Mrs: High Performance MapReduce for Iterative and Asynchronous Algorithms in Python

Jeffrey Lund*, Chace Ashcraft*, Andrew McNabb* and Kevin Seppi*
*Computer Science Department
Brigham Young University, Provo, Utah 84604
jefflund@byu.edu, cc.ash@byu.edu, a@cs.byu.edu, k@byu.edu

***Abstract*—Mrs [1] is a lightweight Python-based MapReduce implementation designed to make MapReduce programs easy to write and quick to run, particularly useful for research and academia. A common set of algorithms that would benefit from Mrs are iterative algorithms, like those frequently found in machine learning; however, iterative algorithms typically perform poorly in the MapReduce framework, meaning potentially poor performance in Mrs as well.**

Therefore, we propose four modifications to the original Mrs with the intent to improve its ability to perform iterative algorithms. First, we used direct task-to-task communication for most iterations and only occasionally write to a distributed file system to preserve fault tolerance. Second, we combine the reduce and map tasks which span successive iterations to eliminate unnecessary communication and scheduling latency. Third, we propose a generator-callback programming model to allow for greater flexibility in the scheduling of tasks. Finally, some iterative algorithms are naturally expressed in terms of asynchronous message passing, so we propose a fully asynchronous variant of MapReduce.

We then demonstrate Mrs' enhanced performance in the context of two iterative applications: particle swarm optimization (PSO), and expectation maximization (EM).

***Index Terms*—MapReduce, high-level parallel programming frameworks, iterative algorithms**

I. Introduction

Mrs [1] is a previously published framework for MapReduce projects implemented in Python. It was shown to be easily accessible, easy to use, and readily available for a variety of environments, scheduling systems, and file systems. This ease of use and availability made it well suited for academic or research environments where it is common for users to have generic, private clusters available rather than dedicated MapReduce clusters. Regarding its performance, Mrs was shown to perform just as well or better than Hadoop for various problems, meaning the user need not sacrifice quality for simplicity.

However, iterative algorithms still suffered a significant performance penalty. Much of this penalty comes from overhead, such as communication time between nodes, writing to reliable storage, and delay between iterations. For algorithms with a single iteration, or those with very few iterations, this overhead is acceptable, but for larger iterative algorithms, it becomes excessive. For example, given a large data set, a one second overhead per iteration may be insignificant for an algorithm requiring only one iteration, but one second per iteration for thousands of iterations on the same set could increase the execution time of a CPU-bound algorithm by hours.

We propose a set of modifications to the original Mrs framework in order to improve its performance on iterative algorithms. The first is to use direct communication between nodes for most iterations, and only occasionally write to reliable memory (Section III-A). Second, we propose that reduce tasks be agglomerated with the subsequent map tasks with the same key, which reduces communication and halves the number of tasks that must be assigned each iteration (Section III-B). Third, we present a generator-callback model for submitting operations for concurrent and asynchronous evaluation (Section III-C). This model makes it easy for iterative algorithms to submit intermittent operations, such as convergence checks, to be evaluated concurrently without requiring significant bookkeeping. Finally, we introduce an asynchronous extension of the MapReduce programming model in Section IV which efficiently supports algorithms such as Particle Swarm Optimization (PSO) [2] where iteration can proceed at a different rate for each key. This model allows the same straightforward map and reduce functions to work in both synchronous and asynchronous operation.

While we consider the generator-callback function novel to our design, the rest of these modifications have already seen success in other publications. We discuss these in Section II.

Finally, we demonstrate the application of these techniques in Mrs [1]. Section V evaluates the performance of Mrs with and without these features, using PSO [2] and expectation maximizization (EM) [3] as examples, and shows significant improvements in performance. Compared to standard MapReduce, using a reduce-map operation improved PSO performance by 31%. For EM, iterations without checkpointing to redundant storage show a 91% improvement, making parallelization feasible, and the reduce-map operation gives an extra 11%. Asynchronous MapReduce improves performance of PSO by an additional 24% in the presence of moderate variability in task execution times for a total gain of 53%. Furthermore, it performs iterations faster than synchronous PSO even when task execution times are uniform. With 768 processors and uniform tasks, Asynchronous MapReduce increases the throughput by 47%.

978-1-5090-5220-2/16 $31.00 © 2016 IEEE
DOI 10.1109/PyHPC.2016.10

II. Related Work

MapReduce [4] is a popular framework for performing parallel processes, with Hadoop being its most well known and widely used open source implementation. Due to its limitations on iterative algorithms, however, several attempts have been made to modify MapReduce, or come up with a novel parallel processing framework, for the purpose of accommodating them. Most improvements or modifications consist of either modifying the the programming model, reducing communication, or optimizing the task scheduler.

MapReduce is technically defined as a map phase followed by a reduce phase, and this model must be extended, at least trivially, to support iterative programs. In most MapReduce systems, a "user program" or "driver" submits a job consisting of a map phase and a reduce phase, waits for it to complete, reads the results, and then repeats. Several MapReduce-like systems allow the user to specify an arbitrary directed acyclic graph of data dependencies [5]–[8]. Frameworks like Maiter [9] and GraphLab [10] have implemented novel models specifically directed at iterative parallel processing. Maiter uses a directed acyclic graph to represent data and dependencies, but instead of updating the data at each iteration, it only keeps track of the changes in the data from iteration to iteration. This method of iterating makes asynchronous task scheduling simple and eliminates wasteful processing. GraphLab represents a given problem with its own type of directed graph along with a shared data table to represent information common to multiple tasks.

Reducing communication has been a common modification to MapReduce because of the amount of excess overhead that is generated with iterative algorithms. One strategy for this is to store data locally. Conch [11] and Twister [12] do this. Conch stores all data in local cache and uses a memory manager to optimize total memory use. It only writes to an HDFS when memory overflows or when the algorithm terminates. Twister pushes intermediate data directly from map tasks to reduce tasks and stores data on the master between reduce and map tasks. Many other frameworks take advantage of this concept in some way [13]–[15]. Both Conch and Twister also combine certain tasks, sending data directly from one task to another instead of having each task read from memory, compute, and then write back to memory like in normal MapReduce.

