Performance Competitiveness of MapReduce Applications in Scientific Research
Literature Review and Related Sections

Daniel Washburn

December 15, 2016

Contents

1 Background 2
  1.1 Motivation ................................................................. 3

2 Literature Review 4
  2.1 K-means clustering ......................................................... 4
  2.2 SNP Genotyping .......................................................... 5
  2.3 Multidimensional Scaling ............................................. 7
  2.4 PageRank ................................................................. 9

3 Summary and Analysis .................................................. 10

4 Open Questions .......................................................... 14
1 Background

MapReduce is a model of parallel computing on computing clusters which has accumulated a large following and a substantial amount of hype in both industry and academia over the past several, and particularly the last few, years. First developed in 2004 by Google [DG04], MapReduce uses a simplified computational model that can nevertheless express many algorithms to hide from the user the complications of adapting an algorithm to parallel computation over potentially very large clusters while maintaining scalability. In this model, the programmer writes a map function which processes a series of key/value pairs and outputs another series of key/value pairs, usually modified in some manner from the originals, and a reduce function which takes the key/value pairs emitted by the map function, aggregated by key, and processes and merges them according to the specification given by the programmer. This model of computation, while apparently very simple and limited, is fairly expressive, though it is certainly not without limitations. The simplified model allows the runtime, or backend, of the MapReduce system to handle partitioning, scheduling of execution, fault-tolerance (for example, handling machine failure in the cluster), and communication between the various processes and machines. MapReduce makes it fairly easy for programmers new to parallel computing to effectively use at-scale clusters for a wide variety of applications and with a low learning curve, because the runtime manages all of these details, and because the model of computation is simple and fairly flexible [DG04] [LLC+12] [DN14]. Initially MapReduce referred only to Google’s original proprietary implementation. However, many other implementations have since been created, also called MapReduce systems, or sometimes MapReduce-style systems. The most prominent of these is Hadoop, an open-source implementation of the MapReduce paradigm and a major Apache project [LLC+12] widely used in both industry and academia.

Fault-tolerance is built into the distributed operating system and the runtime model. The filesystem (Google File System (GFS) in the original, Hadoop DFS (HDFS) in Hadoop, for example) splits the data into large chunks and manages three copies of each on separate machines. In this way, the chance of data loss from sudden machine failure is low. The filesystem is optimized for the MapReduce parallel computing paradigm and large clusters of commodity machines in a variety of other ways as well [LLC+12]. The runtime maintains a master node, which manages everything, and many worker nodes. The input data, like the filesystem itself, is chunked, and the different chunks are assigned to separate map or reduce tasks. Each task handles only its own data so it need not communicate with other tasks or nodes, except to receive its task, to return its data when finished, and to communicate with the master [LLC+12]. The worker nodes report back regularly to the master node when processing a map or reduce task, and if one of them fails (or is simply taking much longer to complete its task than the others), the master will freshly assign that task to another node, without actually canceling the original, and use whichever results come back first [LLC+12] [DN14]. This allows for a certain amount of automatic
load-balancing as well [LLC+12].

Despite the popularity of the MapReduce model (in particular, its implementation as Hadoop), and its benefits (e.g., scalability, ease of use, fault-tolerance), it has received significant criticism. Most of this criticism has to do with issues of performance/efficiency and the limited data model, data flow, and programming model [LLC+12]. More specifically, the paradigm has been compared unfavorably to DBMSs (DataBase Management Systems) for its lack of more structured data via schemas or indexes [LLC+12], for example, though it has been argued also that this not an apples-to-apples comparison and that DBMSs and MapReduce systems serve different roles. It has also been criticized for lacking a high-level language or more complicated forms of data flow (joins and n-way operations, for instance), for being poorly suited to iterative forms of computation, for excessive recomputation and lacking mechanisms for exploratory, approximate, and/or partial computation of the input, and for the cost of communication between nodes and tasks before and after the map and reduce phases [LLC+12] [DN14]. For instance, the shuffling phase, where data are merged, split up again, and assigned to the various nodes as reduce tasks, can take more time than the map and reduce phases in some circumstances where communication is particularly extensive and expensive [DN14].

