MapReduce in MPI for Large-scale Graph Algorithms

Steven J. Plimpton, Karen D. Devine, Others?
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 which 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 [**?**]. 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 [**?**].

Similarly, the open-source Hadoop implementation of MapReduce [**?**], 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, though 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 [**?**], graph algorithmics [**?**], and linear algebra operations [**?**]. The paper by Tu et al [**?**] 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 on larger data sets. These algorithmic improvements are described in this paper and are part of the current downloadable version [**?**].

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 synncrhonous, 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 processor pairs using large aggregated messages to improve bandwidth performance and reduce latency costs. A recent paper by [**?**] also outlines the MapReduce formalism from an MPI perspective, though they advocate a more asyncrhonous 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. For serial operation, a dummy MPI library (provided) can be substituted. As a library, it can be embedded in other codes [**?**, **?**] to enable them to perform MapReduces.

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

- *Flexible programmability:* An advantage of writing a MapReduce program on top of MPI, is that the user program can 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 back to all callback functions, so it is easy for the user program to store "state" on each processor, accessible during the map and reduce operaions. For example, various flags can be stored that alter the operation of a map or reduce function, as can richer data structures, that accumulate the results.

- *C++, C, and Python interfaces:* A C++ interface to the MR-MPI library means a user program instantiates and then invokes methods in one or more MapReduce objects. A C interface means the library can also be called from C or other hi-level languages such as Fortran. A C interface also means the library can 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: 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 HDFS file system which provides for data redundancy, the MR-MPI library simply 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. The next two 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 1 million to 1 trillion vertices or edges. In this section, 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 **??** 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. A related data type 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 integer lengths, one per value.

A typical MR-MPI program makes at least 3 calls to the MR-MPI library, to perform a *map()*, *collate()*, and *reduce()* operation. In a *map()*, zero or more key/value pairs are generated by each processor. Typically, this is done using data read in 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, or a task ID. Options for assigning map tasks to processors.

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 the processor who "owns" it. The N-byte length key is hashed into a 32-bit value whose remainder modulo P generates the owning processor ID. Alternatively, the user can provide a hash function which converts the key into a processor ID. 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 requires $\log(N)$ passes through the N KV pairs. the list of KMV pairs can be created in 2 passes through the KV pairs, one to populate the hash table with needed count and length information, and the second pass to copy the key and value datums into the appropriate location in a new KMV data structure. Since the lookup of a key in a well-formed hash table is a constant-time $O(1)$ operation, the cost of the data reorganization is $O(N)$, rather than $N \log(N)$ for full sort. It also has the added benefit that the number of values in each KMV pair is known, which is passed to the user function during a reduce. For some reduce operations, this is all the information a reduce requires; thus the values need not be looped over. This is not the case for a Hadoop-style data reorg which sorts the values. It does not know a priori how many values are associated with the same key without iterating over them.

Note that the first portion of the *collate()* operation involves all-to-all communication (each processor sends and receives data from every other processor) driven by a distributed hash table. The communication can either be done via a MPI_Alltoall() library call, or by a custom routine that aggregates messages and invokes pointwise MPI_Send() and MPI_IRecv() calls.

The *reduce()* operation processes KMV pairs and can produce new KV pairs for continued computation. 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 *compress()* operation combines on-processor KV pairs to eliminate duplicate keys, forming new KV pairs with

an aggregated value. The *convert()* operation turns a list of KV pairs into KMV pairs, with one value per key. The collapse() operation 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 on the KMV. The *gather()* operation collects KV pairs from all processors to a subset of processors; it is useful for doing output from a single or subset of processors. Library calls for sorting datums by key or value or within multivalues 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 one-by-one into a new data structure in sorted order.

The purpose of providing an interface to various lo-level operations, is that a user program can string them together in various ways to produce interesting MapReduce algorithms. For example, output from a *reduce()* can serve as input to a subsequent *map()* or *collate()*. KV pairs from mutiple MapReduce objects can be combined together 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. If this is the case, then we refer to this as "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 Tbytes, 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 randomized hashing to assign keys to processors is designed for such load-balancing. What happens when data sets do not fit in available memory, is the subject of the next section.

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 oen or more processors do not fit in its 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 NSF-mounted file system for a parallel machine), or on a parallel file system/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 insure the allocated pages fit in physical memory, else the MR-MPI library 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* insures 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 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, then those datums are read in, 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, passes a pointer, one at a time, to each pair to the user function which 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 will then be 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 datums are communicated to new processors in an all-to-all fashion. In out-of-core mode, this is done one page of KV data at a time. Each processor reads in a page of KV pairs, the communication pattern for the datums in that page is determined (who needs what datums), 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, then 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 datums, the total number of memory pages required to perform the all-to-all communication, including the source and target KV pages is at most 7.

