SupMR: Circumventing Disk and Memory Bandwidth Bottlenecks for Scale-up MapReduce

Michael Sevilla, Ike Nassi, Kleoni Ioannidou, Scott Brandt, Carlos Maltzahn
Computer Science Department
University of California, Santa Cruz
Santa Cruz, CA 95060, USA
{msevilla, inassi, kleoni, scott, carlosm}@soe.ucsc.edu

Abstract—**Reading input from primary storage (i.e. the ingest phase) and aggregating results (i.e. the merge phase) are important pre- and post-processing steps in large batch computations. Unfortunately, today's data sets are so large that the ingest and merge job phases are now performance bottlenecks. In this paper, we mitigate the ingest and merge bottlenecks by leveraging the scale-up MapReduce model. We introduce an ingest chunk pipeline and a merge optimization that increases CPU utilization (50 - 100%) and job phase speedups ($1.16\times$ - $3.13\times$) for the ingest and merge phases. Our techniques are based on well-known algorithms and scale-out MapReduce optimizations, but applying them to a scale-up computation framework to mitigate the ingest and merge bottlenecks is novel.**

I. Introduction

Ingesting the input into memory and merging large sets of data are performance bottlenecks for data-intensive computing. Today's big data architectures usually scale-out by adding nodes to the system. These systems attack batch workloads with huge data sets by arming themselves with many resources (*e.g.*, cores, memory, SSDs) and by aggressively parallelizing computation. For example, MapReduce [1] leverages the resources of many connected nodes and distributed DBMSs parallelize query execution. Recently, scale-up architectures, which add more resources to a single node, have been shown to be an effective option for large batch computations at a fraction of the cost [2].

In scale-up systems, scale-out techniques, like adding resources and automatic parallelization, work well for the actual computation. However, they overlook data pre- and post-processing stages, like the ingest and merge phases. These phases are very important because they significantly affect performance and their execution times scale poorly with the input size.

A properly configured scale-up system with faster hardware and more channels can mitigate these bottlenecks. However, the time to move large amounts of data (into memory or off disk) will never be zero. Furthermore, upgrading today's systems is often times limited by insurmountable costs or marriages to legacy software. Because systems limited by the speed of data movement are entrenched in many data centers, it is important to provide techniques to help mitigate these bottlenecks.

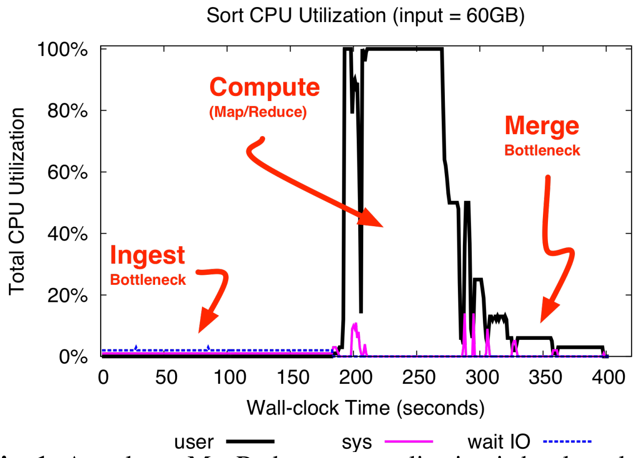


Fig. 1: A scale-up MapReduce sort application is bottlenecked by ingest and merge phases. We leverage the MapReduce model to circumvent these inefficient phases.

Fig. 1 shows how these phases bottleneck the job in a scale-up, MapReduce sort application operating on 60GB of data. The total CPU utilization (y axis) is the percentage of time spent in kernel-space code (`sys`), user-space code (`user`), and in code waiting for IO (`IO wait`); the wall-clock time (x axis) shows when each job phase occurs. The figure shows that the actual compute phase takes less than 25% of the total execution time! We also see low utilization for most of the job's execution time. Since the ingest and merge bottlenecks are largely sequential, the theoretical speedup of the program is limited.

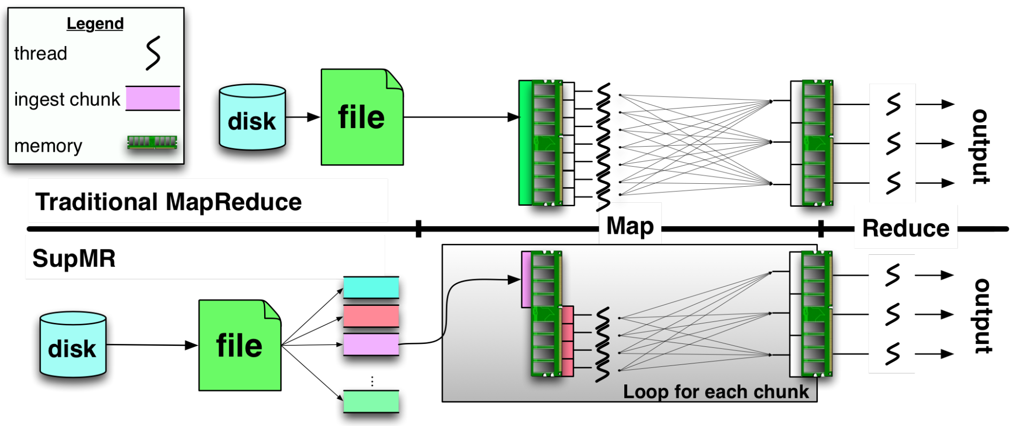


Fig. 2: A comparison of the traditional scale-up MapReduce paradigm and SupMR. The top part of the figure shows how the MapReduce model translates to scale-up components. The bottom part shows how SupMR mitigates the ingest bottleneck with an ingest chunk pipeline. Ingest chunks are read into memory while mapper threads operate on earlier chunks.