1.1 Motivation

Kwon et al. note for science, and Doulkeridis and Nørvåg for industry, that due to the enormous volumes (and flows!) of data currently being generated in both fields, data is no longer a major bottleneck to innovation. Rather, the scalability and efficiency of analyzing the data are the bottleneck [KNG+10] [DN14]. As a result, the ability to transform existing algorithms into forms which can be executed on large clusters without excessive difficulty in adaptation, and to use the computing resources of these clusters in the most optimal and scalable fashion, has suddenly become an important computational challenge for a wide range of commercial and academic fields and applications. Determining whether and when a particular parallel computing paradigm is most effective, and thus diverting resources from less efficient or reliable models to more effective ones, can have very significant impacts. The rapid popularization of the MapReduce paradigm has certainly expanded significantly the number and range of actors capable of effectively utilizing the extensive computing resources of large clusters, in large part due to its ease-of-use for those lacking prior experience with HPC. However, it also needs to be critiqued so that we can know if and when it is being used inappropriately in place of a better model for a given task, and how it can be improved to handle tasks for which it does not currently perform well.
2 Literature Review

As previously mentioned, the MapReduce distributed computation model has seen significant acceptance in the scientific community and has been applied to a wide variety of scientific applications. It is these applications that we focus on, and we give a review of a sample of them, the work most relevant to this paper, below. If not clarified, I will use the term “speedup” to refer to Amdahl scaling (change in time or throughput when increasing the number of nodes and holding the input data size constant) and the term “scaleup” to refer to Gustafson scaling (change in time or throughput when increasing the number of nodes and the input data size simultaneously and proportionally). Other metrics of scalability (such as change in time when data size is changed and number of nodes held constant) will be specified directly.

2.1 K-means clustering

In [EPF08], Ekanayake et al. introduce CGL-MapReduce, a streaming implementation of the MapReduce model designed for improved performance on iterative tasks/algorithms (which they later develop into Twister in [ELZ+10]). They apply this model to two data intensive problems, one very suitable for, but novel to, MapReduce and the other naturally iterative: a high-energy physics (HEP) application which they got from astrophysics researchers they were working with, and the k-means clustering problem. They compare CGL-MapReduce to a Hadoop implementation, and for k-means clustering to an MPI implementation as well. For the HEP application they observe overall similar performance to Hadoop for CGL-MapReduce, with some modest performance and speedup improvements. The k-means algorithm is an EM (expectation maximization) algorithm, which involves the repeated evaluation and recalculation of estimates of the ideal centers. Because of this, as the authors note, “Kmeans clustering is within the class of applications where multiple iterations of MapReduce computations are necessary for the overall computation.” As a result of this heavy iteration Ekanayake et al. observe for the k-means algorithm, as expected, atrocious performance from vanilla Hadoop and excellent performance from the MPI implementation. Their CGL-MapReduce implementation is much better than Hadoop and for larger data/problem sizes attains comparable, though mildly worse, performance to MPI. Notably, however, they do not perform any experiments on speedup or scaleup for k-means, only examining scalability in the sense of considering performance for different input data sizes on the same number of nodes.

Several of the modifications made to the MapReduce model in CGL-MapReduce seem to inherently trade off fault-tolerance for performance. For example, individual machine failure can no longer be recovered from and requires a restart of the whole MapReduce job for the given algorithm iteration. Some of these modifications would also appear to de-
crease the much-vaunted ease-of-use associated with MapReduce. These are acknowledged by the authors, with some vague assurances of future improvements to fault-tolerance, particularly in the streaming system which has replaced writing intermediate results to disk. The authors do not do any empirical or analytical analysis of the degree to which their optimizations have affected reliability, and it is possible a priori that they have lost most of MapReduce’s fault-tolerance in order to gain most of MPIs performance.

Dhillon and Modha in [DM02] have another MPI implementation of k-means which seems well-received and achieved excellent, truly near-linear speedup and scaleup curves and could serve as another MPI competitor to MapReduce k-means implementations. Actually, since this paper was published in 2002 and seems to have become the seminal method for parallelizing k-means based on its extensive citation in the literature, it is possible or even likely that theirs was the method used for the MPI competitor in the CGL-MapReduce [EPF08] paper, particularly since no details of the MPI method used are provided in that paper. Ngazimbi presents MapReduce implementations of three clustering algorithms, including k-means, and applies them to the movie-recommendation problem, inspired by Netflix and its movie rating predictor competition [Nga09]. Heafield also proposes a k-means MapReduce application for Hadoop in an early Google talk from 2008 [Hea08]. These could serve as alternate or additional MapReduce k-means implementations for the purposes of performance competition and comparison.