When the communication stage of the *collate()* operation is complete, each processor now 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, also stored in new out-of-core KMV file. In principle, to create one page of KMV pairs, would require all KV pages be scanned, to find all the keys which 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 which 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 4 times, and writes out new KV or KMV pages at most 3 times. It also uses a finite-size in-memory hash table, which 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 will (likely) fit in the hash table (HT). Initially, all KV pairs are assigned to the first partition file as the 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, then 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 in, they are assigned to the original partition file if the KV pair's key is already in the current HT. If not, then 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.

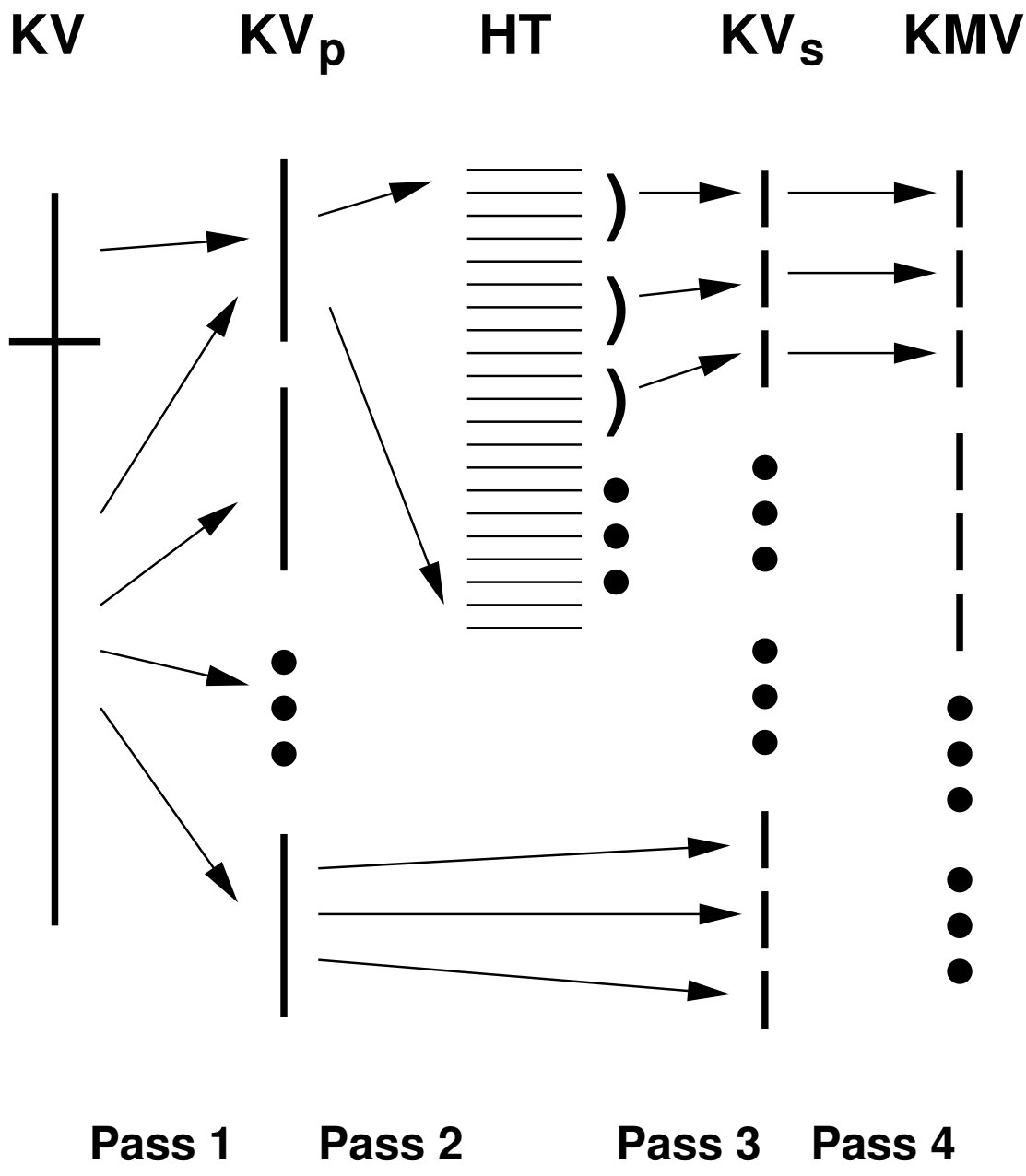

Figure 1: Multi-pass algorithm for converting KV data to KMV data. The vertical lines represent out-of-core data sets. KV is key/value pairs; HT is an in-memory hash table; KMV is key/multivalue pairs.