A job reads data from primary storage into main memory during the ingest phase. With large input sizes, it is probable that data movement will saturate the system's resources thereby dominating the computation time. Unfortunately, the ingest bottleneck is prevalent in modern systems because the scale of today's data sets usually renders the number of channels and the achievable bandwidth insufficient to serve data to the computation framework. For example, a system using disks instead of SSDs may not be able to serve data fast enough [2]. A system using scale-out storage with a slow network or limited number of ports may not transfer data quickly enough [7]. A system using storage silos for data with different priorities may need to move it to a local silo before beginning computation [3].

The merge phase is when the job aggregates the results of many parallel tasks. In MapReduce, the results are usually sorted during the merge phase so that data scientists can understand or query the results. It occurs after the computation and is limited by the number of records to process. For example, many parallel merge algorithms iteratively combine lists, leading to multiple scans of the data. For large input sizes with many values, inefficiencies in this phase lead to noticeable bottlenecks.

We present SupMR, a **S**cale-**up** **M**ap**R**educe runtime that leverages the MapReduce model to provide an efficient computation framework that has better performance for large data sets. Using the MapReduce model to distribute work and parallelize computation on scale-up has already been shown to be an effective model when compared to optimized thread libraries [4]. Our work builds on these techniques and mitigates the effects of the ingest and merge bottlenecks by optimizing the distinct MapReduce job phases. We make the following contributions:

1) an ingest chunk pipeline that mitigates the ingest bottleneck by parallelizing read/compute
2) a runtime modification that introduces a more efficient sort to mitigate the merge bottleneck
3) an implementation that increases CPU utilization (50-100%), resulting in job phase speedups ($1.16\times$ - $3.13\times$) and time-to-result speedups ($1.15\times$ - $1.45\times$) for 2 target applications

Other work has implemented many of these techniques in scale-out job phases, but to our knowledge, this is the first attempt at using well-known techniques (double-buffering and p-way merging) to mitigate the ingest and merge bottlenecks in a scale-up framework. Although we use the MapReduce model in this paper, the methods are still applicable to large-scale HPC systems that have distinct job phases.

II. Background: Scale-up MapReduce

MapReduce [1] is a popular programming model for scale-out because it automatically parallelizes the job, distributes the work, and is easy to program. Map tasks transform the input into key-value pairs and reduce tasks coalesce key-value pairs with common keys. The mappers operate in parallel on chunks of the input called input splits. The user supplies the map and reduce implementations to the

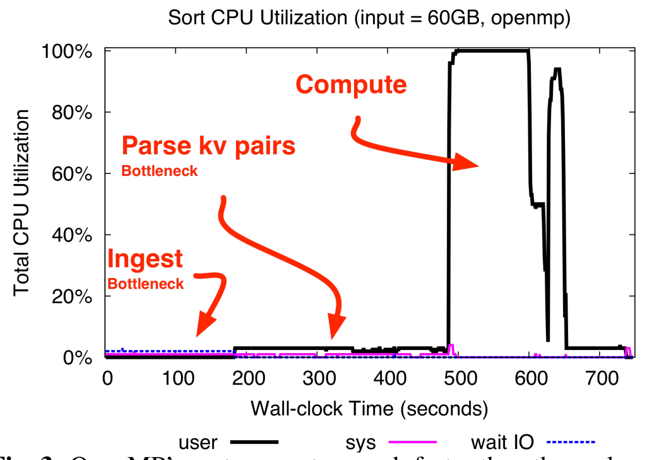


Fig. 3: OpenMP's sort computes much faster than the scale-up MapReduce sort but the total time of the job is slower because it reads data into memory and parses the data with one thread.

runtime.

Phoenix [4] is a MapReduce API and runtime implementation designed for scale-up. Phoenix uses the same model and programming interface but replaces nodes with threads, the network communication with shared memory, and the distributed file system with memory and caches. The top part of Fig. 2 shows how Phoenix executes a MapReduce job: the input data is read into memory, the mapper threads operate on input splits and output the intermediate key-value pairs back into memory, and the reducer threads coalesce intermediate key-value pairs with the same keys. Intermediate key-value pairs (between the map and reduce phase) are stored in containers and Phoenix supports different container implementations tailored to popular workloads.

A big advantage for scale-up MapReduce is the ability to store all the input and the data structures in memory, making the computation memory-bound (instead of network or disk bound). For example, the shuffle phase, which is notoriously slow on scale-out, is much faster on scale-up because all the data does not need to travel to another node's disk. Also, scale-up MapReduce can remove many of the node monitoring techniques that are necessary for scale-out, such as heartbeats.