2.2 SNP Genotyping

SNP genotype calling (also known as SNP genotyping or SNP calling) is the identification of SNPs (single nucleotide polymorphisms, mutations of just one ‘letter’ of DNA) in a given body of DNA, such as an individual’s genome, based on an existing reference genome. Much SNP calling is done using microarrays, but an approach called next-generation sequencing (NGS) which has much higher throughput (i.e., can more quickly sequence large quantities of DNA) but much shorter read lengths (the resulting data is fragmented into much smaller sections of the original DNA, and as a result it tends to more difficult to piece back together) has rapidly gained in popularity.

[LPZ12] presents a MapReduce implementation/adaptation of existing algorithms for microarray-based SNP genotype calling which has greater scalability and flexibility than the previous state-of-the-art solutions, which were either sequential, lightly parallelized in an ad-hoc fashion, or parallelized in a way that required significant user intervention to correctly partition the input data, and even with said user intervention had lower accuracy rates. Their implementation is over an order of magnitude faster than existing approaches to microarray-based SNP calling, and hence a significant improvement. They demonstrate fairly linear speedup, and a sublinear but gradual decline in throughput as input was increased while the number of nodes was held constant.
There has been much more application of the MapReduce paradigm in the field of next-gen sequencing, with promising results. Schatz presents the CloudBurst SNP calling system in [Sch09], which according to [Tay10] is a landmark paper, demonstrating the effectiveness of the MapReduce model for NGS SNP calling. CloudBurst is an adaptation of the RMAP algorithm to a parallel MapReduce-style implementation using Hadoop. Its runtime increases linearly as the input size (# of reads) increases, and it has near linear speedup (increase in throughput as the number of cores is increased) as well – 3.5 times faster with 4 times as many cores. Experiments were performed on more or less commodity hardware running Red Hat Linux. Much of this success has to do with the seed-and-extend nature of the RMAP algorithm, which is well suited to parallelization with via MapReduce. A fallback of CloudBurst is that it fails to be effective at the level of whole-human-genome data inputs – the authors handle only several-million base-pair-length sections of the genome at a time, and base their scalability on increasing the number of these sections processed, rather than handling increasingly large monolithic sections of the genome.

This weakness of CloudBurst is addressed by CrossBow, another Hadoop-based SNP calling system for NGS data, in [LSL+09]. CrossBow, unlike CloudBurst, is applied to a whole human genome (with 38-fold coverage), and finishes within a few hours when run on a large (40 node, 320 CPU core) cloud computing cluster rented through Amazon EC2. CrossBow shows greater than 50% of linear speedup – it is approximately 2.25 times as fast with 4 times as many nodes/cores. This performance is worse speedup than CloudBurst, though the two are not directly comparable due to handling different orders of magnitude of DNA input. Furthermore, the significantly improved overall performance (in large part due to the use of the Bowtie [LTPS09] aligner), the use of a larger cluster, and the ability to handle sequencing of billions of reads (necessary to (re)sequence the whole human genome) make CrossBow a significant improvement. All experiments were performed on commodity clusters, some local and others rented via Amazon EC2.

A recent (2016) paper by Rengasamy and Madduri [RM16], however, presents a strong MPI competitor in the field of NGS SNP calling. Rengasamy and Madduri present SPRITE, an MPI-based SNP calling pipeline with impressive performance and scalability. SPRITE achieves near-linear speedup – 14.6× for an increase in the number of nodes by 16 times. The data set they use is a high coverage (50×) whole reference human genome. Notably, Rengasamy and Madduri do not run their experiments on commodity clusters but on nodes of the TACC Stampede supercomputer, which should significantly improve their wall clock time (which is < 50 minutes on 16 nodes for the entire dataset) as compared to results computed on a commodity cluster. More importantly, however, it should also significantly improve their scalability, since supercomputers are generally designed specifically for this purpose. I was not able to find any specific data on the bisection bandwidth or max point-to-point latency of the TACC Stampede supercomputer, but it is almost certainly
much better than that of a commodity cluster. This raises the question of how the SPRITE system would compare to the MapReduce approaches in direct comparison, when run on comparable hardware and over the same range of nodes and input data sizes, particularly since the authors do not compare the scalability of SPRITE to any other system, MapReduce or otherwise. Since they do use the entire same dataset for their various experiments, they do not present any data on the system’s scaleup (Gustafson scaling, or response to increasing both nodes and input data size), something else which would be interesting to test experimentally.

