	84.9%
Reduce x	2.5%	5.3%
MPI_Allreduce	0.2%	0.0%

Table 4: Percentage of execution time in each operation of Step 5 of the MR-MPI PageRank algorithm in Figure 4.

5.3 Triangle finding results

In Figure 12, we show the performance of the triangle finding algorithm (Figure 5). Execution times for this algorithm were too large to allow the problem to be easily run with RMAT-28 on our cluster. Parallel efficiencies for RMAT-20 ranged from 80% to 140%; for RMAT-24, they ranged from 50% to 120%. The RMAT-20 matrix contained about 100 million triangles; the RMAT-24

contained 2.1 billion.

5.4 Connected Components

In Figure 13, we show the performance of the connected component identification algorithm (Figure 6). A comparison is made with a hybrid "Giant Connected Component" implementation using both Trilinos and MR-MPI. Power law graphs often have one or two very large components and many very small components. The hybrid algorithm exploits this feature by using an inexpensive breadth-first search (BFS) from the vertex with highest degree to identify the largest components one at a time, followed by a more expensive algorithm to identify the small components in the remainder of the graph. In our hybrid implementation, we intially perform matrix-vector multiplications in Trilinos to perform a BFS, finding a handful of components that include 70% or more of the vertices (for these R-MAT matrices). We then apply our MapReduce algorithm to the remaining graph to identify the small components. This hybrid approach is quite effective, reducing the execution time to identify all components in RMAT-20 and RMAT-24 by 80-99%, as shown in Figure 13. For RMAT-28, where the graph is too large to fit into memory, Trilinos cannot perform the initial BFS, so only the MapReduce algorithm was used. The total number of components identified were A, B, C for the 3 increasing R-MAT sizes; the number of iterations required by the algorithm to find them was X, Y, Z respectively.

5.5 Maximally independent set results

The execution times for the maximal independent set (MIS) algorithm (Figure 7) are shown in Figure 14. Like the R-MAT generation results, superlinear speed-up of the algorithm occurs for RMAT-20 and RMAT-24, as more of the graph fits into processor memory and less file I/O is needed. For RMAT-28, the algorithm requires significant out-of-core operations. In this case, parallel efficiency is nearly perfect going from 8 to 64 processors. The number of vertices in the resulting MIS were A, B, C for the 3 increasing R-MAT sizes; the number of iterations required by the algorithm to find them was X, Y, Z respectively.

5.6 Single-source shortest path results

The execution times for the SSSP algorithms of Figures 8 and 9 are shown in Figure 15. The results show the benefit of using the enhanced algorithm, which gave at least a 17% (and often greater) reduction in execution time due to reduced communication; only the updated distances are communicated throughout most of the enhanced algorithm. However, the execution times are still large compared to multi-threaded implementations; for example, Madduri et al. [18] report execution times of only 11 seconds on 40 processors for a multithreaded implementation on a Cray MTA-2 (the predecessor of the XMT) on graphs with one billion edges. The number of iterations required in the MapReduce algorithms to find the SSSP to all vertices for the 3 increasing R-MAT sizes was X, Y, Z respectively.

To compare our MR-MPI implementation with a wider set of algorithms, we performed experiments comparing MR-MPI, Hadoop, PBGL (Parallel Boost Graph Library) [13] and a multithreaded implementation on the Cray XMT using two web graphs: WebGraphA with 13.2M vertices and 31.9M edges, and WebGraphB with 187.6M vertices and 531.9M edges. In Figure 16, we show execution times for the SSSP algorithm using WebGraphA. Like our MR-MPI implementation, the Hadoop implementation is a Bellman-Ford-style [4, 12] algorithm. The XMT and PBGL implementations are based on delta-stepping [19], and do not require full iterations over the entire edge list to advance a breadth-first search. We observe that the MR-MPI implementation runs in less

Implementation	Number of Processes	SSSP Execution Time
Hadoop	48	38,925 secs.
MR-MPI original (Figure 8)	48	13,505 secs.
MR-MPI enhanced (Figure 9)	48	8,031 secs.
XMT/C	32	37 secs.

Table 5: Execution times for SSSP with WebGraphB.

Data Set	R-MAT a	R-MAT b	R-MAT c	R-MAT d	# of vertices	# of edges	Maximum vertex degree
nice	0.45	0.15	0.15	0.25	2^{25}	2^{28}	1108
nasty	0.57	0.19	0.19	0.05	2^{25}	2^{28}	230,207

Table 6: Characteristics of R-MAT input data for PageRank and Connected Components scalability experiments.

time than the Hadoop implementation, but requires significantly more time than the XMT and PBGL implementations. In experiments with WebGraphB, the benefit of the enhanced algorithm (Figure 9) is clearly shown, with a 40% reduction in execution time compared to the original SSSP algorithm (Figure 8). But the Bellman-Ford-style iterations are especially costly in the MR-MPI and Hadoop implementations for WebGraphB, which required 110 iterations to complete; execution times for this data set are shown in Table 5.

5.7 Scalability to large numbers of processors

Finally, we demonstrate the scalability of our MR-MPI library to large numbers of processors. The library was used on Sandia's Redstorm and Thunderbird parallel computers. Redstorm is a large Cray XT3 with 2+ GHz dual/quad-core AMD Opteron processors and a custom interconnect providing 9.6 GB/s of interprocessor bandwidth. Thunderbird is a large Linux cluster with 3.6 GHz dual-core Intel EM64T processors connected by an Infiniband network. Because these systems do not have local disks for each processor, we selected a data set and page sizes that fit in memory, so out-of-core operations were not needed. For these experiments, we used two R-MAT data sets with with 2^{25} vertices and 2^{28} edges, with parameters given in Table 6. The "nice" parameters produce a less-skewed sparsity pattern in the matrix than the "nasty" parameters, as indicated by the maximum degree metric. We ran both the PageRank and Connected Components algorithms on these matrices.

Figure 17, shows the performance of the various PageRank implementations on distributed memory and multi-threaded architectures. The MR-MPI and Trilinos implementations are described in Sections 4.1 and 5.2, respectively. In the multi-threaded MTGL implementation, rank propagates via adjacency list traversal in a compressed sparse-row data structure. To maintain its scalability, code must be written so that a single thread spawns the loop that processes all in-neighbors of a given vertex; this detail enables the compiler to generate hotspot-free code.

The MR-MPI implementation demonstrated good scalability up to 1024 processors; however, as before, it required an order-of-magnitude more execution time than the matrix-based implementations on Redstorm. The Trilinos implementation is competitive with the multi-threaded implementation in MTGL on the Cray XMT, when both are run on the same number of processors.