MapReduce is also a convenient abstraction that helps lift the burden of parallel programming off the developer. As an example, we compare a sort application using Phoenix to a sort application using OpenMP [5], an API for shared memory multiprocessing. For a 60GB input size, the scale-up MapReduce sort compute phase is 214 seconds longer than the OpenMP compute phase. Despite this, the *total execution time* (i.e. the time-to-result) for the OpenMP version is 192 seconds slower. As shown in OpenMP's CPU utilization graph in Fig. 3, we have slower time-to-result because ingesting data into memory and parsing the input into key-value pairs is sequential and slow. Alternatively, the MapReduce map phase automatically parses the input into key-value pairs in parallel, saving time and increasing utilization.

To increase the resource utilization and performance on a scale-up system, SupMR introduces an ingest pipeline and modifications to the sort algorithm to mitigate the ingest and merge bottlenecks.

III. Efficient Ingest Phase

The ingest phase in a scale-up system is usually a consequence of a limited number of data channels. Scale-out can circumvent these bottlenecks by leveraging aggregate data channels in the system. For example, in scale-out Hadoop (the open-source implementation of MapReduce), the ingest phase is parallelized across many disks.

To mitigate the data ingest time on scale-up, we use a technique called double-buffering [6] to overlap the ingest phase with the map phase. In SupMR, the input is broken into ingest chunks and the ingest chunks are sequentially sent into the runtime, where they are ingested and operated on in parallel. The bottom part of Fig. 2 shows how our runtime (1) divides the file into ingest chunks, (2) loops over each chunk and starts ingest/mapper threads to work in parallel, and (3) launches the reduce function to aggregate key-value pairs.

The original MapReduce paradigm is not flexible enough to support ingest/map parallelization, so we make three changes to the map execution engine. First, we add data structures for managing data ingest chunks. Second, we introduce an ingest chunk pipeline so that parts of the input can be ingested into memory while others are operated on. Third, we force the intermediate key-value pair container to persist across the job.

A. *Managing Ingest Chunks*

In the ingest chunk pipeline, we allow different resources to operate on different parts of the data at one time. To allow this, SupMR partitions the input into small, similarly-sized units called ingest chunks. These ingest chunks are sequentially sent into the ingest chunk pipeline.

Recall that in the traditional MapReduce runtime, the entire input is partitioned into input splits and each map thread processes a single input split in parallel. SupMR's ingest chunk partitioning happens *before* producing the input splits. As a result, the ingest chunk pipeline operates on a single ingest chunk instead of the entire input. To accommodate the ingest chunk abstraction, we change the API to force the user to specify the chunking strategy and chunk size.

1) Ingest Chunking Strategy: There are two ways to chunk data: inter-file chunking, where the input is split into large chunks, and intra-file chunking, where multiple files combine to form a chunk. For inter-file chunking, the user specifies the desired chunk size in bytes and for intra-file chunking, the user specifies how many files to combine into one chunk. More complicated abstractions, such as variable sized ingest chunks or a hybrid inter/intra-file chunking approach, could allow the runtime to tune the system (i.e. ingest at size x and operate on size y) but is not implemented in our initial prototype.

We support both types of chunking so that SupMR can process different big data workloads. This flexible partitioning allows our runtime to process data generated for Hadoop jobs. Hadoop processes input as either one big file (*e.g.*, Terasort) or as many small files (*e.g.*, Word count). This flexibility is also useful for scale-out vs. scale-up comparisons, since the the workloads and inputs can be exactly the same [2], [7].

Inter-file chunking splits the file, allocates space for the chunk, and reads the chunk into memory. To ensure that chunking does not separate keys or values into different chunks, the runtime makes small adjustments to the split point: it seeks to the user-defined chunk size, checks to see if it is in the middle of a key or value, and then continually increases the split point until reaching the end of the value. For example, each key-value pair in the input for Terasort is terminated with `\r\n`, so the split function continually increases the split point until reaching a newline.

Intra-file chunking coalesces multiple files into a chunk by allocating space for the chunk (equal to the size of a single file), reading the chunk into memory, and looping until the user-defined threshold is met. The runtime dynamically increases the allocated space to ensure that all files in the intra-file chunk are collocated in RAM. If the user-defined chunk size is higher than the number of files left in the job, then the last chunk is smaller than the rest. For example, if the user wants to process 30 files with an intra-file chunk size of 4 files, the runtime will produce 8 chunks, where 7 chunks will contain the user-defined 4 files and 1 chunk will contain the 2 remaining files.

2) Ingest Chunk Size: The chunk size determines the granularity of the ingest chunk stream and influences the number of map rounds. A fine granularity improves performance because the runtime can parallelize more computation with ingest. A fine granularity also increases resource utilization because the system is starting threads more frequently - if the chunk size is too small, the computation can be dominated by thread overheads, such as synchronization.

SupMR lets the user define the chunk size because the runtime lacks the information necessary to make a good decision, such as information about the job specifications, the system hardware, or the desired streaming properties. The optimal ingest chunk size varies for different types of jobs; the chunk size for a compute-bound job should be larger than a disk-bound job to minimize the CPU cycles spent on thread management. The optimal ingest chunk size also varies depending on how fast the system can move data; a system with a disk array with many channels can handle more data per chunk than a single disk. Finally, the optimal ingest chunk size depends on the desired stream properties; large chunks encourage a slow stream with low overall utilization, which may benefit a shared compute device where many other jobs are running.