### 2.3 Multidimensional Scaling

In [ELZ+10] Ekanayake et al. present Twister, a later version of the system presented as CGL-MapReduce in section 2.1 (K-means clustering). Twister implements a variant of the MapReduce paradigm that has been designed and optimized for iterative MapReduce. Two of the most significant changes to the MapReduce paradigm in Twister are the replacement of writing intermediate results to disk with communication via a publish/subscribe messaging model (inherently less fault-tolerant), the use of map and reduce tasks that persist over multiple iterations, keeping some static input loaded and receiving variable data in each iteration from the previous iterations. These modifications and others mean that Twister cannot directly recover from the loss of an individual map or reduce task due to individual machine failure as vanilla MapReduce does. Rather, Twister saves state between iterations, and recovers from machine failures by rolling back a few iterations as necessary until a recoverable state is found, and restarting from that point. As a result of all of these modifications and others, and as mentioned above, Twister gains a great deal of performance on this class of problems for which vanilla MapReduce is spectacularly non-performant, but also loses a significant amount of its fault-tolerance and some ease-of-use as well. For example, users are required to manually break up large data sets into multiple files [ELZ+10].

Ekanayake et al. apply Twister to a few different problems, including MDS or multidimensional scaling. Multidimensional scaling is an approach to the general problem of dimension reduction. Essentially, we have data in a high-dimensional space which we would like to reduce to a lower-dimensional space, such as a 2- or 3-dimensional one, often for purposes of visualization. We attempt to map down from the higher-dimensional space to the lower-dimensional space in such a way that “distance,” or some appropriate dissimilarity matrix, is preserved as much as possible. The particular MDS algorithm that Ekanayake et al. apply Twister to is a common one known as SMACOF (Scaling by Majorizing a COmplicated Function). Due to the spectacular inefficiency of Hadoop and DryadLINQ on K-means clustering, another iterative problem, earlier in the paper, they do not compare the Twister version to either of these, or for that matter to any other SMACOF implement-
tation. They do, however, run experiments that essentially test the speedup of the Twister implementation of the algorithm, testing it on a data set of 35339 points over a range of CPU cores from 64 to 192 and graphing the resulting “parallel efficiency,” a metric defined so that linear speedup corresponds to a value of 1 and sublinear speedup to proportions between 0 and 1. The observed parallel efficiency for 192 (= 3 × 64) is a little more than 0.8.

Bae et al. present in [BQF12] an MPI-parallelized implementation of the SMACOF algorithm for multi-dimensional scaling. They note that the SMACOF algorithm is quadratic \( (O(n^2)) \) not only in time but also in space, making parallelization in fact necessary for inputs beyond a certain size, since the problem becomes too large to hold the size \( n^2 \) arrays in system memory, without even taking into consideration issues of speed or time efficiency. Much of the SMACOF algorithm consists of large matrix multiplications, which Bae et al. handle by breaking up the large matrices into blocks and multiplying the submatrices in parallel. They experiment with varying ways of breaking up the matrices and find that row-based or highly skew approaches are highly inefficient, others are much better, and roughly square partitions are best. The reduced skew results in load-balancing that improves the overall efficiency. The authors run experiments on two data sets, one taken from PubChem consisting of descriptions of various compounds and their many chemical properties, and the other consisting of biological sequence data. To range over various input sizes, they choose various subsets of these two primary datasets. The authors also use two different clusters, one with 8 nodes, 4 CPUs per node, and 16 cores per node, for a total of 128 cores; the other with 32 nodes, 4 CPUs per node, and 24 cores per node, for a total of 768 cores. They graph the performance (elapsed time vs # of cores/nodes) and the efficiency (parallel efficiency as defined in [ELZ10]) for the experiments on each of the two datasets. The speedup observed is fairly comparable to that in [ELZ10] – roughly 0.8 for a 3x increase in nodes/cores, although in neither paper are the actual data given except in graph form, so that direct and accurate comparison is difficult. The authors also break down the computing work into various subcomponents and graph the efficiency of each share of the overall computation, observing that the pure matrix multiplication portion is in fact almost completely linear, and that much of the overhead comes from the what they term the ‘inevitable’ parallel overhead of the message-passing etc. of MPI. The authors briefly discuss some related work in the parallelization of the MDS problem, but do not compare the performance or scalability of these approaches to their own. They also note that “there are important problems for which the data sizes are too large for even our parallel algorithms to be practical,” and reference an approximate, interpolation approach that they have done work with in other published papers. Finally, it is perhaps worth noting that the three authors of this paper also worked on the Twister paper ([ELZ10]) where parallelization of MDS was accomplished via the Twister MapReduce-style runtime.
2.4 PageRank