Similar results were obtained for the Connected Components algorithm, as shown in Figure 18. As with PageRank, the MR-MPI implementation showed good scalability up to 1024 processors,

but required significantly more time than the hybrid algorithm using Trilinos and MR-MPI or the MTGL algorithm.

6 Lessons Learned

We conclude with several observations about performing MapReduce operations on distributed-memory parallel machines via MPI.

MapReduce achieves parallelism through randomizing the distribution of data across processors, which often intentionally ignores data locality. This translates into maximal data movement (during a data shuffle or *collate()* operation) with communication between all pairs of processors. But the benefit is often good load-balance, even for hard-to-balance irregular data sets, such as those derived from sparse graphs with power-law degree distributions. By contrast, MPI-based parallel algorithms, e.g. for matrix operations, or grid-based or particle-based simulation codes, typically work hard to localize data and minimize communication. To do this they often require application-specific logic and parallel communication coding to create and maintain a data decomposition, generate ghost copies of nearby spatially-decomposed data, etc.

MapReduce algorithms can be hard to design, but are often relatively easy to write and debug. Thinking about a computational task from a MapReduce perspective is different from traditional distributed-memory parallel algorithm design. For example, with an MPI mindset, it often seems heretical to intentionally ignore data locality. However, writing small *map()* and *reduce()* functions is typically easy. And writing an algorithm that involves complex parallel operations without needing to write application-specific code to communicate data via MPI calls or to move data between out-of-core disk files and processor memory is often a pleasant surprise. By contrast, extending a library like Trilinos (or other in-core MPI applications) to enable it to perform its suite of linear algebra operations in out-of-core mode on large data sets would be a considerable effort. Moreover, if a MapReduce algorithm is initially coded so that it runs correctly on one processor, it often works out-of-the-box on hundreds or thousands of processors, without the need for additional debugging.

Performing MapReduce operations on a fixed allocation of processors on a traditional MPI-based parallel machine is a somewhat different conceptual model than performing MapReduce operations on a cloud computer using (for example) Hadoop. In the former case, one can potentially control which processor owns which data at various stages of an algorithm. This is somewhat hidden from the user in a typical cloud-computing model, where data simply exists somewhere in the cloud and Hadoop ensures data moves where it is needed and is operated on by some processor. The cloud model is a nice data-centric abstraction which allows for fault tolerance both to data loss (via redundant storage) and to processor failure (via reassignment of work), neither of which is typically possible on current MPI-based parallel machines.

However, since the MPI implementation of MapReduce as described in this paper is processor-centric, one can sometimes fruitfully exploit the possibility for processors to maintain "state" over the course of multiple map and reduce operations. By controlling where data resides for maps and reduces (e.g. via a user-specified hash function), and by assuming that processor will always be available, more efficient operations with less data movement are sometimes possible. The discussion of enhanced graph algorithms in Section 4 illustrated this. To fully exploit this idea for large data sets, mechanisms and data structures are needed for processors to store, retrieve, and efficiently find needed datums on local disk (e.g. static edges of a graph), so that archived state can be used efficiently during subsequent map and reduce operations.

Finally, though this paper focuses on graph algorithms expressed as MapReduce operations,

there is nothing about the MR-MPI library itself that is graph specific. We hope the library can be generally useful on large-scale monolithic or cloud-style parallel machines that support MPI, for a variety of data-intensive or compute-intensive problems that are amenable to solution using a MapReduce approach.

7 Acknowledgements

The MR-MPI library is open-source software, which can be downloaded from `http://www.sandia.gov/~sjplimp/mapreduce.html`. It is freely available under the terms of a BSD license. Benchmark programs which call the library to implement the algorithms of and which produced the timing results of Section 5 are included in the distribution.

We thank the following individuals for their contributions to this paper: Greg Bayer and Todd Plantenga (Sandia) for explaining Hadoop concepts to us, and for the Hadoop implementations and timings of Section 5; Jon Cohen (DoD) for fruitful discussions about his MapReduce graph algorithms [8]; Brian Barrett (Sandia) for the PBGL results of Section 5; Jon Berry (Sandia) for the MTGL results of Section 5, and for his overall support of this work and many useful discussions.

Sandia National Laboratories is a multi-program laboratory operated by Sandia Corporation, a wholly owned subsidiary of Lockheed Martin company, for the U.S. Department of Energy's National Nuclear Security Administration under contract DE-AC04-94AL85000.

References

[1] Apache Hadoop WWW site: `http://hadoop.apache.org/`.

[2] Titan Informatics Toolkit WWW site: `http://titan.sandia.gov/`.

[3] S. Balay, W. D. Gropp, L. C. McInnes, and B. F. Smith. Efficient management of parallelism in object oriented numerical software libraries. In E. Arge, A. M. Bruaset, and H. P. Langtangen, editors, *Modern Software Tools in Scientific Computing*, pages 163–202. Birkhäuser Press, 1997.

[4] R. Bellman. On a routing problem. *Quarterly of Applied Mathematics*, 16(1):87–90, 1958.

[5] J. W. Berry, B. Hendrickson, S. Kahan, and P. Konecny. Software and algorithms for graph queries on multithreaded architectures. In *Proceedings of the 21st International Parallel and Distributed Processing Symposium*, March 2007.

[6] C. Bisciglia, A. Kimball, and S. Michels-Slettvet. Lecture 5: Graph algorithms and PageRank, 2007. `http://code.google.com/edu/submissions/mapreduce-minilecture/lec5-pager%ank.ppt`.

[7] D. Chakrabarti, Y. Zhan, and C. Faloutsos. R-MAT: A recursive model for graph mining. In *SIAM Data Mining*, 2004.

[8] J. Cohen. Graph twiddling in a mapreduce world. *Computing in Science and Engineering*, 11:29–41, 2009.

[9] J. Dean and S. Ghemawat. MapReduce: Simplified data processing on large clusters. In *OSDI04: Sixth Symposium on Operating System Design and Implementation*, 2004.

[10] J. Dean and S. Ghemawat. Mapreduce: simplified data processing on large clusters. *Commun. ACM*, 51(1):107–113, 2008.

[11] J. Ekanayake and G. Fox. High performance parallel computing with clouds and cloud technologies. In *First International Conference on Cloud Computing (CloudComp09)*, 2009.

[12] L. R. Ford and D. R. Fulkerson. *Flows in networks*. Princeton University Press, 1962.