- (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. The associated values may create KMV pairs which will 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 which will contribute to the KMV pairs that will populate one page of KMV output. If a single KMV pair spans multiple pages, because it contains a very large number of values, then the corresponding KV pairs will still be 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 4 times, and written to disk 3 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. how many unique keys it contains), 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, requires $O(\log M)$ read/write passes through the KV data, where M is the number of pages of KV pairs, which can be a large number for big data sets.

The memory page requirements for the above algorithm 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 memory suffice for all needed buffers. Thus the total pages required for the data reorganization stage of the *collate()* operation is less than the 7 used by the all-to-all communication stage.

Other MR-MPI library calls discussed in the previous section, such as *compress()*, *convert()*, *collapse()*, *gather()*, etc, 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 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 5 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 5 pages. The number of read/write passes through the KV data set for the merge sort is $O(\log M)$, where M is the number of pages of KV pairs.

4 Graph Algoriths 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 graphs [**?**] are recursively generated graphs with power-law degree distributions. They are commonly used to represent web and social networks. The user specifices 6 parameters which define the graph: the number of vertices N and edges M, and 4 parameters a, b, c, d which sum to 1.0 and are discussed below. The algorithm in Figure 7 generates $N_z = MN$ unique non-zero entries in a sparse NxN matrix A, where each entry A_{ij} represents an edge between graph vertices (V_i, V_j).

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

Figure 2: MapReduce algorithm for R-MAT graph generation.

In the map() operation, each of P processors generates a $1/P$ fraction of the desired edges. A single random i, j edge is computed recursively as follows. Pick a random quadrant of the A 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 non-zero matrix element A_{ij}.

The map() will often generate some small number of duplicate edges. The collate() and reduce() operations remove the duplicates. The entire map-collate-reduce sequence is repeated until the number of resulting key/value pairs N_{kv} equals N_z. 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 3, a highly skewed distribution results. Variants of the above algorithm can be used when N is not a power-of-two, to generate graphs with weighted edges (assign a numeric value to the A_{ij} edge), graphs without self edges (require $i! = j$), 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.

The PageRank algorithm assings a relative numeric rank to each vertex in a graph.

It models the web as a directed graph $G(V, E)$, with each vertex $v \in V$ representing a web page and each edge $e_{ij} \in E$ 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), $d_{out}(v)$ is the outdegree of vertex v, 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 $\alpha/|V|$, indicating equal likelihood to move to any other page. Equivalently, the graph can be represented by a matrix A [**?**], with matrix entries $A_{ij} = \alpha/d_{out}(v_i)$ if vertex v_i links to v_j. The PageRank algorithm, then, is simply a power-method iteration in which the dominating computation is matrix-vector multiplication $A^T x = y$, where x is the PageRank vector from the previous iteration.

The algorithm in Figure 3 performs these iterations.

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

Figure 3: MapReduce algorithm for PageRank vertex ranking.

The MapReduce implementation performs two *map()* operations to initialize the graph matrix A and PageRank vector x. A *collate()* operation gathers all row entries a_{ij} with their associated x_i entry, and a *reduce()* computes $a_{ij}x_i$. A second *collate()* gathers, for each j, all contributions to the column sum $\sum a_{ij}x_i$, which is computed by a second *reduce()*. MPI_Allreduce calls are used to compute global norms and residuals.

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 4 outlines a MapReduce algorithm that enumerates all triangles, 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. This exposition follows the triangle-finding algorithm presented in [**?**].

The initial step is to store a copy of the graph edges as key/value (KV) pairs in a auxiliary MapReduce object G_0, for use later in the algorithm. The first *map()* operation converts edge keys to vertex keys with edge values. After the *collate()*, each vertex has a 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 key/value (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 volume of KV pairs emitted at this stage is minimized by having only the low-degree vertex in each edge generate angles.

In stage 4, the KV pairs in the original graph G_0 are added to the current working set of KV pairs. The KV pairs in G_0 are 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 which are complete triangles and can be emitted as a triplet key.

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 5 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 a vertex in the component. We assume an input graph of undirected edges (V_i, V_j). This exposition also follows the connected-component algorithm presented in [**?**], with the addition of logic that load-balances data across processors 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 $Z_i = V_i$. Each iteration will grow 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 will have become a fully connected component. 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()* operation emits the vertices in each edge as keys, with the edge as a value. The current zone assignment of each vertex is added to the set of key/value pairs. The first collate() operation collects all the edges of a vertex and its zone assignment together in one multi-value. The first reduce operation then re-emits each edge, tagged by the zone assignment of one of its vertices.

Since each edge was emitted twice, the second collate operation collects the zone assignments for its two vertices together. If the two zone IDs are different, the second reduce operation chooses a winner (the min of the 2 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 iteration exits.