One of the other problems/algorithms that Ekanayake et al. test their iterative MapReduce runtime Twister on is PageRank [ELZ+10]. They implement the standard PageRank algorithm, with a few optimizations tailored to the MapReduce paradigm and the specific advantages of Twister, such as holding static data in memory through successive iterations of MapReduce jobs. Their implementation exhibits nearly linear scalability under data increase (measurement of runtime when input data size is increased and number of nodes held constant), and maintains 80% efficiency when the number of cores is doubled (with the same data) from 128 to 256.

In [LK13] LaSalle and Karypis present an MPI runtime designed for “the efficient and transparent disk-based execution of distributed-memory parallel programs” which implements a significant subset of MPI and runs existing MPI programs unmodified, so long as they utilize only that subset. It achieves its goals through careful control and coordination of memory and disk use. They test their system on three algorithms, including PageRank (PageRank, spherical k-mean, and stochastic gradient descent matrix factorization). The cluster used was a small commodity cluster, and the experiments were run on either one or four nodes. For PageRank and spherical k-means they compared their BDMPI implementation directly to MapReduce approaches (Pegasus for PageRank and Mahout for k-means, both of which run on Hadoop), with favorable results, though they ran the MapReduce solutions only on four nodes, and therefore can compare only efficiency and not scalability. LaSalle and Karypis observe that all of their BDMPI algorithm implementations scale superlinearly (4.2x, Amdahl scaling) to four nodes. They note that this superlinear scaling is likely due to the increased memory going from one node to four, but do not elaborate on the fact that this means it probably does not say much about the general scalability of these BDMPI algorithms. It is common for scalability to superlinear at first, due to memory constraints on a single core and inefficiencies of the single-core version of the application, and this does not mean that they will scale well later, as the number of nodes continues to increase. By not noting this, by only considering the scalability to four nodes, and by not comparing their scalability to the MapReduce implementation LaSalle and Karypis emphasize the strengths of the BDMPI system and do not investigate its possible (even likely) weaknesses. However, they do show drastically greater performance than the MapReduce implementations they have chosen to compare themselves to - the Pegasus (MapReduce) implementation of PageRank took 234.93 minutes on 4 cores and their best BDMPI implementation took less than 4 minutes on the system (their worst took less than 10 minutes!).

A recurring theme I noted throughout the literature I could find on parallel implementations of PageRank, whether they used MapReduce, MPI, or some other parallelization technique, was a lack of emphasis on scalability. There was often little data or discussion
on scalability, an absence of scalability metrics, and experiments were often run only a small number of nodes (roughly 1-8). As a result, there is also little comparison of scalability between different PageRank algorithms, including comparison between MapReduce and MPI approaches.

3 Summary and Analysis

In the scientific applications examined, and indeed generally, the MapReduce paradigm displays both clear advantages and equally apparent weaknesses. Its scalability, fault-tolerance, and ease-of-use, combined with the wide adoption of, and support for, the open-source Hadoop implementation, have opened up the capabilities of parallelization to a much wider field of applications in both science and industry than were previously practical, and allowed the scaling of solutions to data sets larger than ever practically considered before in “big data.” Nevertheless, in its original form, MapReduce has a number of weaknesses, one of the most apparent of which is its poor performance on iterative algorithms. Because of its advantages, these problems have attracted considerable attention in the form of attempts to modify or extend particular MapReduce implementations or the MapReduce paradigm to better accommodate iterative algorithms and other weaknesses of MapReduce.