[13] D. Gregor and A. Lumsdaine. The parallel BGL: A generic library for distributed graph computations. *Parallel Object-Oriented Scientific Computing*, July 2005.

[14] M. Heroux, R. Bartlett, V. H. R. Hoekstra, J. Hu, T. Kolda, R. Lehoucq, K. Long, R. Pawlowski, E. Phipps, A. Salinger, H. Thornquist, R. Tuminaro, J. Willenbring, and A. Williams. An Overview of Trilinos. Technical Report SAND2003-2927, Sandia National Laboratories, 2003.

[15] T. Hoefler, A. Lumsdaine, and J. Dongarra. Towards efficient mapreduce using mpi. In M. Ropo, J. Westerholm, and J. Dongarra, editors, *PVM/MPI*, volume 5759 of *Lecture Notes in Computer Science*, pages 240–249. Springer, 2009.

[16] A. N. Langville and C. D. Meyer. A survey of eigenvector methods for web information retrieval. *The SIAM Review*, 47(1):135–161, 2005.

[17] M. Luby. A simple parallel algorithm for the maximal independent set problem. *SIAM J. Comput.*, 15(4):1036–1055, 1986.

[18] K. Madduri, D. A. Bader, J. W. Berry, and J. R. Crobak. An experimental study of a parallel shortest path algorithm for solving large-scale graph instances. In *ALENEX*. SIAM, 2007.

[19] U. Meyer and P. Sanders. Delta-stepping: A parallel single source shortest path algorithm. In *In ESA '98: Proceedings of the 6th Annual European Symposium on Algorithms*, pages 393–404. Springer-Verlag, 1998.

[20] T. Tu, C. A. Rendleman, D. W. Borhani, R. O. Dror, J. Gullingsrud, M. O. Jensen, J. L. Klepeis, P. Maragakis, P. Miller, K. A. Stafford, and D. E. Shaw. A scalable parallel framework for analyzing terascale molecular dynamics simulation trajectories. In *SC '08: Proceedings of the 2008 ACM/IEEE conference on Supercomputing*, pages 1–12, Piscataway, NJ, USA, 2008. IEEE Press.

Input:

Pre-aggregated MapReduce object $A = |V| \times |V|$ matrix: Key = i, Value = $[j, A_{ij}]$
Pre-aggregated MapReduce object MT = indices of all-zero rows of A: Key = i, Value = NULL
Pre-aggregated MapReduce object x = initial PageRank vector: Key = i, Value = $1/|V|$
MapReduce object y = vector, initially empty

while ($residual > tolerance$)

1 Compute contribution for random jumps and zero-outdegree vertices
  Add x to MT
* Convert MT: Key is vertex i
  Reduce MT:
    Input: Key = i, MultiValue = $(x_i, NULL)$ if $i \in MT$ else MultiValue = x_i
    If $nvalues = 2$: $c_{dot} = c_{dot} + x_i$
    Output: Key = i, Value = NULL if $nvalues = 2$
  c_{dot} = MPI_Allreduce(c_{dot},MPI_SUM)
2 Compute $y = A^T x$
  Copy x to y
  Add A to y
* Convert y: Key is row index of A
  Reduce y:
    Input: Key = i, Multivalue = $y_i, [j, A_{ij}]$ for nonzeros A_{ij} in A
    Output: Key = j, Value = $A_{ij} y_i$
  Collate y: Key is column index of A
  Reduce y:
    Input: Key = j, Multivalue = $A_{ij} y_i$ for nonzeroes A_{ij} in column j of A
    Output: Key = j, Value = $\sum_i A_{ij} y_i$
3 Add c_{dot} to y; compute y_{max} = max element in y
  Map y:
    Input: Key = i, Value = y_i
    $y_i = y_i + c_{dot}$; $y_{max} = \max(y_i, y_{max})$
    Output: Key = i, Value = y_i
  y_{max} = MPI_Allreduce(y_{max},MPI_MAX)
4 Scale y by y_{max}
  Map y:
    Input: Key = i, Value = y_i
    Output: Key = i, Value = y_i / y_{max}
5 Compute residual
  Add y to x
* Convert x: Key is vertex i
  Reduce x:
    Input: Key = i, Multivalue = x_i, y_i
    $residual = \max(|x_i - y_i|, residual)$
  $residual$ = MPI_Allreduce($residual$,MPI_MAX)
6 Copy y to x

Output:

MapReduce object x = PageRank vector: Key = i, Value = x_i = numeric rank

Figure 4: *MapReduce algorithm for PageRank vertex ranking, using 4 MapReduce objects A, MT, x, and y. Pre-aggregating several of the MapReduce objects allows several communication intensive collate() operations to be replaced by local convert() operations (marked with asterisks).*

Input:
MapReduce object G = edges of graph: Key = (V_i, V_j) with $V_i < V_j$, Value = NULL
MapReduce object G_0 = edges of graph, initially empty

1 Copy G to G_0
1 Map G: Convert edges to vertices
Input: Key = (V_i, V_j), Value = NULL
Output: Key = V_i, Value = V_j
Output: Key = V_j, Value = V_i
1 Collate G
1 Reduce G: Add first degree to one vertex in edge
Input: Key = V_i, MultiValue = $(V_j, V_k, ...)$
for each V in MultiValue:
if $V_i < V$: Output: Key = (V_i, V), Value = $(D_i, 0)$
else: Output: Key = (V, V_i), Value = $(0, D_i)$
2 Collate G
2 Reduce G: Add second degree to other vertex in edge
Input: Key = (V_i, V_j), MultiValue = $((D_i, 0), (0, D_j))$
Output: Key = (V_i, V_j), Value = (D_i, D_j) with $V_i < V_j$
3 Map G: Low degree vertex emits edges
if $D_i < D_j$: Output: Key = V_i, Value = V_j
else if $D_j < D_i$: Output: Key = V_j, Value = V_i
else: Output: Key = V_i, Value = V_j
3 Collate G
3 Reduce G: Emit angles of each vertex
Input: Key = V_i, MultiValue = $(V_j, V_k, ...)$
for each V_1 in MultiValue:
for each V_2 beyond V_1 in MultiValue:
if $V_1 < V_2$: Output: Key = (V_1, V_2), Value = V_i
else: Output: Key = (V_2, V_1), Value = V_i
4 Add G_0 to G
4 Collate G
4 Reduce G: Emit triangles
Input: Key = (V_i, V_j), MultiValue = $(V_k, V_l, NULL, V_m, ...)$
if NULL exists in MultiValue:
for each non-NULL V in MultiValue:
Output: Key = (V_i, V_j, V), Value = NULL