Intelligent task scheduling can also help improve performance. Several frameworks have developed optimized task schedulers that take advantage of specific modifications in their framework or plan ahead to reduce waiting and communication [11], [13], [15], [16]. Another technique implemented in iMapReduce [17], [18] aims to eliminate most of the overhead by making all tasks persistent. It seems, however, that the ideal scheduling would be a form of asynchronous scheduling. iHadoop's [19] primary modification to Hadoop was the addition of asynchronous scheduling, but several frameworks have since implemented some sort of asynchronicity into their designs [9], [14], [17], [18].

We build on these concepts and apply our modifications to Mrs, resulting in a convenient, high-performance Python implementation of an iterative MapReduce framework. Each of the previously mentioned concepts is implemented in Mrs using methods described Sections III and IV, as well as the generator-callback model to handle task scheduling and completion.

III. Synchronous MapReduce

Iterative programs are sensitive to overhead such as communication costs because such overhead accumulates from iteration to iteration. We propose three improvements to reduce overhead. Section III-A shows a principled approach for limiting the frequency of checkpoints to distributed storage. Section III-B describes a reduce-map operation for agglomerating reduce and map tasks. Section III-C defines a generator-callback model for defining a directed acyclic graph of operations in an iterative program. These three improvements are evaluated later in the paper in Section V.

A. Infrequent Checkpointing to Distributed Filesystems

Traditional MapReduce implementations communicate all intermediate data through a distributed filesystem. Such filesystems replicate all data to ensure fault tolerance but come with a significant performance penalty. Communication and storage in MapReduce should explicitly address the tradeoff of speed vs. capacity and fault tolerance. An ideal runtime would be able to automatically move data between levels of the memory hierarchy, a well-known strategy for storage devices [20]. While an advanced automatic memory hierarchy may be impractically complex for a MapReduce system, communicating data from some iterations directly between nodes and storing data from other iterations to reliable storage is a simple way to balance speed and fault tolerance.

We advocate storing the output of most map and reduce tasks on the local filesystem, while storing the output from occasional checkpoint iterations to reliable storage. The operating system buffers data on the filesystem in RAM and automatically migrates it to disk if necessary. With fast iterations, short-lived intermediate data is usually deleted before ever being written to disk. This approach provides the speed of RAM when possible and gracefully sacrifices speed for capacity when the size of data is great. In the event that a node fails and makes its local storage unavailable, a MapReduce runtime can roll back to the most recent checkpoint iteration.

In almost any realistic iterative program, checkpointing should occur far less than every iteration, unlike most MapReduce systems, including Hadoop. Some other implementations, like Twister [12], go to the other extreme and do not support distributed storage, sacrificing fault tolerance. The ideal checkpointing frequency depends on the expected cost of failures vs. the cost of redundancy. We estimate and compare these costs using a simple model. While specific circumstances may warrant a customized model to determine the ideal checkpoint frequency, this simple model gives a rule

of thumb and demonstrates the cost of checkpointing every iteration.

In this simple model, failures are assumed to be independent. We also assume that the times required to compute an iteration, perform a checkpoint, or initiate a recovery are constant. Let n be the number of iterations between checkpoints, t the time to perform each iteration, c the extra time required for a checkpointed iteration, and r the time to initiate recovery after a failure. Let X be a Bernoulli-distributed random variable indicating whether a failure occurs during an iteration, with probability determined by the product of the mean time between failures in a cluster f and the total time per iteration (including the amortized cost of checkpointing):

$$X \sim Bernoulli\left(\frac{1}{f}\left(t + \frac{c}{n}\right)\right)$$

Let $Y \sim Uniform(n)$ be a random variable indicating the number of iterations since the last checkpoint, which is independent of X. Then the expected value of the number of seconds of extra work in an iteration is:

$$E\left[X\left(r + Yt\right)\right] = \frac{1}{f}\left(t + \frac{c}{n}\right)\left(r + \frac{n}{2}t\right)$$

If this is less than the amortized cost of checkpointing per iteration ($\frac{c}{n}$), then redundancy costs more than it helps. The breakeven point is given by solving for n:

$$n = \max\left[1, \frac{1}{t}\left(\sqrt{\left(\frac{c}{2} + r\right)^2 - 2c(r - f)} - \left(\frac{c}{2} + r\right)\right)\right]$$

Most reasonable values cause n to be larger than 1. For example, suppose that writing to reliable storage adds 10 seconds per iteration ($c = 10$) and that initiating recovery from a checkpoint requires 60 seconds ($r = 60$). Note that the values for c and r are conservative, and increasing c or decreasing r would increase n. For a program with moderately slow one-minute iterations ($t = 60$) and frequent failures on average once every three hours ($f = 10800$), the breakeven point n is 6.7. For a program with fast iterations ($t = 1$) and a moderate failure rate of one failure in a cluster per week ($f = 604800$), the breakeven point n rises to 3413. The actual ideal frequency of checkpointing depends on individual circumstances, and many short-running programs may not require checkpointing at all.

B. Reduce-map Operation

Iterative MapReduce programs consist of a string of iterations, each with a map operation and a reduce operation. The new task dependencies between iterations motivate rethinking the decomposition of work into tasks. The output from each reduce task is the sole input to a single map task in the next iteration. Some systems take advantage of this relationship between tasks by scheduling them to the same processor or starting a map task before all preceding reduce tasks are complete [17], [19]. We instead agglomerate each reduce task with the map task that uses its output, which removes this communication and halves the number of tasks that the master must assign each iteration. Figure 1 shows the dependencies between tasks with separate reduce and map tasks (Figure 1a) and with combined reduce-map tasks (Figure 1b).