A prominent example of this approach which has come up multiple times is Twister [ELZ+10], an independent MapReduce-style runtime which is engineered to efficiently and scalably run iterative algorithms, such as k-means and PageRank. Although the authors demonstrate performance vastly superior to vanilla Hadoop and near to an MPI solution in at least a few applications, the modifications to the MapReduce paradigm in Twister significantly affect its fault-tolerance and perhaps also its ease-of-use. With regards to fault-tolerance, the author’s themselves note (emphasis mine):

Our approach is to save the application state of the computation between iterations so that in the case of a failure the entire computation can be rolled back few [sic] iterations. Supporting individual map or reduce failures require [sic] adopting an architecture similar to Google, which will eliminate most of the efficiencies we have gained using Twister for iterative MapReduce operations. [ELZ+10]

Ekanayake et al. [ELZ+10] achieve the improved performance of Twister on iterative algorithms (relative to Hadoop or other vanilla MapReduce) through several modifications. One of the ways they do this is by allowing individual map and reduce tasks to receive both static data and variable data, and allowing the tasks to persist over multiple iterations, holding onto their (usually much larger) static data and only needing to read anew the variable data for the given iteration. They also do not implement a full distributed file-
or operating-system such as GFS or HDFS, but rely on passing filenames directly and implementing only the minimum file IO requirements they thought necessary to support the MapReduce concept of "moving the computation to the data." This system requires manual file partitioning, which may significantly impact usability. Crucially, however, they also replace the writing of intermediate results and all other IO and communication between tasks and nodes with a publish/subscribe messaging system. This means that the variable data for the given iteration, and all intermediate results, will be lost in the case of single machine failure, requiring Twister to roll itself back for a few iterations until it reaches a stable recovery point, and losing a significant amount of work, and as they themselves have observed, these modifications were a crucial part of the improved performance and scalability that they achieved (relative to traditional MapReduce). Therefore, Twister seems to have sacrificed fault-tolerance for performance.

As mentioned earlier, since the authors did not rigorously evaluate the effect of their modifications on the fault-tolerance of Twister relative to traditional MapReduce (say, via Hadoop) or MPI through either empirical or theoretical means, it is entirely possible they have achieved the performance of MPI precisely by abandoning the fault-tolerance of MapReduce. Similarly, by requiring manual partition of files, they have also decreased ease-of-use, and they have also almost certainly reduced ease-of-use by creating a separate implementation of MapReduce that is different from established, supported ones such as Hadoop. The barrier of changing tools is not an insignificant one, particularly when one is well-supported, well-used, and heavily tested, and the other is not. Nevertheless, it is also possible that these tradeoffs have achieved a useful and effective compromise between the advantages of MapReduce and MPI. Perhaps the ease-of-use is almost that of Hadoop and the fault-tolerance, while significantly worse than Hadoop, vastly exceeds that of MPI. If this were the case, Twister would arguably be a very worthwhile compromise, particularly for the large and important class of data- and compute-intensive iterative algorithms. This indicates a potential topic of interest for future research in examining concretely the degree to which fault-tolerance and ease-of-use have been sacrificed in Twister, and perhaps also performing further performance and scalability comparisons. One possible tool for theoretical analysis of the tradeoffs in fault-tolerance is given by [JS13], where Jin and Sun provide an analytical model for evaluating when the losses to machine failure outweigh the usual performance benefits of MPI.

In the original CGL-MapReduce paper ([EPF08], the system that later became Twister) Ekanayake et al. examine the k-means clustering problem as a means of testing their modified MapReduce runtime on iterative algorithms against Hadoop and MPI and demonstrating the superiority of CGL-MapReduce to vanilla MapReduce such as Hadoop for this kind of application. However, as mentioned earlier, they notably do not run any experiments testing scalability in terms of changing numbers of nodes, either speedup (Amdahl scaling) or scaleup (Gustafson scaling). They only test scalability in terms of increasing
the data size while holding nodes constant. The parallelization method using MPI proposed by Dhillon and Modha in 2002 [DM02] demonstrated truly near-linear scalability with regards to both speedup and scaleup via both theoretical/analytical and empirical means, and has been well-received as the standard method of parallelizing the algorithm. For this reason, it seems important to compare a proposed MapReduce method, such as Twister, in comparison to the standard MPI approach. While Ekanayake et al. did compare themselves to an MPI implementation (presumably using more or less the same good method proposed by Dhillon and Modha), they did so only on performance over varying data input sizes. Admittedly, Dhillon and Modha demonstrate that it is on large data sets (not smaller ones) that their approach is most scalable, and gets truly near-linear results, and it is on the larger data sizes that the CGL-MapReduce implementation is most competitive with MPI, so that is a good sign for their implementation and its performance and scalability. However, since no empirical comparison of the two on these metrics (speedup and scaleup) has been done, we do not actually know how they compare, particularly since the CGL-MapReduce experiments used only 5 nodes. As mentioned previously, if CGL-MapReduce/Twister is not able to compete with MPI in performance and is more difficult to use and less fault-tolerant than Hadoop, it is worth questioning its true utility.