The third map() operation inverts the vertex/zone key/value pairs to become zone/vertex pairs. The third add operation adds the changing zone assignments to the set of key/value pairs. The third collate() can then collects all the vertices of a zone and zero or more reassignments for the zone ID. Since a zone could collide with multiple other zones on the same iteration due to shared edges, the new zone ID becomes the minimum ID of any of the neighboring zones. If no zone reassignment value appears in the multi-value, the zone ID is unchanged. The final reduce() a key/value pair for each vertex in the zone, with the vertex as a key and the new zone ID as a value.

Note that if a graph has only a few components, then the third collate operation, which keys on the zone ID, may generate a few very large key/multi-value (KMV) pairs. For example, if the graph is fully connected, then on the last iteration, a single KMV pair will contain all vertices in the graph and be assigned to one processor. This imbalance in memory and computational work can lead to poor parallel performance of the overall algorithm. To counter this effect, the various operations of stage 3 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() operation, if the zone has been partitioned, then 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 5. Likewise, if Z_i has been partitioned, the third add() operation emits the zone ID change key/value pair (Z_i,Z_{winner}) 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).

This additional partitioning logic means that the third collate() operation, which keys on zone IDs, some of which now include processor bits, will collect only a $1/P$ subset of the vertices in large zones onto each processor. But the multivalue on each processor contains all the zone-reassignments relevant to the unpartitioned zone. This allows the third reduce operation to change the zone ID (if necessary) in a consistent manner across all P multi-values that contain the zone's vertices. When the reduce() operation emits new zone assignments for each vertex, the zone retains its partitioned status, and 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 will be 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 will 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 third add() operation emits $P+1$ copies of new zone reassignments for both partitioned and unpartioned zone IDs, to insure all vertices in the zone will

know the reassignment information.

Discussion of Luby algorithm for maximally independent sets.

footnote on NP-complete for maximal; MapReduce is unlikely to help with that issue.

Discussion of SSSP.

Discuss new optimization in SSSP?

5 Performance Results

In this section, we present performance results for the MapReduce 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. Describe Odin.

We ran each of the algorithms on 3 artificial R-MAT graphs of different sizes, each on a varying number of processors. The *small* problem size (around 1M edges) was a problem that could be run on a single processor. The *medium* problem size (around 1B edges) could be run on a few processors. The *large* problem size (around 1T edges) required most of the machine to run.

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.

Discuss R-MAT generation times.

Discuss PageRank performance. Include Trilinos and other algorithms and machines?

Discuss triangle finding times.

Discuss connected component identification times.

Discuss Luby maximally independent sets times.

Discuss SSSP times.

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 shuffle) with communication between all pairs of processors. But the benefit is often good load-balance, even for hard-to-balance irregular data sets. By contrast, more traditional distributed-memory parallel algorithms, e.g. for matrix operations, or grid-based or particle-based simulation codes, tend to work hard to localize data and minimize communication. To do this they typically require a lot of application-specific logic and parallel communication coding, to create and maintain a data decomposition, generate ghost versions 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 than 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 actually needing to write application-specific parallel code to move data via MPI calls, is often a pleasant surprise. Moreover, if the 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 that of cloud-computing MapReduces 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 somehere in the cloud, and Hadoop ensures data moves where it needs to 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, at least 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 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 centric. We hope the library can be generally useful on large-scale monolithic or cloud-style parallel machines which support MPI, for a variety of data-intense or compute-intense problems that are amenable to solution using a MapReduce paradigm.

7 Acknowledgements

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 (NSA) for fruitful discussions about his MapReduce graph algorithms [**?**]; 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.

1 Copy: G_0 = copy of edge KV pairs from input graph

1 Map: 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

1 Reduce: 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

2 Reduce: 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: 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

3 Reduce: 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: Add G_0 edge KV pairs to angle KV pairs

4 Collate

4 Reduce: 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

Figure 4: MapReduce algorithm for triangle enumeration.

Iterate:

- 1 Map: Convert edges to vertices
  - input Key = E_{ij}, Value = NULL
  - output Key = V_i, Value = E_{ij}
  - output Key = V_j, Value = E_{ij}
- 1 Add: Zone assignment of each vertex
  - output Key = V_i, Value = Z_i
- 1 Collate: Vertex as key
- 1 Reduce: 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: Edge as key
- 2 Reduce: 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 no output by Reduce 2
- 3 Map: 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
- 3 Add: Changed zones (Z_i,Z_{winner})
  - 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}
- 3 Collate: Zone ID as key
- 3 Reduce: 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}

Figure 5: MapReduce algorithm for connected component labelling.

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

Figure 6: MapReduce algorithm for R-MAT graog generation.

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

Figure 7: MapReduce algorithm for single-source shortest path (SSSP) determination.