We acknowledge that the user might not know enough about the workload or the expected gains in performance to determine the optimal ingest chunk size. The best approach, which is left as future work, is to design components that factor in the expected performance and the workload characteristics (i.e. a feedback loop).

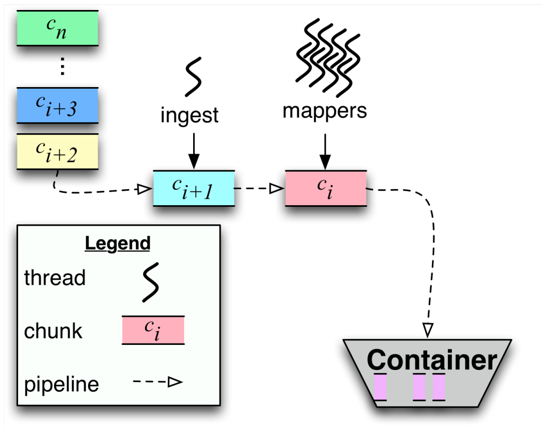


Fig. 4: In the ingest chunk pipeline, the next ingest chunk (c_{i+1}) is read into memory while the mapper threads operate on the current chunk (c_i). Processed chunks are stored as intermediate key-value pairs in a container.

B. Ingest Chunk Pipeline

The ingest chunk pipeline parallelizes the job by starting mapper threads to operate on the current chunk and an ingest thread to read the next chunk into memory. This is shown in Fig. 4, where mapper threads operate on the current chunk, c_i, while the next chunk, c_{i+1}, is read from disk into memory.

In the traditional MapReduce runtime, mapper threads are triggered once to process the entire input; the number of mapper threads is configurable by the user. The ingest chunk pipeline starts mapper threads multiple times to operate on new chunks as they arrive. These implementations are part of the "loop for each chunk" phase of the SupMR runtime in the bottom part of Fig. 2 and pseudo-code is below:

```
partition input into ingest chunks
ingest 1st chunk
for each ingest chunk do
    create thread to ingest next chunk
    run mappers on previous chunk
    destroy thread
end
run mappers on last chunk
```

In a SupMR job, there are $n+1$ rounds, where n is the number of ingest chunks. In the first round, the ingest chunk pipeline reads one ingest chunk serially (i.e. no other threads are operating). The ingest chunk pipeline then loops over the rest of the chunks, reading and computing in parallel. After the last chunk is ingested, the runtime starts the mapper threads one final time to compute on the last ingest chunk.

C. Persistent container

Recall that the container stores intermediate key-value pairs between the map and reduce phases. Each map thread inserts intermediate key-value pairs into this thread-safe container. In the traditional MapReduce runtime, this container is initialized when mappers start and destroyed after the reducers are finished.

Our runtime cannot allow the container to re-initialize every time the mappers start because we must allow multiple map tasks *across multiple rounds* to insert intermediate key-value pairs into the container. We make the container persistent, as was done in [8], by only initializing it the *first* time mappers are triggered. All subsequent map thread waves output to this same container. Since the container is still holding the same amount of key-value pairs, its size does not change from that of the original runtime.

IV. Efficient Merge Phase

For large computations with many key-value pairs, merging data is a bottleneck because the long latency of scanning keys makes the multiple round computation inefficient. Merge sort breaks the job into rounds, where each round (1) sorts many small lists in parallel and (2) merges the lists. To merge the lists, the worker repeatedly compares the first values of the lists and inserts the smaller into a new list. The next round repeats the process with half the threads - this is shown by the "step" curve in the 280 - 400 second interval in sort's CPU utilization trace (Fig. 1). That figure shows high utilization at the beginning of the interval as all cores are sorting small lists in parallel and low utilization as less and less threads are merging data.

Merge sort works well for small volumes of key-value pairs because merging small lists is fast. Unfortunately, merge sort suffers with large lists because the algorithm ends up comparing many keys in each round. This latency is multiplied with each round and as a result, most of the time is spent re-scanning the keys.

In scale-out architectures, this merge phase can be sped up by leveraging extra resources on idle nodes. For example, in Hadoop, daemons merge values in the background while other data is transferred in the reduce phase. Other studies on scale-

TABLE I: These additions to the Phoenix++ runtime implement the ingest chunk pipeline. API/callback functions are used to interact with the application. Runtime functions let the ingest chunk library access the internal runtime data structures.

function name	function call type	# of times called	function call purpose & description
`run_ingestMR()`	API	once	launch the SupMR runtime
`run_mappers()`	runtime	multiple	initialize data structs; determine # of mappers/reducers; launch mappers
`run_reducers()`	runtime	once	launch reducers
`set_data()`	callback	multiple	pass the chunk length and ingest chunk pointer back to the application

up MapReduce looked at scaling threads instead of the actual data, so this merge bottleneck has gone unnoticed. Furthermore, scale-out Hadoop can be modified to use custom sort functions[1].

To minimize the excessive re-scanning, SupMR uses OpenMP's sort version. OpenMP's sorting algorithm uses p-way merging [9], an algorithm that merges N ordered lists into a single ordered array using p processors.