In the field of bioinformatics, MapReduce-based tools for SNP calling such as CloudBurst and Crossbow (both running on top of Hadoop) have made a big splash and had tremendous success, and are perhaps the dominant method at this point for parallelized SNP calling of next-gen sequencing data. Still, there are challengers from the more traditional MPI side, as demonstrated by SPRITE [RM16]. Here, however, it is the MPI challenger that seems weaker. Rengasamy and Madduri perform their experiments on a supercomputer, an unfair comparison on either performance or scalability to the commodity clusters generally used in MapReduce studies, and do not actually compare their proposed system SPRITE to any parallel competitors in their experiments. They only compare themselves on a single-node system to representative, state-of-the-art sequential SNP calling pipelines. As a result of the use of different and powerful hardware and a lack of direct comparison to parallel competitors run on said hardware, we have little ability to compare their scalability or performance to MapReduce (or other!) competitors using the data they have provided. An open topic for further research, then, would be to directly compare SPRITE to one or more MapReduce competitors such as CloudBurst or CrossBow on the same hardware (perhaps a mid-size commodity cluster) on the basis of performance and scalability. Comparisons could also be made to other, non-MapReduce competitors.

If SPRITE’s impressive results hold in that case, it would perhaps indicate that, at least in some contexts, MapReduce and MPI (or other traditional approaches to parallelization) serve distinct and complementary roles. In light of MapReduce’s ease-of-use and massive popularity, particularly via its open-source implementation Hadoop, perhaps it serves the role of a “scripting language” in the context of HPC, and traditional tech-
nologies the role of a low-level language such as C. In other words, Hadoop and other MapReduce implementations may be particularly well-suited to rapidly prototyping new applications of parallelization, dramatically expanding the number of amenable parallel-computing applications and the speed with which they can be developed. Then traditional, more engineering-intensive techniques such as MPI would serve the distinct and complementary role of taking existing parallel applications prototyped in a MapReduce-style system and, through greater engineering effort and attention to the particular challenges of the algorithm at hand, optimizing them for repeated high-performance use, once their general utility has been established and there is high demand for squeezing out more parallel efficiency from them.

While I found this hard to believe, it seems that PageRank has not been systematically examined from the perspective of its scalability in parallelized applications. As I noted above, many papers on parallel PageRank implementations failed to consider scalability metrics, or ran their experiments only on small numbers of nodes, or failed to compare their scalability results to those of other, existing approaches. This would seem to be a significant opportunity for further research. Given its iterative nature, it is likely that the MapReduce paradigm will not prove very effective for PageRank — indeed, it has been criticized on this account, though approaches to iterative MapReduce such as Twister provide some answers to this critique. In practice, however, PageRank is generally run on the converged results from earlier data, and may thus involve only a few iterations. In this context, the suitability of PageRank to Twister depends on the ratio of queries served to updates to the PageRank graph, and how tolerable results from previous PageRank calculations on old data are.

An interesting perspective on this issue is raised by Lin in his paper entitled “MapReduce is Good Enough? : If All You Have is a Hammer, Throw Away Everything That’s Not a Nail!” [Lin13]. Lin makes the argument, from a very engineering and industry based perspective, that since Hadoop and the Hadoop stack is so widespread and well-supported, it makes more sense to use it even when it is not theoretically the ideal tool by, in essence, adapting the problem to fit the tool. More precisely, he argues that iterative algorithms can be avoided by finding non-iterative algorithms that solve the same problems, or can be ameliorated by running them only over a few iterations. To be more specific about the latter point, he notes that in practice, PageRank is rarely run from scratch (having been initialized with a uniform distribution and then running from there until convergence is achieved). Instead, PageRank in application is almost always run starting from the previously computed PageRank vector on current version of the previous graph. For example, the previous PageRank vector has been stored and the web graph, now updated since the last web crawl a week past, is run through PageRank using the previous PageRank vector as the starting point for convergence. This means that it only takes a few iterations to converge to the new PageRank vector, eliminating much of the iteration overhead expe-
rienced by MapReduce systems such as Hadoop. Since Lin does not consider commonly criticized pain points of the MapReduce paradigm such as iterative algorithms to be a serious problem, he suggests instead that work focus on either improving MapReduce and the Hadoop stack as it is (the “low-hanging fruit” or low-risk research), or modifying it to solve issues he considers it to be more fundamentally incompatible with under the present model. In particular, he considers the MapReduce paradigm fundamentally incompatible with real-time or ‘online’ computation (as opposed to batch computation, which MapReduce was designed for and excels at) and with making rapid, real-time queries on large data-sets (as opposed to data analysis tasks of greater latency that often require the user to wait several minutes between queries/tasks).

