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, RMAT 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 RMAT 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.

- *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

keys and values, KV and KMV data structures map and reduce are naturally parallel on distributed KV, KMV user owns data, just set of bytes passed to library flavors of map() collate is equivalant to all2all() on distributed hash table how we do irregular communication big difference is we are doing the comm in syncronous manner data is reorganized locally, once it is all on-proc other related operations: compress, gather, collapse, sort

3 Out-of-core Issues

what part needs to be out-of-core in MPI context each proc does it's own disk IO basic idea: modified data structures for KV and KMV map() operation aggreate() operation convert() operation reduce() operation

4 Graph Algoriths in MapReduce

RMAT generation, pagerank, CC, triangle-finding, Luby, SSSP, other? how many variants of something like CC do we want to include? I would vote for just one discuss possible inefficiency due to a giant CC in CC algorithm

5 Performance Results

ideally, for each problem: define chosen characteristics of graph, e.g. edges/vertex choose 3 sizes: small = fits on 1 proc) medium = out-of-core on 1 proc, but not on P procs huge = impressive size show performance plots for each size running on 1 to P procs would like to see kink in plots as go out-of-core be able to judge and explain basic scaling results what machine: Odin or Tbird or RStorm? something that maps to a vanilla cluster where we have data: show multiple machines alternate non-MR algorithms discuss alternate CC algorithms via Trilinos and hybrid alternate MR implementations (Hadoop)

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.