MMR: A Platform for Large-Scale Forensic Computing

Vassil Roussev, Liqiang Wang, Golden G. Richard III, Lodovico Marziale
Department of Computer Science
University of New Orleans
New Orleans, Louisiana 70148, USA

vassil,lwang,golden,vico@cs.uno.edu

Chapter 1

MMR: A PLATFORM FOR LARGE-SCALE FORENSIC COMPUTING

Middleware Support for MapReduce Processing

Abstract The timely processing of large-scale digital forensic targets demands the empoyment of large-scale distributed resources, as well as the flexibility to customize the processing performed on the target. We present*MMR*– a new, open implementation of the MapReduce processing model, which significantly outperforms prior work on typical forensic tasks. It demonstrates linear scaling for CPU-intensive processing and even super-linear scaling for indexing-related workloads.

Keywords: Digital forensics, mapreduce, large-scale forensics, mmr, mpi mapreduce, cluster computing

1. Introduction

According to FBI statistics [3], the size of the average case has been growing at an average annual rate of 35from 83 GB in 2003 to 277 GB in 2007. With capacity growth outpacing both bandwidth and latency improvements [8], forensic targets are not only getting bigger in absolute terms, but they are also growing larger relative to our capabilties to process them on time. With the price of 1.5TB hard drives dropping below $200, it is difficult to overstate the urgent need to develop scalable forensic solutions that can match the explosive growth in target size with adequate computational resources.

The problem of scale is certainly not unique to the field of digital forensics, however researchers and developers have been relatively slow to recognize and address it. Generally, there are three possible approaches to accommodate more processing in the same amount of time: a) improve algorithms and tools; b) support the utilization of more hardware resources; and c) support human collaboration. These are largely indepent of each other and support large-scale forensics in complimentary ways: the first approach allows more efficient use of machine resources, the second allows more machine resources to be deployed, the third one allows more human resources to be efficienty deployed in response to a big challenge. In all likelihood, next generation forensic tools will have to support all three approaches in order to be effective.

The focus of this paper is exclusively on supporting the use of commodity distributed computational resources to improve the turnaround time of forensic investigations. Unfortunately, this topic has received very little attention despite the fact that is a rather obvious means of dealing with problems of scale.

1.1 Related Work

The earliest discussion appears to be [10], where an early prototype and experiments demonstrate the great potential of using cluster resources to speed up typical forensic functions. Specifically, it was demonstrated that it is possible to achieve linear speedup, that is, speedup proportional to the number of processors/cores in the system. Further, for memory-constrained functions, it is possible to achieve *super-linear* speedup due to the fact that a larger fraction of the data can be cached in memory.

ForNet [11] is another research effort aimed at distributed forensics which is mostly concerned with the distributed collection and query of network evidence, which is simply a different concern from ours. [7] is an effort to take advantage of new hardware and software development–the use of highly-parallel graphics processing units (GPU) for general-purpose computing. The approach has the same goals as this work–bring more hardware resources to the investigation process–and is inherently complimentary to it. Obviously, using the CPU and GPU resources on multiple machines will bring even more speedup than using a single one.

Recently, commercial vendors, such as AccessData, are starting to include support for multi-core processors as part of their main product line *FTK* [1], and provide limited distributed capabilities as part of specialized tools, such as password cracking [2]. At the same time, *FTK* includes a heavyweight database back-end, which is undoubtedly capable of managing terabytes of data but also uses precious CPU and memory resources necessary for executing core forensic functions.

1.2 Background: MapReduce

MapReduce [4], is a new distributed programming paradigm, developed by *Google*, is aimed at simplifying the development of scalable, massively-parallel applications that process terabytes of data on large commodity clusters. The goal is to make the development of such application easy for programmers with no prior experience in distributed computing. Programs written in this style are automatically executed in parallel on as large a cluster as necessary. All the I/O operations, distribution, replication, synchronization, remote communication, scheduling, and fault-tolerance are done without any further input from the programmer who is free to focus on application logic. (We defer the technical discussion of the MapReduce model to the next section.) Arguably, essential ideas behind MapReduce are recognizable in much earlier work on functional programming, however, the idea of using *map* and *reduce* functions as the sole means of specifying parallel computation, as well as the robust implementation on a very large scale are certainly novel. After the early success of the platform, *Google* moved all of their search-related functions to the MapReduce platform.

Pheonix [9] is an open-source research protype, which demonstrates the viability of the MapReduce model for shared memory multi-processor/ multi-core systems. It has demonstrated close to linear speedup for workloads that we believe are very relevant to forensics applications.