V. Implementation

To implement SupMR, we built on the Phoenix++ [10] system. The Phoenix++ runtime creates and maintains all the data structures, schedules all map, reduce, and merge tasks, and pushes data through the system. The Phoenix++ application allocates memory for the data, ingests the data, and provides the runtime with map/reduce callback functions and pointers to data in memory. We had to slightly modify the applications but the runtime is still backwards compatible.

A. Additional Functions & Data Structures

The SupMR modifications are summarized in Table I. The API library call, `run_ingestMR()`, is part of the runtime and implements the ingest chunk pipeline. The SupMR runtime launches in exactly the same way as the original library with a few additional chunk-related parameters. The two runtime functions are wrappers for the original map/reduce calls. Their purpose is to provide access to the runtime data structures for our external ingest chunk library. The map function wrapper, `run_mappers()`, sets the number of mapper/reducer threads and initializes the persistent container. The reduce function wrapper, `run_reducers()`, is the same as the regular internal reduce function. Finally, to achieve the optimized sort and p-way merge, we use OpenMP's parallel sort. We disable the Phoenix++ runtime sort and manually call `gnu_parallel::sort()`.

The ingest chunk management structures and functions are linked into the runtime from an external library. This library contains the chunk struct, a struct for passing around the job state, and functions for reading chunks and locating chunk boundaries. This is where the runtime manages ingest chunks using the user-defined ingest chunking strategy and ingest chunk size.

The original runtime forces the application to manage memory and ingest the data. Since the memory is part of the application's address space, execution of the callback functions does not violate process address space semantics. SupMR manages all the memory allocation to reduce redundant code and enhance backwards compatibility. With this design, applications need not re-implement the ingest chunk pipeline.

B. Application Modifications

In addition to the Phoenix++ application responsibilities, the SupMR applications need to (1) allocate memory for the chunk data structures and (2) define the `set_data()` callback function. Allocating the chunk data structures ensures that all pointers to data can still be operated on by the application map/reduce functions. The callback function, `set_data()`, passes the chunk information back to the application. This lets the runtime dictate which part of memory (i.e. which ingest chunk) the application map/reduce callbacks should operate on.

The Phoenix++ runtime generalizes to a range of applications by supporting three different containers for storing intermediate key-value pairs. These containers let the application choose the best container given the workload. The default container is a hash container, where each key is hashed to a cell in an array. This works well for applications like word

[1]As of Hadoop version 2.2, the developer can specify custom functions using the Pluggable Shuffle and Sort capabilities.

count that have many pairs with the same key because the large input set is transformed into a much smaller intermediate set.

Unfortunately, the hash container is a poor data structure for applications like sort, where the large input set is transformed to an equal sized intermediate set. The mappers must check the container for the key before insertion, which is a problem for a large number of key-value pairs. Furthermore, after the map phase, the container has unique key values all inhabiting the same cell. When the reducers are called, they must iterate over every key, needlessly sweeping the array to handle different keys. Because the sort application has unique keys, SupMR uses Phoenix's unlocked storage, which allows all threads to write to a single array without synchronization. Each mapper outputs to its key range in the array and each reducer operates only on its key range.

VI. Results

To demonstrate the benefits of SupMR, we choose the word count and sort applications as benchmarks because these applications represent different spectrums of the application space; they experience different speedups when mitigating the ingest or merge phases. We choose input sizes that showed extreme performance degradation in previous work [7]; 155GB for word count and 60GB for sort. Although we tested on traditional multi-core servers with a few target applications, we feel that the techniques can still be extended to other large-scale deployments.

A. Measurements and Experimental Setup

To measure execution times we use the Phoenix++ internal timing functions[2]. The programmer can start/stop a timer and print the elapsed time with microsecond granularity. Each experiment is run 3 times and the average is taken. The CPU utilization is collected with `collectl`, a daemon that can monitor all CPUs for a specified amount of time. Each resource utilization graph traces one run, since we care more about the behavior rather than the exact timing measurements.

We ran the following experiments on a scale-up Red Hat Enterprise Linux 6 system with 384GB of RAM and 2 8-core processors with hyperthreading enabled (32 hardware contexts). The system has 3 data HDDs in a RAID-0 configuration and a separate HDD houses the operating system. The RAID-0 device reports a read speed of 384 MB/s maximum.

[2]These functions use the Linux time libraries in `time.h`.