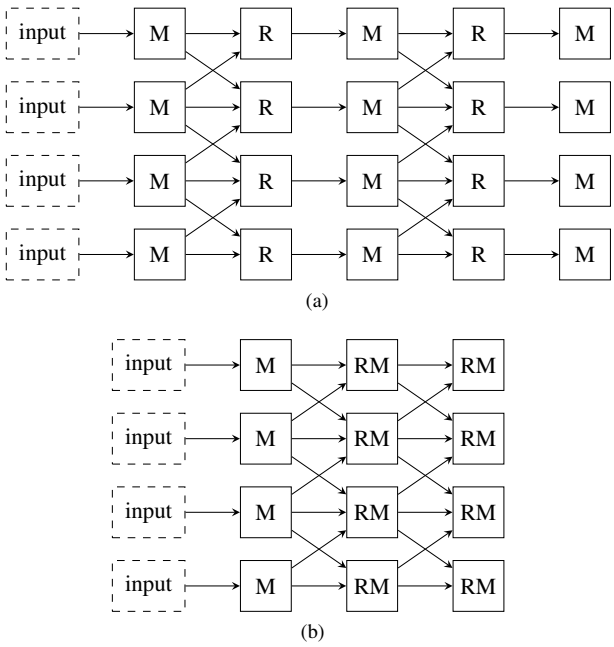


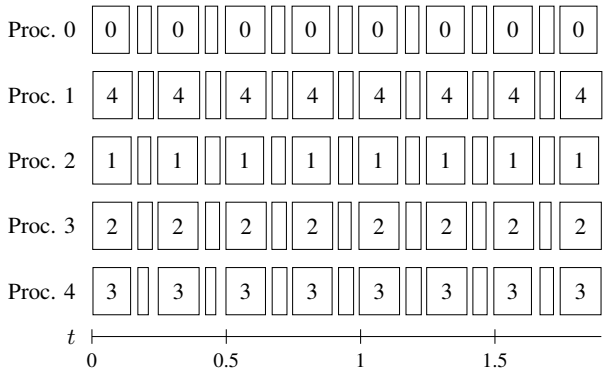
Fig. 1: Task dependencies of a typical iterative MapReduce program with (a) standard map (M) and reduce (R) operations, contrasted with (b) combined reduce-map (RM) operations.

In principle, the master might be able to autodetect these fine-grained data dependencies, but we allow the user to either specify a reduce-map dataset or separate reduce and map datasets. The user still provides a map function and a reduce function, but specifying a reduce-map operation allows the runtime to combine tasks and eliminate communication.

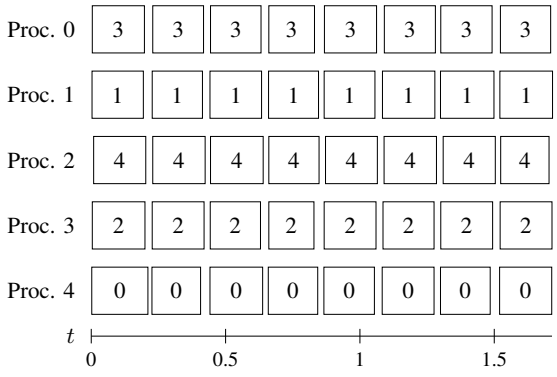
Figure 2 demonstrates the difference between MapReduce using separate reduce and map operations and MapReduce using combined reduce-map operations. Combining the reduce and map eliminates the time spent in assigning each reduce task and waiting for it to complete.

C. Iterative Programming Model

The standard MapReduce model defines a single map phase followed by a single reduce phase [4], but iterative programs execute an arbitrary number of operations and often need to compute a loop termination condition that depends on the results. Computing convergence checks infrequently and concurrently with subsequent iterations improves performance, but most MapReduce implementations do not provide any mechanism to specify this behavior. We propose an alternative model for defining operations that allows programs to specify complex behavior without becoming inherently complicated. This model is available for iterative programs that require such behavior but is not required for traditional single-iteration MapReduce programs.



(a) standard reduce and map operations



(b) combined reduce-map operations

Fig. 2: Actual task execution traces generated from a sample application (particle swarm optimization, see Section V-A1 for details) without and with combined reduce-map tasks. The run with reduce-map operations avoids the overhead of an independent reduce task and completes sooner. The horizontal axis is measured in seconds, with the left and right sides of each box aligning with the task's start and stop times. The number in each box is the key of the map task.

Varying the operations that are performed each iteration—for example, only performing convergence checks or printing intermediate output occasionally—can significantly improve performance. Suppose a program runs one second per iteration and that evaluating the loop termination condition requires a tenth of a second. If this loop condition computation is performed every iteration, it adds about 6 minutes over the course of an hour. Reducing the check to once per minute extends execution by an average of 30 iterations but still saves about 5 minutes total.

We represent parallel computation with a directed acyclic graph of datasets. A *dataset* represents data to be produced along with the associated operations required to produce it. In the representation of computation as a directed acyclic graph, the edges are the work, and the vertices are the data. Such datasets are similar in spirit to resilient distributed datasets [6]. When a user program submits datasets for asynchronous evaluation, the runtime performs computations in any order consistent with the dependency graph. Unlike the lazily evaluated tasks in Ciel [7], these datasets are evaluated eagerly. Because the next iteration can begin before evaluation of the loop condition completes, both operations can be performed concurrently. Likewise, the runtime can begin work on subsequent iterations while a user program is collecting and printing intermediate results. Unfortunately, manually managing a backlog of submitted datasets is tedious and error-prone, particularly if the work varies between iterations.

We propose a generator-callback model for submitting an arbitrary directed acyclic graph of asynchronously evaluated datasets and for handling their completion. The generator-callback model requires the program to provide a `generator` method. The `generator` method serves as an iterator or coroutine that produces work to be done. It submits each dataset for computation, along with an optional callback function to be called when computation completes. The master keeps a backlog of pending datasets, and if the backlog gets full, the generator blocks when it submits a dataset, later resuming when the backlog shrinks. Implementation is especially straightforward in languages that natively support coroutines, such as Python. As each dataset completes, the master calls the associated callback method, which can optionally read and process the results in parallel with subsequent MapReduce iterations. Termination is triggered either by the backlog exhausting after the generator completes or by a callback function returning `False` to indicate that the loop termination condition has been met. This model allows the MapReduce system itself to manage the backlog of datasets rather than exposing the details to the user. Manually maintaining a backlog requires bookkeeping that runs contrary to the simplicity of MapReduce.

Programs using the generator-callback model have greater flexibility and performance. This model is optional but may provide significant benefits for iterative programs that use it. Program 1 is a program which submits one MapReduce step at a time. Unfortunately, the structure of this program forces computation to wait while the master blocks on pending operations, performs the convergence check, and outputs intermediate results. Program 2 uses a generator-callback API to gain flexibility and performance. Note that in this example, an operation is submitted in the form of a declaration of the dataset it is to produce, not the operation itself. The generator function submits several iterations in advance, pausing only when the submit call (or yield statement) blocks. This allows tasks to be assigned with lower latency. The generator function also runs convergence checks with limited frequency to reduce overhead. These convergence checks are performed concurrently with subsequent iterations and could be submitted as datasets if they represent significant computation. The simple generator-callback structure makes it easy to specify computation that varies from iteration to iteration and to read data asynchronously as computation completes. Both the blocking program and the generator-callback program use the same simple map and reduce functions.