Hadoop [2] was developed as an open-source *Java* implementation of the MapReduce programming model and has been adopted by large companies like *Yahoo!*, *Amazon*, and *IBM.* The National Science Foundation has partnered with Google and IBM to create the Cluster Exploratory (CluE), which provides a cluster of 1,600 processors to enable scientist to easily create new types of scientific applications using the *Hadoop* platform. In other words, there is plenty of evidence the MapReduce model, while not the answer to all problems of distributed processing, fits quite well with the types tasks required in forensic application–string operations, image processing, statistical analysis, etc.

The obvious question is: Is *Hadoop* an appropriate platform to implement scalable forensic processing? One potential concern is that a Java-based implementation is inherently not as efficient as as Google's C-based one. Another one is that the *Hadoop File System* (HDFS), which is implemented as an abstraction layer on top of the regular file system, is not be nearly as efficient as the original *Google File System* [5]. From a cost perspective, it may make perfect sense for a large company to pay a performance penalty if that would lead to a simplification of its code base and make it easier to maintain. However, the average forensic lab has very modest compute resources relative to the large data centers available to most the companies adopting *Hadoop*.

2. MMR: MPI MapReduce

Based on the published results from prior work and our own experience, we can confidently conclude that MapReduce is a very suitable conceptual model for describing forensic processing. From a practioner's perspective, the next relevant question is: Is the performance penalty too big to be practical for use in a realistic forensic lab? If yes, can we build a more suitable implementation? To answer these questions, we developed our own prototype version, called *MPI MapReduce*, or *MMR* for short, and evaluated its performance relative to *Hadoop*.

Our work builds on the *Phoenix* shared-memory imlementation of MapReduce and the *MPI* (Message Passing Interface) standard [3] for distributed communication. The essential idea is to start out with the shared-memory implementation and extend it with the ability to spread the computation to multiple nodes. MPI has been around for a while and has found wide use on scientific and other high-performance applications. MPI was designed with flexibility in mind and does not prescribe any particular model of distributed computation. While this is certainly an advantage in the general case, it is also a drawback as it requires fairly good understanding of distributed programming on part of the developer who must explicitly manage all communication and synchronization among distributed processes.

Our goal was to hide MPI behind the scenes and to create a middleware platform that allows programmers to focus entirely on expressing aplication logic and not worry about scaling up the computation. This becomes possible only because we assume the MapReduce model, which allows us to automatically manage communication and synchronization across tasks. We should also mention that by choosing MPI, we also simplify cluster management. For example, one could easily (within a day) create a dual-boot setup for a set of workstations in the lab that could be used as a cluster to run long processing jobs overnight. Modern MPI distributions have become quite user-friendly and should not present any major issues for administrators.

2.1 Conceptual Model

To avoid repetition, we provide only a brief description of the general MapReduce model following the discussion in [4]; for a more detailed discussion, please refer to the original paper.

The MapReduce computation takes a set of input key/value pairs, and produces a set of output key/value pairs. The developer expressed the computation as two programmer-provided functions–*map* and *reduce*. The former, takes an input pair and produces a set of intermediate key/value pairs. The run-time engine automatically groups together all intermediate values associated with the same intermediate key I and passes them on to the *reduce*. The *reduce* function accepts an intermediate key I and a set of values for that key, and merges together these values (however it deems necessary) to form another (possibly smaller) set of values. Typically, just zero or one output value is produced per *reduce* invocation. The I values are supplied to the *reduce* function via an *iterator*, which allow for arbitrarily large lists of values to be passed on.

As an illustration, consider the `wordcount` example–given a set of text documents, count the number of occurences for each word. In this case, the *map* function simply uses the word as a key to construct pairs of the form (*word*, 1). Thus, if there are n distinct words in the document, the run-time will form n distinct lists, and will feed them to n different instances of the *reduce* function. The *reduce* function needs to simply count the number of elements in its argument list and output that as the result. Below is a psuedo-code solution from [4]:

```
map(String key, String value):
    // key: document name
    // value: document contents
    for each word w in value:
        EmitIntermediate(w, "1");

reduce(String key, Iterator values):
    // key: a word
    // values: a list of counts
    int result = 0;
    for each v in values:
        result += ParseInt(v);
    Emit(AsString(result));
```

Without getting into a detailed discussion, we should point out the natural sources of data parallelism in this solution. For the *map* function, we can create as many independent instances (each of which executable in parallel) as we have documents. If that is not enough, we could also split up the documents themselves. The *reduce* computation is also readily parallelizable–we can have as many distinct instances as we have words. In general, we would expect that different instances of the functions to take different amounts of time so it is the runtime's job to keep track of partial results and to load balance the cluster. Note that the code written by the programmer is completely free of explicit concerns with regard to the size of the inputs and outputs, and distribution of the computation. Consequently, the system can transparently spread the computation across all available resources. Further, it can schedule redundant task instances to improve fault-tolerance and reduce bottlenecks.