Output:
MapReduce object G = triangles in graph: Key = (V_i, V_j, V_k), Value = NULL

Figure 5: *MapReduce algorithm for triangle enumeration, using 2 MapReduce objects G and G_0*

Input:
MapReduce object E = edges of graph: Key = (V_i, V_j), Value = NULL
MapReduce object V = zone assignment of each vertex: Key = V_i, Value = Z_i
MapReduce object W = workspace
Iterate:
1 Map W using E as input: Convert edges to vertices
Input: Key = $E_{ij} = (V_i, V_j)$, Value = NULL
Output: Key = V_i, Value = E_{ij}
Output: Key = V_j, Value = E_{ij}
1 Add V to W
1 Collate W: Vertex as key
1 Reduce W: Emit edges of each vertex with zone of vertex
Input: Key = V_i, MultiValue = $EEEE...Z$
for each E in MultiValue:
Output: Key = E_{ij}, Value = Z_i
2 Collate W: Edge as key
2 Reduce W: Emit zone re-assignments
Input: Key = E_{ij}, MultiValue = Z_iZ_j
Z_{winner} = min(Z_i,Z_j); Z_{loser} = max(Z_i,Z_j)
if Z_i and Z_j are different:
Output: Key = Z_{loser}, Value = Z_{winner}
2 Exit: if W is empty (no output by Reduce 2)
3 Map V: Invert vertex/zone pairs
Input: Key = V_i, Value = Z_i
if Z_i is not partitioned:
Output: Key = Z_i, Value = V_i
else:
Output: Key = Z_i^+, Value = V_i for a random processor
4 Map V using W as input: Add zone reassignments in W to V
if Z_i is not partitioned:
Output: Key = Z_i, Value = Z_{winner}
else:
Output: Key = Z_i^+, Value = Z_{winner} for every processor
Output: Key = Z_i, Value = Z_{winner}
4 Collate V: Zone ID as key
4 Reduce V: Emit new zone assignment of each vertex
Input: Key = Z_i or Z_i^+, MultiValue = $VVVV...ZZZ...$
Z_{new} = min(Z_i or Z_i^+,Z,Z,Z,...)
partition Z_{new} if number of V > threshhold
for each V in MultiValue:
Output: Key = V_i, Value = Z_{new}

Output:
MapReduce object V = zone assignment of each vertex: Key = V_i, Value = Z_i

Figure 6: *MapReduce algorithm for connected component labeling, using 3 MapReduce objects E, V, and W.*

Input:
    MapReduce object V = maximal independent set vertices, initially empty
    MapReduce object E = edges of graph: Key = (V_i, V_j) with $V_i < V_j$, Value = NULL
Map E: Assign random values consistently to each vertex V_i and V_j in E
Clone E: Convert KV pairs in E directly to KMV pairs with multivalue = NULL

while N_{edges} in $E > 0$:
    1 Reduce E: Determine WINNER/LOSER vertex of each edge
        Input: Key = E_{ij}, Multivalue = NULL or LOSER
        if multivalue = LOSER, emit nothing
        V_w = WINNER vertex, V_l = LOSER vertex
        Output: Key = V_w, Value = (V_l, WINNER)
        Output: Key = V_l, Value = (V_w, LOSER)
    1 Collate E: Vertex as key
    2 Reduce E: Find WINNER vertices
        Input: Key = V_i, MultiValue = $VVVV$...
        if all V are flagged with WINNER, V_i is a WINNER
            for each V_j in Multivalue:
                Output: Key = V_j, Value = (V_i, WINNER)
        else:
            for each V_j in Multivalue:
                Output: Key = V_j, Value = V_i
    2 Collate E: Vertex as key
    3 Reduce E: Find LOSER vertices
        Input: Key = V_i, MultiValue = $VVVV$...
        if any V is flagged with WINNER, V_i is a LOSER
            for each V_j in Multivalue:
                Output: Key = V_j, Value = (V_i, LOSER)
        else:
            for each V_j in Multivalue:
                Output: Key = V_j, Value = V_i
    3 Collate E: Vertex as key
    4 Reduce E: Emit WINNER vertices and LOSER edges
        Input: Key = V_i, MultiValue = $VVVV$...
        if all V are flagged with LOSER, V_i is a WINNER
            Output: to V: Key = V_i, Value = NULL
        for each V_j in Multivalue:
            if V_j is flagged with LOSER:
                Output: to E: Key = E_{ij} with $V_i < V_j$, Value = LOSER
            else:
                Output: to E: Key = E_{ij} with $V_i < V_j$, Value = NULL
    4 Collate E: Edge as key

Output:
    MapReduce object V = maximal independent set vertices: Key = V_i, Value = NULL

Figure 7: *MapReduce algorithm for finding a maximal independent set of graph vertices via Luby's algorithm, using two MapReduce objects E and V.*

Input:
MapReduce object E = graph edges with weights: Key = (V_i, V_j), Value = w_{ij}
MapReduce object V = vertices with distances from source vertex V_s:
Key = V_i, Value = ∞, except Value for $V_s = 0$

while not *done*:
done = 1
Map V using E as input: Add edge weights in E to V
Input: Key = E_{ij}, Value = w_{ij}
Output: Key = V_i, Value = (V_j, w_{ij})
Collate V: Vertex is key
Reduce V: Loop over edges of vertex to update distance to V_s
Input: Key = V_i, Multivalue = d_i, edges (V_j, w_{ij}), candidate distances d_j
$d_{min} = \min(d_i, d_j, d_j, \ldots)$
If $d_{min} \neq d_i$:
done = 0
Output: Key = V_j, Value = $d_{min} + w_{ij}$ for each edge vertex V_j
Output: Key = V_i, Value = d_{min}
done = MPI_Allreduce(*done*,MPI_MIN)

Output:
MapReduce object V = vertices with distances from source vertex V_s:
Key = V_i, Value = d_i

Figure 8: *MapReduce algorithm for single-source shortest path (SSSP) calculation, using two MapReduce objects V and E.*