Program 1 The structure of a generic iterative program using a standard iterative-unaware MapReduce API.

```
run_batches():
    # Intialize key value pairs with empty data.
    init_file = makeTempPath()
    for element_id = 1 to NUM_ELEMENTS
        init_file.writePair(element_id, "")

    # Perform mapreduce to obtain initial data.
    job = new_job()
    job.setInput(init_file)
    job.setMapper(init_map_func)
    job.setReducer(identity_reduce_func)
    data_path = makeTempPath()
    job.setOutput(data_path)
    job.waitForCompletion()
    last_data = data_path

    # Perform mapreduce iteratively.
    for iteration = 1 to MAX_ITERATIONS
        # Run a mapreduce iteration and wait for a dataset.
        job = new_job()
        job.setInput(last_data)
        job.setMapper(map_func)
        job.setReducer(reduce_func)
        data_path = makeTempPath()
        job.setOutput(data_path)
        job.waitForCompletion()
        last_data = data_path

        # Occasionally output and run convergence check.
        if iteration % CHECK_FREQUENCY = 0
            # Iteration stalls until this completes in serial.
            data = readAllFiles(data_path)
            perform_output(data)
            if converged(data)
                break
```

Program 2 The structure of a generic iterative program using a generator-callback MapReduce API for performance and flexibility.

```
generator(queue):
    # Intialize key value pairs with empty data.
    kv_pairs = empty list
    for element_id = 1 to NUM_ELEMENTS
        kv_pairs.append(element_id, "")

    # Submit request to initialize curr_data.
    curr_data = MapDataset(kv_pairs, init_map_func)
    queue.submit(curr_data, NULL)

    for iteration = 1 to MAX_ITERATIONS
        # Submit asynchronous request to map iterm_data.
        interm_data = MapDataset(curr_data, map_func)
        queue.submit(interm_data, NULL)

        # Submit asynchronous request to reduce curr_data.
        curr_data = ReduceDataset(interm_data,
                                  reduce_func)

        # Occasionally submit output or convergence check.
        if iteration % CHECK_FREQUENCY = 0
            # Iterations continue in parallel with callback.
            queue.submit(curr_data, output_callback)
        else
            queue.submit(curr_data, NULL)

output_callback(data):
    data.readAllFiles()
    perform_output(data)

    # Continue processing if not converged.
    return !converged(data)
```

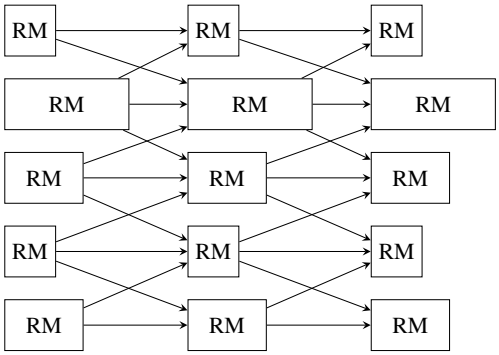
IV. Asynchronous MapReduce Programming Model

Iterative MapReduce can serve as a simple message passing framework. A map task serves to update an object, emit it, and emit messages to other objects. Between map tasks and reduce tasks is an implicit barrier for communication to complete, and a reduce task aggregates messages and emits the object, updated with information from the messages. With a reduce-map operation, the second implicit barrier, between the reduce and the following map, is removed. In the context of message passing algorithms, the MapReduce framework conceptually manages all communication, leaving map and reduce functions focused on the essence of the algorithm.

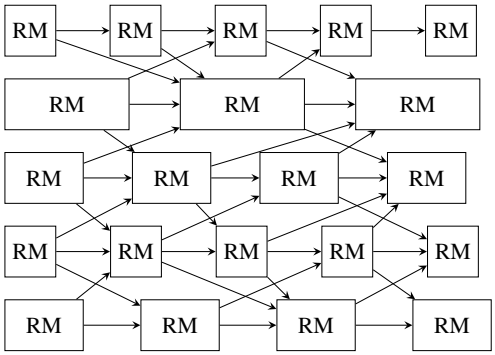
Not all iterative message passing algorithms require a barrier between each map operation and the following reduce. Such algorithms take advantage of all of the messages that have been received so far, and consideration of late-arriving messages is delayed to the next iteration. This class of algorithms is not expressible in the standard MapReduce programming model. Figure 3 illustrates task dependencies in an iterative program with heterogeneous task execution times. In synchronous MapReduce (Figure 3a), the barrier betweeen iterations leaves the faster processors idle, but in asynchronous MapReduce (Figure 3b), the faster processors evaluate more iterations. The benefit can be similar on homogeneous processors if the map and reduce execution times vary or if there are a large number of processors.

We extend the MapReduce programming model to allow asynchronous message passing algorithms. In Asynchronous MapReduce, the programmer may specify that computation of a dataset may begin before all of the tasks in its parent have completed. Unfinished tasks continue execution, and upon completion, their results are added to a subsequent dataset specified by the programmer. The runtime framework keeps track of messages sent to keys with uncompleted tasks and ensures that they do not get lost. These pending messages are included in the same dataset as the results of the task when it eventually finishes. This simple model assumes only that a key refers to a specific object that remains fixed in each iteration. It works for programs that require multiple map and reduce phases in each iteration, and it is compatible with optimizations like the reduce-map operation.

Adapting a message passing MapReduce program to the asynchronous model requires the programmer to be aware of



(a) synchronous



(b) asynchronous

Fig. 3: Task dependencies for reduce-map tasks in synchronous and asynchronous iterative MapReduce. Asynchronous MapReduce makes much more efficient use of processors.

three new parameters to datasets:

- `async_start`
- `blocking_ratio`,
- `backlink`.

The `async_start` parameter is a boolean indicating whether a dataset can start asynchronously while some tasks in its input are still running. The `blocking_ratio` parameter determines the minimum fraction of tasks that must be completed before any child dataset can start asynchronously and defaults to 1 (fully synchronous). The `backlink` parameter specifies an earlier dataset from which uncompleted tasks are inherited. New tasks are only started for those keys whose corresponding tasks in the `backlink` dataset were completed before any asynchronous execution of its children began.