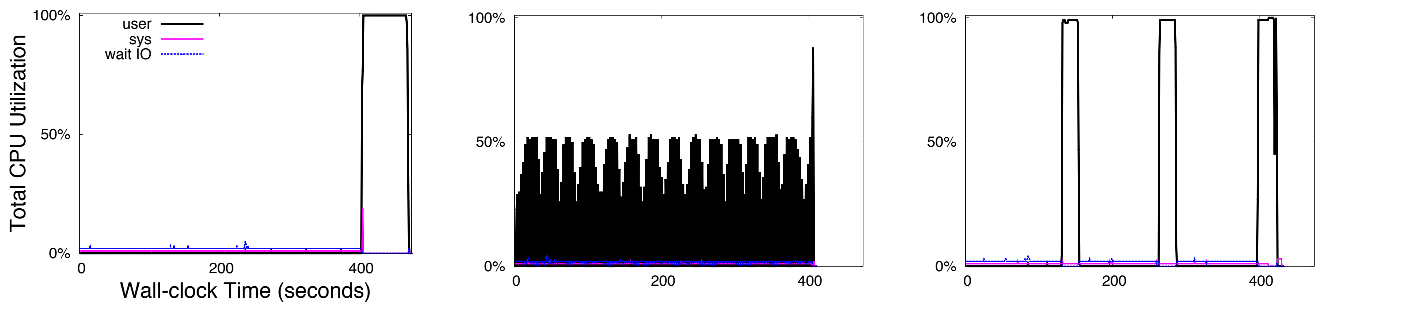
TABLE II: Execution times of the different job phases show how SupMR mitigates the ingest and merge bottlenecks. Word count operates on 155GB of data and sort operates on 60GB of data.
***Note: rows = chunk sizes, columns = job phase**

	total	read	map	reduce	merge
Word Count: mitigate ingest bottleneck					
none	471.75s	403.90s	67.41s	0.03s	0.01s
1GB	407.58s	406.14s		1.08s	0.01s
50GB	429.76s	423.51s		0.08s	0.01s
Sort: mitigate merge bottleneck					
none	397.31s	182.78s	6.33s	7.72s	191.23s
1GB	272.58s	196.86s		9.04s	61.14s

B. Timing Analysis

Our timing analysis in Table II provides raw performance numbers to support our claims that SupMR effectively mitigates the ingest and merge bottlenecks. The rows of that table are the runtimes with different chunk sizes, with "none" corresponding to the original runtime. The columns of that table show the total runtime of the job (total) and a breakdown of the job into the four different MapReduce phases (ingest, map, reduce, merge). Experiments with different chunk sizes indicate that 1GB and 50GB are good minimum and maximum chunk sizes, respectively. This concept is explained more fully in the next section. For the total job time, the timing breakdown shows performance improvements for SupMR. The word count application experiences a speedup between 1.16× and 1.10× over the original runtime, depending on the chunk size. The sort application experiences a 1.46× speedup over the original runtime. All job execution times do not add up to the total execution time because we do not list the cleanup or setup times.

For the four job phases, the timing breakdown shows how SupMR (1) parallelizes the ingest/map phases for word count and (2) improves the merge phase for sort. In the original runtime, there is a read and a map phase but in SupMR, these phases are combined because of the ingest chunk pipeline. Word count experiences a speedup between 1.16× and 1.12× in these phases; it is a better speedup than sort because word count has a more complicated



(a) Word count without ingest chunks. **(b)** Word count, chunk size = 1GB. **(c)** Word count, chunk size = 50GB.

Fig. 5: For word count, the CPU utilization shows a long ingest bottleneck without ingest chunks (a) and improved performance and utilization with ingest chunks (b and c). Smaller ingest chunks have better performance during the ingest/map phase ($1.16\times$ over the original runtime), while larger ingest chunks ($1.11\times$ over the original runtime) have less thread management overheads.

map phase, namely checking a container before inserting a key. This results in a longer map phase, which in turn allows more of the job execution to be parallelized with ingest.

SupMR's sort achieves a $3.12\times$ speedup in the merge phase. This speedup is better than word count's speedup because sort manages more key-value pairs, resulting in a longer merge phase. The benefit of the ingest chunk pipeline depends on the map phase execution time (i.e. how map-intensive the application is). The benefit of the sort modification depends on the merge phase execution time (i.e. # of key-value pairs).

Conclusion 1: the benefit of these modifications depends on the complexity of the individual job phases.

C. Resource Usage Analysis

The resource usage analysis reveals important properties about SupMR's ingest chunk pipeline and p-way merge. In this section, we also draw conclusions on the optimal ingest chunk size and the relative job phase execution times.

1) Word count - efficient ingest: All three graphs in Fig. 5 show the total CPU utilization for the word count application and demonstrate how SupMR (Figures 5b and 5c) achieves better utilization than the original runtime (Fig. 5a). The large spike in Fig. 5a encapsulates all compute phases (map, reduce, merge), while the dense spikes in Figures 5b and 5c show how ingest chunks are overlapped with the map phase.

Experiments with different ingest chunk sizes reveal that small chunks have higher utilization and better performance. Fig. 5b shows small (1GB) chunk sizes and Fig. 5c shows large (50GB) chunk sizes. Large chunks produce sparse, well-defined utilization spikes, indicating low overall utilization and minimal thread overhead. Small chunks produce dense utilization spikes because the mapper threads finish faster[3]. This shows that, as we decrease the chunk size, the number of map/ingest rounds increases, the time to ingest a chunk decreases, and the CPU utilization increases.

For the highest performance, the chunks should be small. Of course, small ingest chunk sizes also have disadvantages, such as high energy consumption. For example, in our system more map/ingest rounds incur repetitive thread operations. This leads to long periods of very high CPU utilizations and stresses the thread library with unnecessary work. During our experiments, CPU heat thresholds were occasionally breached leading to throttling. Also, increasing the CPU utilization decreases the availability of the system, as it limits the performance of other jobs.

Conclusion 2: the benefit of the ingest chunk pipeline depends on the ingest chunk size.