2.2 How *MMR* Works

Figure 2 illustrates the overall data flow in an *MMR* application. At present, we use a single text or binary data file as the input. This is done only to simplify the experimental setup and to isolate file system influence on execution time. The system can just as easily use multiple files as input. A file splitter function splits the input into N equal blocks, where N is the number of machines (nodes) available. In our experimental setup the blocks are created of equal size as the machines we use are identical but the block-splitting can be quite different in a heterogeneous setting.

Each node reads its own block of data and splits it up further into M *chunks* according to the number of mapper threads to be created on each node. Normally, that number would be equal to the level of hardware-supported concurrency, i.e., the number of threads that the hardware can execute in parallel. The chunk size is also related to the amount of the on-board CPU cache–the system makes sure that there is enough room in the cache for all the mappers to load their chunks without interference.

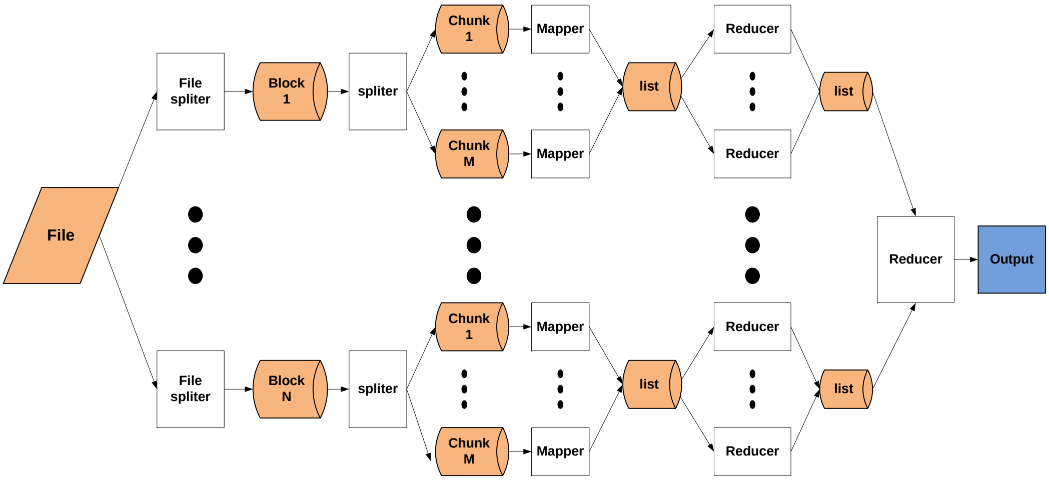


Figure 1. *MMR* data flow

After the threads are created, each thread gets a chunk of data, and the programmer-defined *map* function is called to manipulate that data and to produce key/value pairs. If the programmer has specified reduction function, the results are grouped according to keys and, as soon as all the mapper threads complete their tasks, a number of reducer threads are created to complete the computation with each reducer thread invoking the programmer-defined *reduce* function. After the reduction, each node has a reduced key/value pair list. If all the lists should be merged, each node packs its result data into an data buffer and sends content in the buffer to the master node (node 0). When the master receives data from another node, it uses a similar reduce function to reduce the received key/value pairs with its own result. This procedure repeats until all the data from other nodes are reduced. Finally, the master outputs a complete key/value list as the final result.