Input:
    Pre-aggregated MapReduce object E = graph edges with weights: Key = (V_i, V_j), Value = w_{ij}
    Pre-aggregated MapReduce object V = vertices with distances from source vertex V_s:
        Key = V_i, Value = ∞, except Value for $V_s = 0$
    MapReduce object U = updates to distances, initially empty

while not *done*:
    done = 1
    Aggregate U
    Add U to V, U = empty
    Compress V: Compute best distance for each vertex
        Input: Key = V_i, Multivalue = d_i, candidate distances d_j
        $d_{min} = \min(d_i, d_j, d_j, \ldots)$
        If $d_{min} \neq d_i$:
            done = 0
            Output to U: Key = V_i, Value = d_{min}
        Output to V: Key = V_i, Value = d_{min}
    done = MPI_Allreduce(*done*,MPI_MIN)
    if not *done*:
        Add U to E, U = empty
        Compress E: Generate candidate distances for neighbors of changed vertices.
            Input: Key = V_i, Multivalue = edges (V_j, w_{ij}) and d_i if updated in previous step
            If updated d_i exists:
                Output to U: Key = V_j, Value = $d_i + w_{ij}$ for each edge vertex V_j

Output:
    MapReduce object V = vertices with distances from source vertex V_s:
        Key = V_i, Value = d_i

Figure 9: *Enhanced MapReduce algorithm for single-source shortest path (SSSP) calculation, with an additional MapReduce object U. Pre-aggregating the vertices V and edges E, and storing only changed distances in U, reduces communication and execution time.*

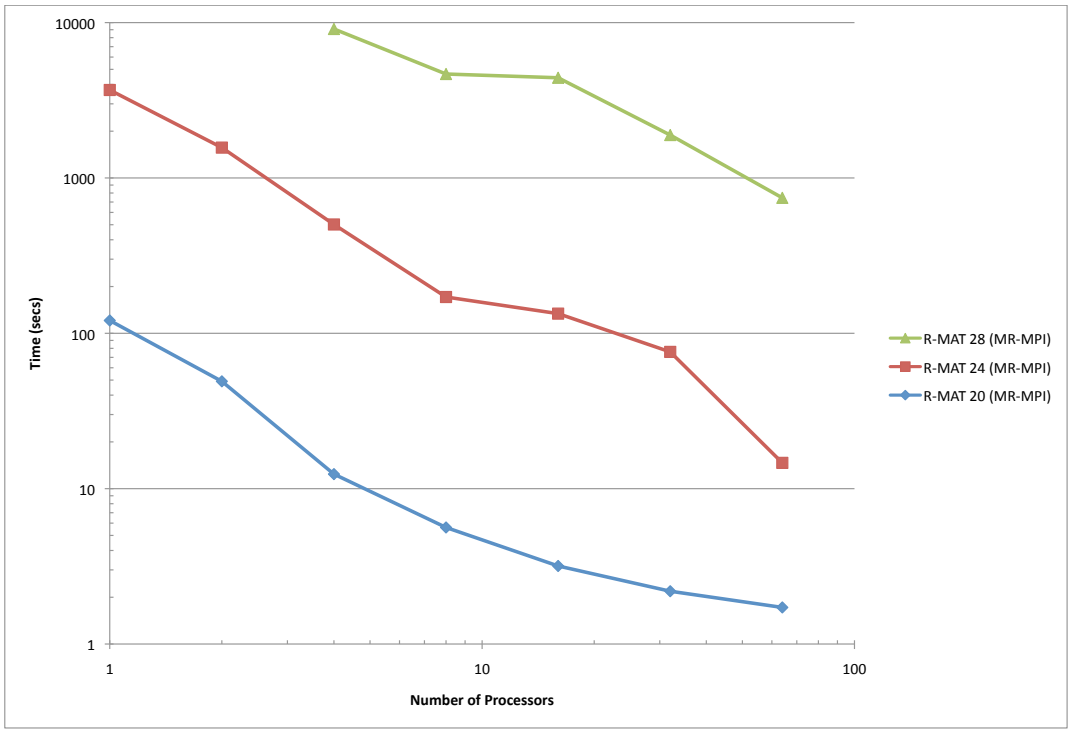

Figure 10: Performance of the MR-MPI R-MAT generation algorithm (Figure 3)

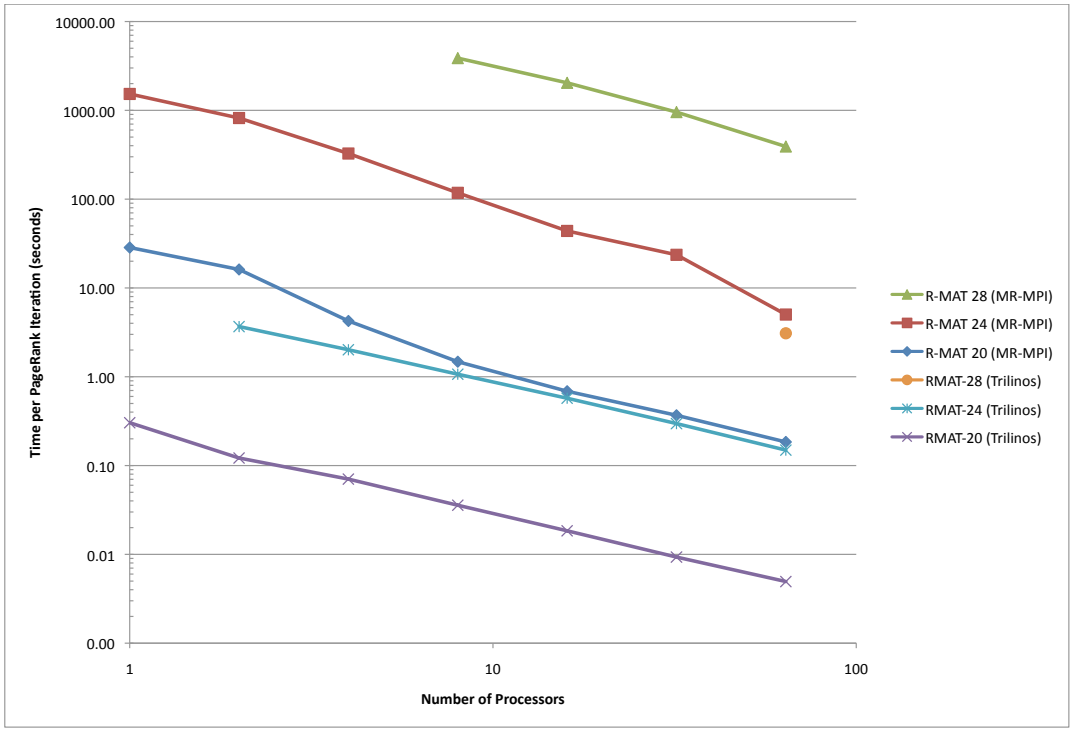

Figure 11: Comparison of PageRank implementations using MR-MPI (Figure 4) and Trilinos' matrix/vector classes on R-MAT data sets.