2) Sort - efficient merge: The original runtime shown in Fig. 1 shows poor utilization and performance in the ingest and merge phases; we focus on the merge phase, since speedup of the ingest phase is limited for sort. For the original runtime, the utilization starts high, at 100%, and progressively

[3]We believe that these intervals reach 100% CPU utilization but the sampling interval of our measurement tool is too large.

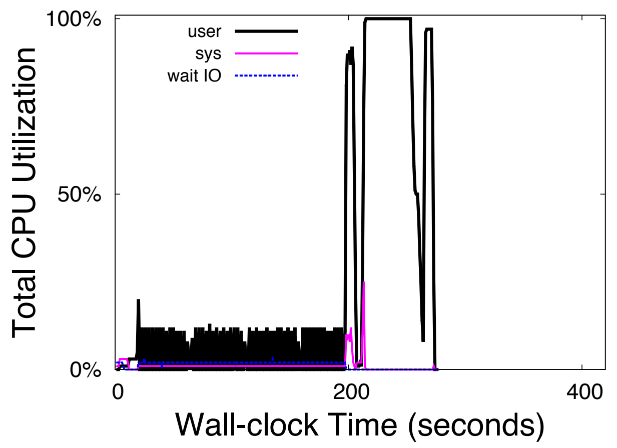


Fig. 6: The CPU utilization for sort on SupMR does not incur the same merge bottleneck as the original runtime in Fig. 1. SupMR uses OpenMP's p-way merge implementation.

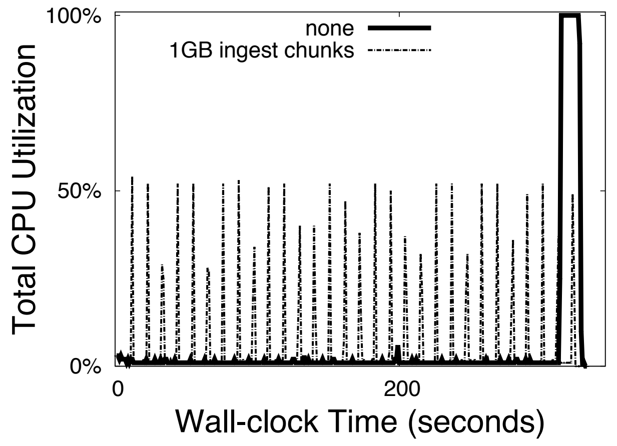


Fig. 7: The CPU utilization for word count on SupMR with a 32-node HDFS storage system shows high utilization but only a 7 second speedup.

degrades to lower utilization.

Fig. 6 shows that SupMR achieves higher utilization and less key scans in the merge phase, resulting in better performance. We see $3.13\times$ speedup in the merge phase because SupMR handles many key-value pairs more efficiently than the original runtime by avoiding multiple merge steps and thread operations. There is only one merge round because p-way merge merges N ordered lists into a single ordered array and the CPU utilization is high because p-way merge parallelizes computation more efficiently while using less total threads.

Conclusion 3: the benefit of the sort modification depends on the number of merge rounds it avoids.

3) A Case Study - ingest chunks on HDFS: As a proof of concept, we show how SupMR can benefit one of the scenarios outlined in the introduction: a scale-up computation framework using a distributed file system as primary storage. We run HDFS on our 32 node scale-out system that is connected with 1Gbit ethernet behind one link. We use the `libhdfs` library to ingest data from the distributed file system directly into memory. For the original runtime, we copy 30GB of data from all the nodes onto one node and then start the word count computation. For SupMR, the runtime copies the data from all the nodes in parallel with the computation.

Fig. 7, which compares the original runtime to SupMR, shows that SupMR achieves high CPU utilization during the ingest phase, proving high compute activity and map task overlay. Unfortunately, most of the utilization is from starting parallel resources - the actual speedup is minimal because the map phase is such a small fraction of the total job time. The longer the ingest phase, the smaller the map phase becomes relative to the total job time. With a small map phase, there is less opportunity to overlay the map computation with ingest. As a result, we only see a 7 second speedup, despite achieving higher utilization. A job with a longer and more complicated map phase would achieve better speedup.

Conclusion 4: the benefits of SupMR depend on the relative execution time of the job phases.

VII. Related Work

To mitigate the ingest and merge bottlenecks, we use techniques from existing big data runtimes, distributed storage, and scale-out systems. Traditional distributed computation lets the user specify the parallel tasks while the runtime handles the communication and load distribution. The community quickly learned that MapReduce is not the silver-bullet for all large scale computation and began tweaking the model to help move data within the computation.

Runtime modifications were made to accommodate iterative tasks with HaLoop [11], Twister [8], and CGL-MapReduce [12]. Other work, most notably streaming-based [13] and in-memory [14] architectures also addressed the problem of moving data through the runtime.

These models for scale-out address the data movement problem within a computation by attempting to leverage in-memory storage to achieve fine-grained updates to mutable states (i.e. shared memory, key-value stores, database techniques, etc.). Although SupMR leverages many of these concepts (data caching, persistent data structures, multiple map/reduce rounds), we are attacking a different part of the data flow - the pre-compute ingest bottleneck. We do not try to pass data to different parts of the system; instead we work on getting the data into the system.