Although implementation of this model in the runtime framework is not quite trivial, its effect on the map and reduce functions is minimal. The semantics of the map function is unchanged. It still updates an object, emits it, and emits messages. The reduce function, however, is no longer guaranteed to be given the object at every iteration. It might receive only messages intended for the object. In iterations where the reduce function does not receive the object, it can combine messages together, but these messages cannot be incorporated into the object yet. Note that a program that works with Asynchronous MapReduce can also run in traditional synchronous mode.

Particle swarm optimization (PSO), described in more detail in Section V-A1, is an example of a simple iterative message passing algorithm that is naturally expressed in MapReduce [2]. The map function updates the position of a particle, emits the updated particle, and emits messages to neighboring particles. The reduce function aggregates the messages from neighboring particles, and emits the particle with updated information about its neighbors. Asynchronous parallel PSO is a variant of PSO which allows the evaluation of a particle to proceed even if messages have not been received from all of its neighbors [21], [22]. The fully distributed variant of asynchronous parallel PSO makes its message passing nature particularly clear [23].

Adapting a MapReduce implementation of parallel PSO to the asynchronous model requires very few changes. The reduce function must be tolerant of input that includes several messages but no complete particle; in this case it simply emits the best message. Assuming that this case is correctly handled, the map and reduce functions are identical to those in the synchronous MapReduce PSO implementation. The driver must be updated only to include the asynchronous MapReduce parameters. The map dataset at each iteration must be specified with a `blocking_ratio` below 1 and with a `backlink` pointing at the map dataset from the previous iteration. The reduce dataset at each iteration must be specified with the `async_start` parameter set to true. In the case that a single reduce-map dataset is used, it must be given all of these options.

Figure 4 shows the improved efficiency of Asynchronous MapReduce compared to synchronous MapReduce for tasks with variable execution times. In synchronous MapReduce, all tasks in an iteration start at the same time, which is limited by the end time of the slowest task in the previous iteration. In Asynchronous MapReduce, each task can start as soon as the corresponding task from the previous iteration completes. Also note that the time between tasks is slightly less in asynchronous MapReduce, presumably due to the load on the master and the traffic on the network being less bursty.

V. Experimental Results

Although the approaches described in this paper are applicable to any MapReduce implementation, we evaluate their effects using the Mrs [1] framework. Experiments are performed on two clusters: a 2560-core cluster of 320 nodes, each with two quad-core 2.8 GHz Intel Nehalem processors and 24 GB of memory, and a 150-core cluster of 25 nodes, each with a 6-core 3.2 GHz AMD Phenom II X6 1090T processor with 16 GB of RAM, We run Mrs with and without various techniques enabled, compare the average time per iteration, and measure the average parallel efficiency per iteration. Parallel efficiency is the speedup per processor, relative to the fastest serial algorithm [24], for which we use typical serial implementations.

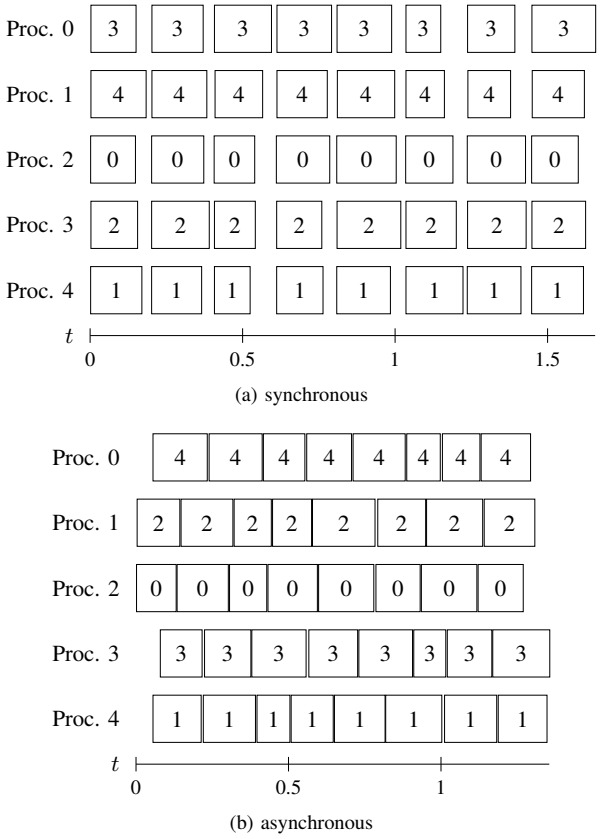


Fig. 4: Actual task execution traces for PSO with synchronous and Asynchronous MapReduce. The horizontal axis is measured in seconds.

A. Synchronous MapReduce

In addition to the serial baseline, we compare with a baseline parallel configuration. This configuration uses redundant storage and convergence checks in serial every iteration, as is common in most MapReduce frameworks, but it also performs some optimizations, such as locality-aware scheduling, which are unavailable in some frameworks.

Although most users will wish to use redundant storage and perform convergence checks, these do not need to be run every iteration. Even if the occasional iteration cannot take advantage of the improved performance, the majority of iterations are accelerated. Section V-A1 describes particle swarm optimization (PSO) and shows the parallel efficiency of parallel PSO in MapReduce with the cumulative effects of direct communication, concurrent convergence checks, disabled convergence checks, and combined reduce-map tasks. Section V-A2 describes the EM algorithm and shows similar cumulative improvements.

1) Particle Swarm Optimization: Particle Swarm Optimization (PSO) is an empirical function optimization algorithm inspired by simulations of flocking behaviors in birds and insects [25], [26]. The algorithm simulates the motion of a set of interacting particles within a multidimensional space.