4 Open Questions

There are a number of topics for further research here. One possibility is to do analytical and/or empirical analysis of the fault-tolerance losses of Twister relative to traditional MapReduce implementations such as Hadoop, and similarly its potential improvements in fault-tolerance relative to MPI. In this vein, and probably in the same research, it would be worthwhile to also consider how Twister compares to Hadoop in terms of ease-of-use, and so whether the cost of changing tools is worth any performance or scalability gains.

Another topic of interest would be to compare a classic MPI implementation of k-means clustering (such as the seminal one proposed by Dhillon and Modha in [DM02]) to the Twister implementation on the basis of scalability (both speedup/Amdahl scaling and scaleup/Gustafson scaling). In particular, it would be helpful to compare the two on a substantial range of cluster sizes.

Another interesting possibility would be to compare one of the Hadoop-based SNP calling systems for next-gen sequencing data, such as CloudBurst or CrossBow, to SPRITE. As noted earlier, Rengasamy and Madduri ran all of their scalability tests on the TACC supercomputer, which is likely to inflate their performance and scalability considerably compared to the same programs running on a commodity hardware cluster, and they did not compare their scalability results to any competitor, MapReduce-based or otherwise. It is also not clear how the inflation of scalability due to the use of a supercomputer would affect an MPI program versus a MapReduce one – one paradigm may benefit more from these hardware differences. Experimental results comparing SPRITE to a strong MapReduce competitor such as CrossBow on the basis of scalability and performance and on a mid-size commodity cluster, therefore, would provide valuable information on whether MapReduce is in fact particularly well-suited to the problem or if other techniques such as MPI are simply underutilized and under-tested in this domain.
It would be particularly worthwhile, as well, to fill in the apparent gap in the literature around scalability of PageRank implementations. An example comparison would be between Twister and BDMPI. In particular, it would be interesting to test the scalability of BDMPI on a greater number of nodes, past the point where memory effects give it superlinear scaling, to see what its true scaling behavior is, and to compare it to Twister or other competitors on this basis.

It could also be interesting to examine multidimensional scaling and compare Bae et al.'s MPI approach ([BQF12]) to the Twister implementation, particularly on the basis of scaleup (Gustafson or strong scaling), which neither group of researchers tested. A direct comparison of speedup (Amdahl or weak scaling) would also be somewhat interesting, since both papers published graphs but did not provide direct numbers on that metric. Of the topics proposed, however, this is perhaps less interesting than some of the others, as both tested speedup and did so on clusters with large numbers of nodes and cores, and seem fairly comparable on the basis of scalability from the data as presented.

A consistent trend is that researchers are not evaluating their work with metrics of performability. Performability metrics involve simultaneous analysis of performance and dependability. In essence, they model the expected performance of a system in the face of varying levels of disruption due to, say, machine or network failure. Jin et al. have given a general theoretical and empirical analysis of the relative performability of MapReduce and MPI in [JS13], but researchers proposing particular MapReduce solutions to parallelizing a given algorithm or problem, or proposing modified MapReduce runtimes, have not evaluated their algorithms or systems on the basis of performability. To be concrete, none of the scientific applications examined for this paper used a metric of performability, though it is an important to dimension to analyze, particularly in the case of modifications to MapReduce such as Twister [ELZ+10] which attempt to improve the performance of MapReduce via tradeoffs in fault-tolerance. A useful next step would be identifying appropriate metrics of performability with which to compare specific parallel solutions to a problem, both within MapReduce and between MapReduce, its variations, and other approaches such as MPI. Having identified appropriate metrics, it would also be insightful to apply these metrics to testing a number of the existing scientific applications of MapReduce examined here and comparing them to the competition.

References


16

Jimmy Lin. Mapreduce is good enough? if all you have is a hammer, throw away everything that’s not a nail! *Big Data*, 1(1):28–37, mar 2013.