Caching on scale-out systems addresses the data movement problem within storage tiers and streams data into the system, like SupMR. MixApart [3] introduces a cache layer so that compute tasks can overlay with data ingest. Filtering in Hadoop [15] can assist caching by reducing the working set of the job, resulting in less data transmitted on the wire. SupMR adopts many of these caching techniques in the double-buffering scheme and applies them to scale-up computation frameworks.

VIII. Conclusions and Future Work

The ingest and merge bottlenecks plague many large scale-up systems and reducing their effect is important for performance. We focus on the MapReduce model and our ingest chunk pipeline and merge runtime modification show between 50 - 100% more CPU utilization and speedups between $1.16\times$ and $3.13\times$ for the ingest and merge phases. These speedups result in $1.10\times$ - $1.46\times$ time-to-result speedups. We also identify utilization and energy consumption as significant factors in comparing this approach to an "equivalent" scale-out implementation. Integrating functionality for determining (1) the optimal chunk size and (2) the optimal runtime parameters could improve the ingest/map phases but are left as future work.

Acknowledgements

We thank all of our anonymous reviewers for their helpful suggestions, especially in regards to the significance of this work for HPC.

References

[1] J. Dean and S. Ghemawat, "MapReduce: Simplified Data Processing on Large Clusters," in *Proceedings of the 6th USENIX Symposium on Operarting Systems Design & Implementation*, ser. OSDI'04, 2004.

[2] R. Appuswamy, C. Gkantsidis, D. Narayanan, O. Hodson, and A. Rowstron, "Scale-up vs Scale-out for Hadoop: Time to Rethink?" in *Proceedings of the 4th ACM Symposium on Cloud Computing*, ser. SOCC'13, 2013.

[3] M. Mihailescu, G. Soundararajan, and C. Amza, "MixApart: Decoupled Analytics for Shared Storage Systems," in *Proceedings of the 4th USENIX Conference on Hot Topics in Storage and File Systems*, ser. HotStorage'12, 2012.

[4] C. Ranger, R. Raghuraman, A. Penmetsa, G. Bradski, and C. Kozyrakis, "Evaluating Mapreduce for Multi-core and Multiprocessor Systems," in *Proceedings of the 13th IEEE International Symposium on High Performance Computer Architecture*, ser. HPCA'07, 2007.

[5] OpenMP Architecture Review Board, "OpenMP Application Program Interface Version 3.0," accessed 08/09/2012, http://www.openmp.org/mp-documents/spec30.pdf.

[6] M. M. Rafique, B. Rose, A. R. Butt, and D. S. Nikolopoulos, "Supporting MapReduce on Large-scale Asymmetric Multicore Clusters," *SIGOPS Operating Systems Review*, 2009.

[7] M. Sevilla, I. Nassi, K. Ioannidou, S. Brandt, and C. Maltzahn, "A Framework for an In-depth Comparison of Scale-out and Scale-up," in *Proceedings of the 2nd International Workshop on Data Intensive Scalable Computing*, ser. DISCS'13, 2013.

[8] J. Ekanayake, H. Li, B. Zhang, T. Gunarathne, S.-H. Bae, J. Qiu, and G. Fox, "Twister: A Runtime for Iterative MapReduce," in *Proceedings of the 19th ACM International Symposium on High Performance Distributed Computing*, ser. HPDC'10, 2010.

[9] B. Salzberg, "Merging Sorted Runs Using Large Main Memory," *Acta Informatica*, vol. 27, no. 3, pp. 195–215, 1989.

[10] J. Talbot, R. M. Yoo, and C. Kozyrakis, "Phoenix++: Modular MapReduce for Shared-Memory Systems," in *Proceedings of the 2nd ACM International Workshop on MapReduce and its Applications*, ser. MapReduce'11, 2011.

[11] Y. Bu, B. Howe, M. Balazinska, and M. D. Ernst, "HaLoop: Efficient Iterative Data Processing on Large Clusters," *Proceedings of the VLDB Endowment*, vol. 3, no. 1-2, 2010.

[12] J. Ekanayake, S. Pallickara, and G. Fox, "MapReduce for Data Intensive Scientific Analyses," in *Proceedings of the 4th IEEE International Conference on eScience*, ser. ESCIENCE'08, 2008.

[13] L. Neumeyer, B. Robbins, A. Nair, and A. Kesari, "S4: Distributed Stream Computing Platform," in *Proceedings of the 10th IEEE International Conference on Data Mining Workshops*, ser. ICDMW'10, 2010.

[14] M. Zaharia, M. Chowdhury, T. Das, A. Dave, J. Ma, M. McCauley, M. J. Franklin, S. Shenker, and I. Stoica, "Resilient Distributed Datasets: A Fault-Tolerant Abstraction for In-Memory Cluster Computing," in *Proceedings of the 9th USENIX Conference on Networked Systems Design & Implementation*, ser. NSDI'12, 2012.

[15] C. Gkantsidis, D. Vytiniotis, O. Hodson, D. Narayanan, F. Dinu, and A. Rowstron, "Rhea: Automatic Filtering for Unstructured Cloud Storage," in *Proceedings of the 10th USENIX Conference on Networked Systems Design and Implementation*, ser. NSDI'13, 2013.