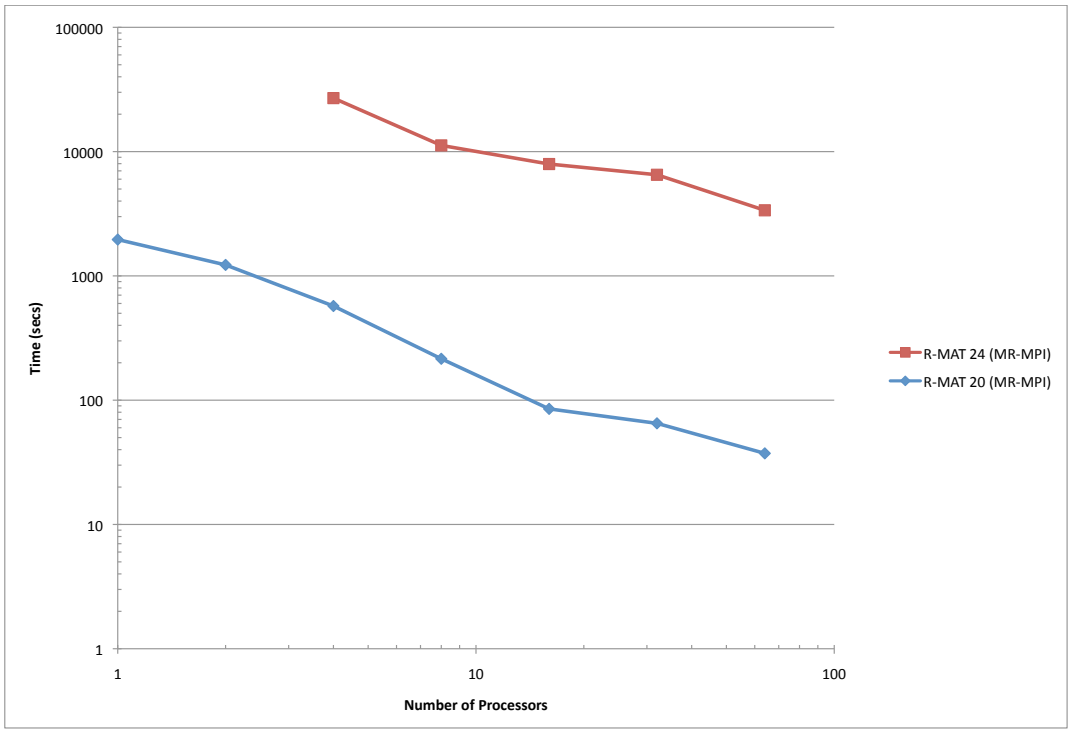

Figure 12: Performance of the MR-MPI triangle-finding algorithm (Figure 5).

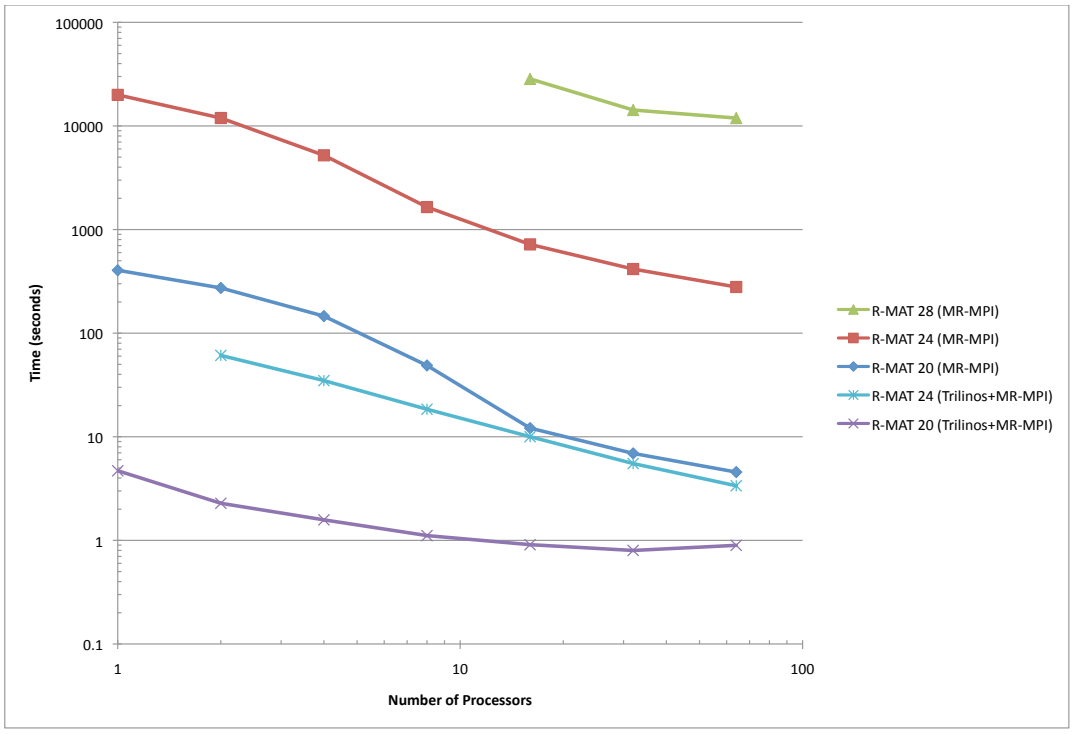

Figure 13: Performance of the MR-MPI connected components algorithm (Figure 6) compared with a hybrid "Giant Connected Component" algorithm based on Trilinos.