Program 3 PSO program using a generator-callback MapReduce API.

```
def run(self, job):
    job.default_reduce_tasks = NUM_PARTICLES
    job.default_reduce_splits = NUM_PARTICLES
    self.check_datasets = set()
    IterativeMR.run(self, job)

def producer(self, job, iteration):
    if iteration == 0:
        kvpairs = []
        for i in range(NUM_PARTICLES):
            kvpairs.append(i, '')
        start_data = job.local_data(kvpairs)
        self.swarm_data = job.map_data(start_data,
                self.init_map)
        start_data.close()
    elif iteration <= MAX_ITERS:
        tmp_data = job.map_data(self.swarm_data,
                self.pso_map)
        self.swarm_data.close()
        self.swarm_data = job.reduce_data(tmp_data,
                self.pso_reduce)
        tmp_data.close()
        if iteration % CHECK_FREQ == 0:
            tmp_data = job.map_data(self.swarm_data,
                    self.collapse_map, splits=1)
            check_data = job.reduce_data(tmp_data,
                    self.findbest_reduce, splits=1)
            self.check_datasets.add(check_data)
    else:
        return []

def consumer(self, dataset):
    if dataset in self.check_datasets:
        self.check_datasets.remove(dataset)
        dataset.fetchall()
        self.output(dataset.data())
        if self.converged(dataset.data()):
            return False
    return True
```

At each iteration, a particle moves and evaluates the objective function at its new position. A particle is drawn toward the best value it has seen and the best value that any of its neighbors has seen. PSO can be naturally expressed as a MapReduce program, with the map function performing motion simulation and evaluation of the objective function and the reduce function calculating the neighborhood best by combining the updated particle with messages from its neighbors [2]. For computationally inexpensive objective functions, task granularity is too fine if each map task operates on a single particle. In this case, a swarm can be divided into several subswarms or islands, and each map task operates on several iterations of a subswarm of particles [27], [28].

Program 3 is an implementation of PSO using a generator-callback API as in Program 2 from Section III-C.

We find significant performance improvements for PSO in MapReduce. We use PSO with subswarms of 5 particles applied to the 250 dimensional Rosenbrock function [29]. Each

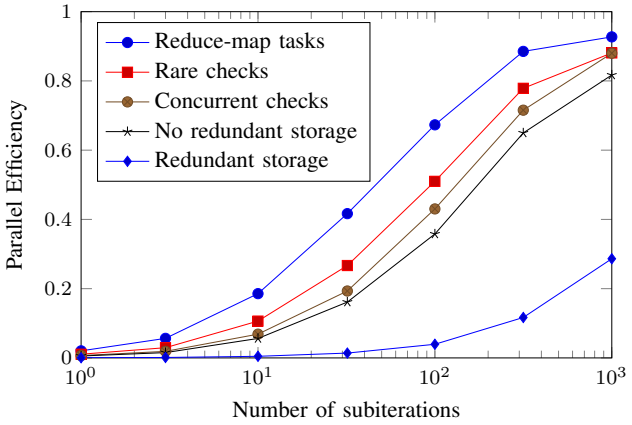


Fig. 5: Parallel Efficiency (per iteration) of PSO in MapReduce with a sequence of cumulative optimizations. The x-axis represents the problem size (the number of subiterations in each map task). "Redundant storage" represents the baseline performance, with all data stored to a redundant filesystem and with convergence checks occurring after each iteration. "No redundant storage" shows performance for iterations with data communicated directly between processors. "Concurrent checks" shows further improvements when the convergence check is performed alongside the following iteration's work. "Rare checks" avoids unnecessarily frequent convergence checks. Finally, "reduce-map tasks" agglomerates each pair of reduce and map tasks into a single reduce-map task.

subswarm runs for 50 "subiterations" in each map task. A baseline serial implementation of PSO takes an average of 0.26 seconds to simulate 5 particles for 50 iterations. Note that unlike the parallel implementation, this serial baseline does not serialize the state of particles between iterations. Combining reduce and map operations into a single reduce-map operation significantly reduces the overhead of assigning tasks. With separate reduce and map operations, the average time per iteration is 0.79 seconds. With a combined reduce-map operation, the average time per iteration drops to 0.55 seconds. This represents a reduction of 30.7% in each iteration.

Even a most inefficient MapReduce implementation would be able to provide reasonable parallel efficiency for a large enough problem size, but features that take into account the nature of iterative algorithms are able to extend the range of reasonable performance to more modestly sized problems. Figure 5 demonstrates the benefits of several techniques with respect to the problem size, which in the case of PSO is the number of subiterations performed by each subswarm within each map task. Note that the improvements are cumulative and optional. Though the figure only shows the performance of a reduce-map task in conjunction with direct communication, a configuration using redundant storage would still benefit from using combined reduce-map tasks. Furthermore, a program need not be equally efficient in each iteration. For example, even if redundant storage and convergence checks are performed occasionally, the majority of iterations can benefit from these optimizations. In MapReduce implementations that make redundant storage optional, a program only pays for the level of redundancy it needs.

2) Expectation Maximization: Expectation Maximization (EM) is an iterative algorithm commonly used to optimize parameters of finite mixture models in order to maximize the likelihood of the observed data [30]. Specifically, we apply the algorithm to a mixture of multinomials model in the context of clustering text documents [31], [32]. For each multinomial component in the model, we must maintain vectors with the same dimensionality as the number of features, which can be large. This greatly increases the communication cost when running in parallel, making efficiency difficult to obtain. Other mixture models, such as mixture of Gaussians, have much smaller parameter sizes, and have been parallelized successfully with the EM algorithm [33], [34]. We choose this particular model because it is inherently difficult to parallelize. With redundant storage and convergence checks at every iteration, performance was abysmal. However, the suggested improvements give much better parallel efficiency.

A single iteration of the EM algorithm consists of two steps. For our model, the expectation step (E-step) uses the current state of the parameters to estimate partial label assignments for the data. This is followed by the maximization step (M-step), which re-estimates the parameters using those partial label assignments. This algorithm is guaranteed to never decrease the log-likelihood of the data and will always converge to a local maximum.

EM for mixture of multinomials can be expressed as a two-stage iterative MapReduce program. The first stage of the program performs the E-step. Each map processes a shard of the documents and computes a posterior distribution given the current state parameters. The reduce then combines the posterior into partial counts for each of the labels. The second stage of the program re-estimates the parameters of the model. The map task performs normalization for each of the labels, and then the reduce task combines the normalized counts to produce the updated model parameters.

We tested the MapReduce implementation of EM with the 20 newsgroups dataset, a common benchmark for document clustering [35]. After preprocessing, the dataset had a vocabulary size of approximately 80,000 unique words. As a final step, we applied random feature hashing, which maps each unique word to a predefined number of bins. Although simple, this type of feature selection has been shown to perform surprisingly well [36], [37], but other more principled dimensionality reductions such as latent dirichlet allocation [38] could also be used to reduce the feature set size.