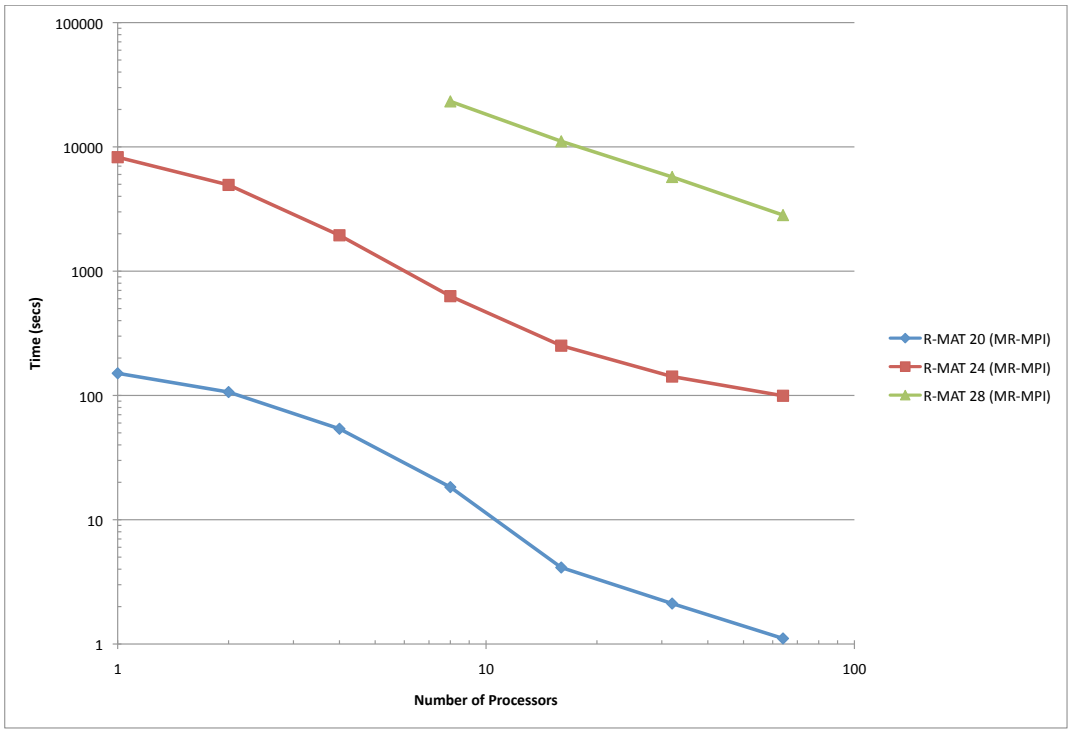

Figure 14: Performance of the MR-MPI maximal independent set algorithm (Figure 7).

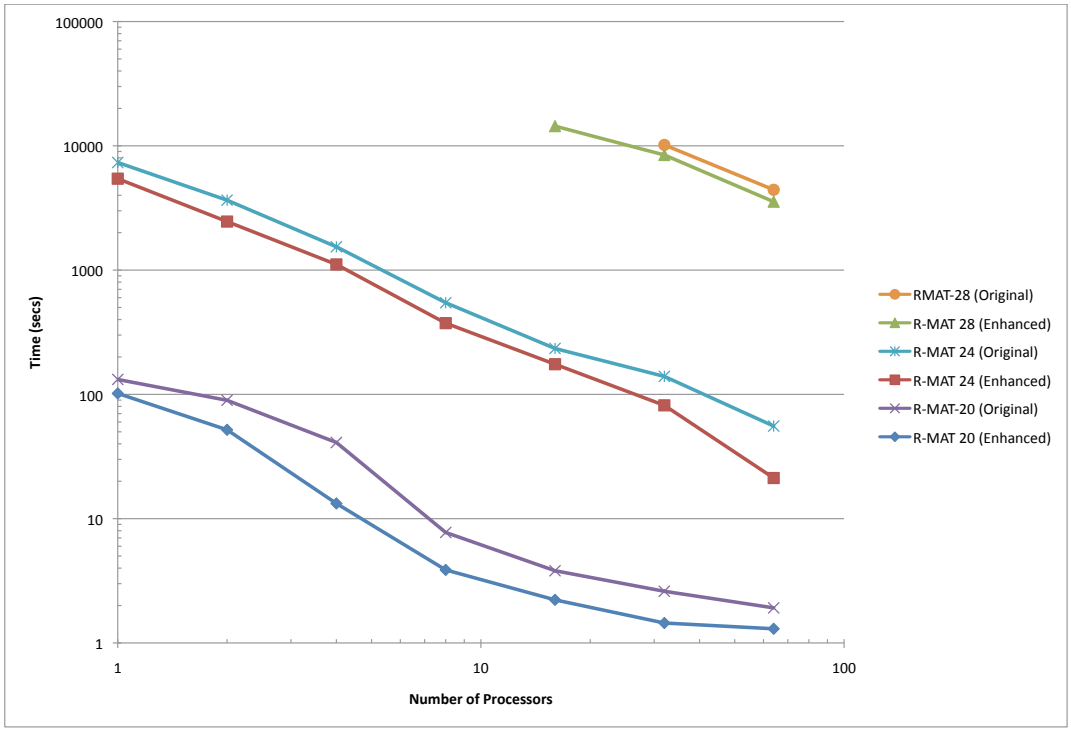

Figure 15: Execution times for SSSP using MR-MPI with R-MAT matrices. Both the original algorithm (Figure 8) and the enhanced algorithm (Figure 9) are included.

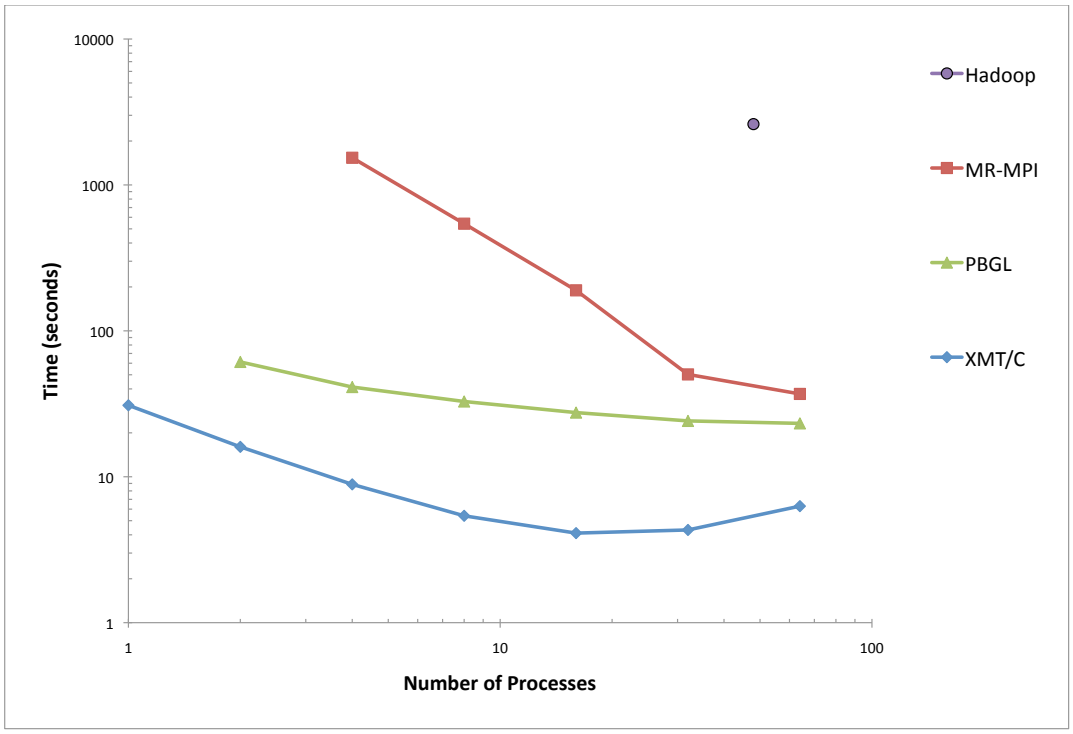

Figure 16: SSSP using MR-MPI for WebGraphA with 13.2M vertices and 31.9M edges. Runtimes using Hadoop and PBGL are also shown.