Table I shows the efficiency of parallel EM for various reasonable feature set sizes. Note that as the feature set increases in size, the amount of communication increases at a faster rate than the amount of computation which must be performed for each task, which decreases parallel efficiency. In fact, if one were to do no feature engineering whatsoever and use all 80,000 words as features, the cost of writing

TABLE I: Parallel efficiency per iteration of EM for various feature set sizes. As expected, higher feature set sizes lead to lower parallel efficiency, but removing redundant storage significantly helps. Further gains are realized by reducing convergence checks and using the reduce-map operation.

Optimization	80	252	8000	25298
Reduce-map tasks	0.411	0.357	0.277	0.193
Rare checks	0.362	0.314	0.253	0.18
Redundant storage	0.013	0.013	0.013	0.012

this large number of features is so high, that when using a distributed filesystem, the serial implementation of EM runs nearly twice as fast as the parallel version. However, that is not the point here, rather we show that in this application, for any reasonable number of features, eliminating the use of redundant storage significantly improves performance. In addition, rare convergence checks in combination with our reduce-map operation brought runtime down from an average of 83.93 seconds per iteration to only 3.41 seconds, a 95.9% improvement.

B. Asynchronous MapReduce

The asynchronous programming model of Section IV allows asynchronous parallel PSO [21], [22] to be expressed in MapReduce. This variant of PSO is particularly well-suited for functions whose execution time has high variance, with heterogeneous processors, and in distributed environments [23]. To evaluate the behavior of asynchronous parallel PSO in MapReduce, we vary the number of subiterations performed in each map task.

With a varying number of subiterations, asynchronous parallel PSO is distinctly faster than standard parallel PSO. We draw the number of subiterations from a normal distribution with a mean of 50 and a standard deviation ranging from 0 (no variability) to 20. Figure 6 shows the difference in throughput between synchronous and asynchronous PSO in MapReduce as the standard deviation varies. The throughput of asynchronous PSO is fairly constant at around 115 tasks per second. Synchronous PSO, on the other hand, slows as the standard deviation increases, with a throughput of 73 tasks per second when the standard deviation is 20.

Even with small or no standard deviation, asynchronous parallel PSO outperforms the synchronous variant. With a standard deviation of 5, synchronous PSO with combined reduce-map tasks requires an average of 0.58 seconds per iteration, while asynchronous PSO requires only 0.44 seconds. Note that reduce-map operations provide a similar benefit with variance as it does without variance: with separated reduce and map tasks, the time per iteration for synchronous PSO rises to 0.82 seconds.

We speculate that the advantage of Asynchronous MapReduce in the case where task times are uniform is due to a more even load on the master. With synchronous MapReduce, as soon as the last task in a dataset completes, the master is suddenly able to make assignments to each of the slaves. This

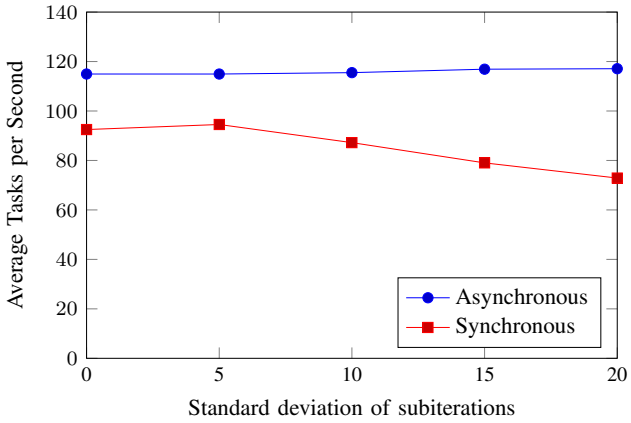


Fig. 6: The average throughput (in tasks per second) for synchronous and asynchronous PSO. The number of subiterations per map task vary, with an average of 50 and a standard deviation ranging from 0 to 20. Throughput of the asynchronous implementation is unaffected by task variance and is better even when there is no variance.

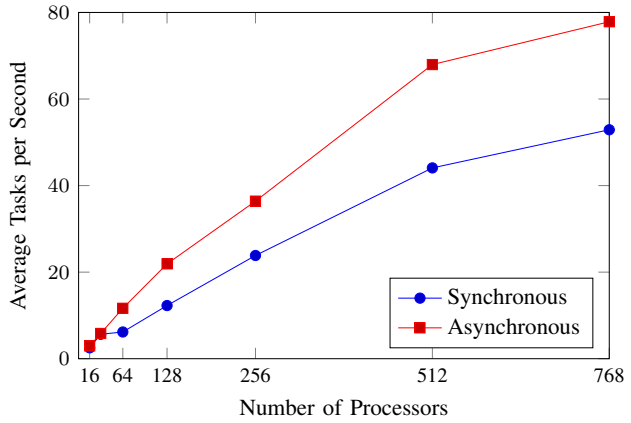


Fig. 7: The average throughput (in tasks per second) for synchronous and asynchronous PSO with respect to the number of processors. The number of subswarms is equal to the number of processors, and the number of subiterations is 1000.

creates a bottleneck, not only in the master as it makes assignments, but also in the slaves as they all start communicating at the same time. In Asynchronous MapReduce, the master has no such bottleneck because it can make an assignment as soon as a single task completes, without waiting for all other tasks in the dataset to finish. Figure 7 explores this phenomenon and shows that the effect increases with the number of processors.

VI. Conclusion

This paper takes the following approaches to make Mrs more appropriate for computationally intensive iterative algorithms:

- Checkpointing: we combine direct task-to-task communication with strategic use of a distributed filesystem to improve performance while preserving fault tolerance.
- The reduce-map operation: this operation is a combination of the reduce and map tasks which span successive iterations. It eliminates unnecessary communication and scheduling latency.
- Fully asynchronous operation: iterative algorithms which are naturally expressed in terms of asynchronous message passing can now be easily expressed and efficiently run.

These approaches have been previously shown to improve performance of parallelized iterative algorithms, and we have shown that they do the same in Mrs. Further, we add an additional approach novel to our implementation:

- A generator-callback model for task management: This model provides for both greater flexibility in the scheduling of tasks and better supports operations typically found in iterative programs, such as convergence checking being scheduled less frequently and outside of the regular MapReduce iterations.

These approaches improve the efficiency of Mrs MapReduce for all iterative algorithms but also makes it feasible for a wide range of applications where its overhead was previously too high to be practical.

REFERENCES

[1] A. McNabb *et al.*, "Mrs: MapReduce for scientific computing in Python," in *Proc. Python for High Performance and Scientific Computing*, 2012.

[2] ——, "MRPSO: MapReduce particle swarm optimization," in *Proc. Conference on Genetic and Evolutionary Computation*, 2007.

[3] C. Chu *et al.*, "Map-Reduce for machine learning on multicore," in *Proc. Advances in Neural Information Processing Systems*, 2007.

[4] J. Dean and S. Ghemawat, "MapReduce: Simplified data processing on large clusters," in *Proc. Operating System Design and Implementation*, 2004.

[5] M. Isard *et al.*, "Dryad: Distributed data-parallel programs from sequential building blocks," *ACM SIGOPS Operating Systems Review*, vol. 41, no. 3, 2007.

[6] M. Zaharia *et al.*, "Spark: Cluster computing with working sets," in *Proc. USENIX Conference on Hot Topics in Cloud Computing*, 2010.

[7] D. Murray *et al.*, "Ciel: a universal execution engine for distributed data-flow computing," in *Proc. Network Systems Design and Implementation*, 2011.

[8] C. Chambers *et al.*, "FlumeJava: Easy, efficient data-parallel pipelines," in *ACM Sigplan Notices*, 2010.

[9] Y. Zhang *et al.*, "Maiter: An asynchronous graph processing framework for delta-based accumulative iterative computation," *IEEE Transactions on Parallel and Distributed Systems*, vol. 25, no. 8, pp. 2091–2100, Aug 2014.

[10] Y. Low *et al.*, "Graphlab: A new framework for parallel machine learning," *CoRR*, vol. abs/1408.2041, 2014.

[11] R. Zheng *et al.*, "Conch: A cyclic mapreduce model for iterative applications," in *2016 24th Euromicro International Conference on Parallel, Distributed, and Network-Based Processing (PDP)*, Feb 2016, pp. 264–271.

[12] J. Ekanayake *et al.*, "Twister: a runtime for iterative MapReduce," in *Proc: High Performance Distributed Computing*, 2010.

[13] G. Mishra *et al.*, "Glister: A framework for iterative mapreduce," in *Computer, Communication and Control (IC4), 2015 International Conference on*, Sept 2015, pp. 1–6.

[14] H. Singhal and R. M. R. Guddeti, "Modified mapreduce framework for enhancing performance of graph based algorithms by fast convergence in distributed environment," in *Advances in Computing, Communications and Informatics (ICACCI, 2014 International Conference on*, Sept 2014, pp. 1240–1245.

[15] Y. Bu *et al.*, "HaLoop: Efficient iterative data processing on large clusters," *Proc. VLDB Endowment*, vol. 3, no. 1-2, 2010.

[16] J. Rosen *et al.*, "Iterative mapreduce for large scale machine learning," *CoRR*, vol. abs/1303.3517, 2013.

[17] Y. Zhang *et al.*, "iMapReduce: a distributed computing framework for iterative computation," in *IEEE International Parallel and Distributed Processing Symposium Workshops*, 2011.

[18] ——, "imapreduce: A distributed computing framework for iterative computation," *Journal of Grid Computing*, vol. 10, no. 1, pp. 47–68, 2012.

[19] E. Elnikety *et al.*, "iHadoop: Asynchronous iterations for mapreduce," in *Proc. IEEE Cloud Computing Technology and Science*, 2011.

[20] E. Morenoff and J. McLean, "Application of level changing to a multilevel storage organization," *Communications of the ACM*, vol. 10, no. 3, pp. 149–154, 1967.

[21] G. Venter and J. Sobieszczanski-Sobieski, "A parallel particle swarm optimization algorithm accelerated by asynchronous evaluations," in *Proc. World Congress on Structural and Multidisciplinary Optimization*, 2005.

[22] B.-I. Koh *et al.*, "Parallel asynchronous particle swarm optimization," *International Journal of Numerical Methods in Engineering*, vol. 67, 2006.

[23] I. Scriven *et al.*, "Asynchronous multiple objective particle swarm optimisation in unreliable distributed environments," in *Proc. IEEE Congress on Evolutionary Computation*, 2008.

[24] A. Grama *et al.*, *Introduction to Parallel Computing*, 2nd ed. Harlow, England: Addison-Wesley, 2003.

[25] J. Kennedy and R. C. Eberhart, "Particle swarm optimization," in *Proc. International Conference on Neural Networks IV*, 1995.

[26] D. Bratton and J. Kennedy, "Defining a standard for particle swarm optimization," in *Proc. IEEE Swarm Intelligence Symposium*, 2007.

[27] J. Schutte *et al.*, "Parallel global optimization with the particle swarm algorithm," *International Journal for Numerical Methods in Engineering*, vol. 61, no. 13, 2004.

[28] J. Romero and C. Cotta, "Optimization by island-structured decentralized particle swarms," in *Proc. Fuzzy Days: Computational Intelligence, Theory and Applications*, 2005.

[29] K. Tang *et al.*, "Benchmark functions for the CEC'2010 special session and competition on large scale global optimization," IEEE Congress on Evolutionary Computation, Tech. Rep., November, 2009.

[30] A. Dempster *et al.*, "Maximum likelihood from incomplete data via the em algorithm," *Journal of the Royal Statistical Society, Series B*, vol. 39, no. 1, 1977.

[31] D. Walker and E. Ringger, "Model-based document clustering with a collapsed gibbs sampler," in *Proc. ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, 2008.

[32] M. Meila and D. Heckerman, "An experimental comparison of model-based clustering methods," *Machine Learning*, 2001.

[33] N. Kumar *et al.*, "Fast parallel expectation maximization for gaussian mixture models on GPUs using CUDA," in *Proc. High Performance Computing and Communications*, 2009.

[34] G. Mann *et al.*, "Efficient large-scale distributed training of conditional maximum entropy models," *Advances in Neural Information Processing Systems*, 2009.

[35] K. Lang, "Newsweeder: Learning to filter netnews," in *Proc. International Conference on Machine Learning*, 1995.

[36] K. Ganchev and M. Dredze, "Small statistical models by random feature mixing," in *Proc. Workshop on Mobile NLP at ACL*, 1998.

[37] K. Weinberger *et al.*, "Feature hashing for large scale multitask learning," in *Proc. International Conference on Machine Learning*, 2009.

[38] D. Blei *et al.*, "Latent dirichlet allocation," *Journal of Machine Learning Research*, vol. 3, 2003.