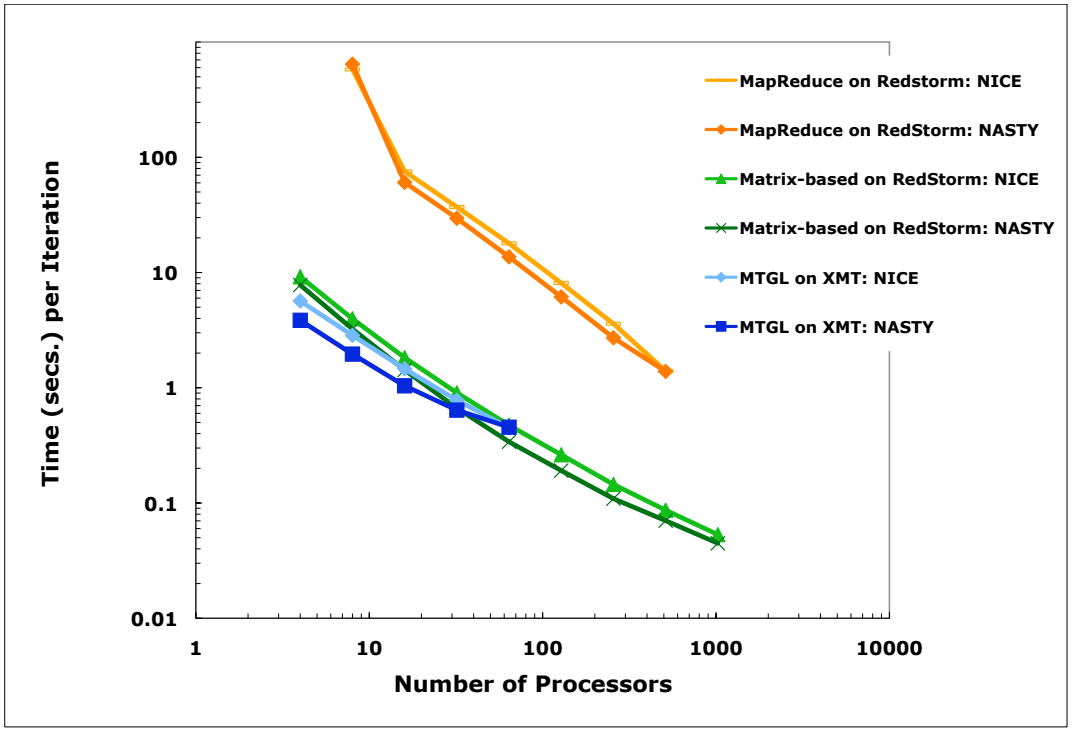

Figure 17: Scalability comparison of PageRank using MapReduce (MR-MPI), matrix-based (Trilinos), and multi-threaded (MTGL) implementation on the R-MAT data sets in Table 6.

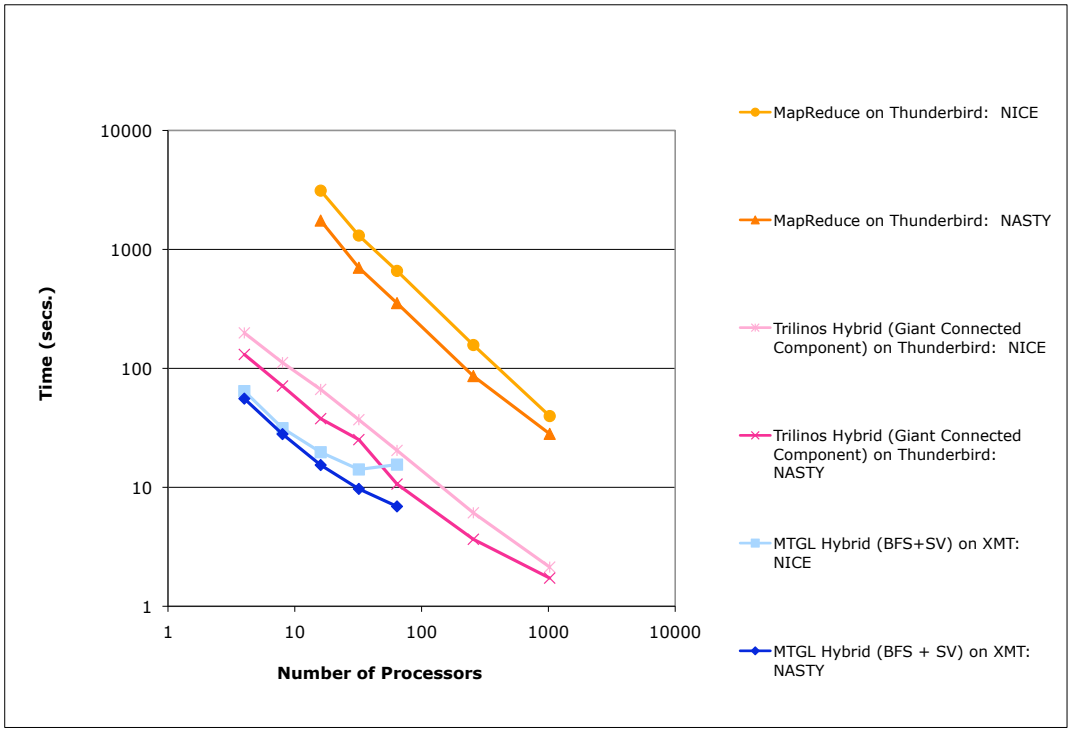

Figure 18: Scalability comparison of Connected Components algorithms using MapReduce (MR-MPI), matrix-based/MapReduce hybrid (Trilinos/MR-MPI), and MTGL implementations on the R-MAT data sets in Table 6.