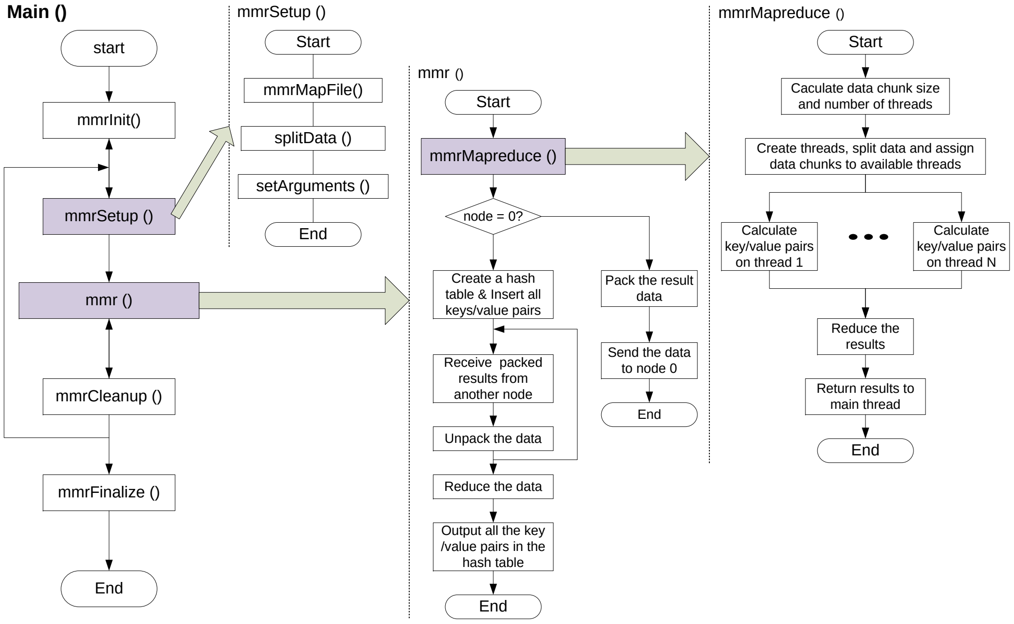

Main ()
start
mmrInit()
mmrSetup ()
mmr ()
mmrCleanup ()
mmrFinalize ()
End
mmrSetup ()
Start
mmrMapFile()
splitData ()
setArguments ()
End
mmr ()
Start
mmrMapreduce ()
node = 0?
Create a hash table & Insert all keys/value pairs
Receive packed results from another node
Unpack the data
Reduce the data
Output all the key /value pairs in the hash table
End
Pack the result data
Send the data to node 0
End
mmrMapreduce ()
Start
Caculate data chunk size and number of threads
Create threads, split data and assign data chunks to available threads
Calculate key/value pairs on thread 1
Calculate key/value pairs on thread N
Reduce the results
Return results to main thread
End

Figure 2. MMR flowchart

Figure 2 illustrated the *MMR* flow of execution on each node and the basic steps a developer needs to perform. All the functions with the ”mmr” prefix are *MMR* API functions supplied by the infrastructure. The first step in this process is to invoke the system-provided *mmrInit()* function, which performs a set of initialization steps, including MPI initialization. Next, the application invokes *mmrSetup()* to specify necessary arguments such as file name, unit size, and number of map/reduce threads on each node (optional), as well a list of application-defined functions such as key comparison, map and reduce functions.

Many of the mandatory arguments have default values to simplify routine processing. By default, *mmrSetup()* automatically opens and memory-maps the specified files, although this behavior can be overriden if the application needs access to the raw data (e.g., read header information). After the data is mapped into memory, the *splitData()* function calculates the offset and length of the data block on a node, while *setArguments()* sets all the necessary arguments for map, reduce and MPI communication.

After the above initialization steps, the application can calls *mmr()* to launch the actual compuation. The final result list generated on the master node and returned to the application. More complicated processing may require multiple map/reduce rounds of computation. If that is necessary, the applications invokes *mmrCleanup()* to reset buffers and state information. After that, the application can set up and execute another MapReducing (using *mmrSetup()* and *mmr()*.

The mmr() function first calls *mmrMapReduce()* to do map/reduce on a node. If a node is *not* the master, it must pack its result (a list of key/value pairs) into an MPI buffer and sends the content of that buffer to the master node. Since *MMR* does not know the data types of keys and values, the developer must define a functions to pack/unpack the data. This is routine code and a number of default functions are provided for the most frequently used data types. If a node *is* the master, after *mmrMapReduce()* it generates a hash table to store hashed keys. The hash table is used to accelerate the later reduction of values on same keys. The master receives byte streams from each node, unpacks them into keys-value pairs, aggragates them into a list, and returns them to the nodes for the reduction step. If an application does not need each node to send its partial result to the master, it can shortcircuit the computation by calling *mmrMapReduce()* rather than *mmr()*. The *mmrMapReduce()* function does map/reduce on each node and returns a partial list of key/value pairs. If there is no need to further reduce and merge the lists, then each node just uses its own partial list.

2.3 Implementation and Availability

Due to space limitations, a running code example and a full description of the API were not included in this paper. Instead, we created a dedicated web site where readers can download the code and read related how-to documents. The address is: (*anonymized*).

3. Performance Evaluation

The primary purpose of this section is to compare the performance of the *MMR* and *Hadoop*. We also quantify the scalability of *MMR* for a set of applications by comparing it with baseline serial implementations. To provide a fair comparison, we used three *Hadoop* examples as provided by the system developers and wrote functional equivalent in *MMR* . We did not make any changes to the *Hadoop* examples except to add a time stamp to calculate execution time.

3.1 Test environment

MMR has fairly modest hardware requirements: a cluster of networked Linux machines, a central user account management tool, gcc 4.2, ssh, gunmake, OpenMPI, and a file sharing system. Certainly better machines provide more benefits but, as our experiments show, even more modest hardware provides tangible speedup. For our experiments we used two ad-hoc clusters made up of lab workstations. The first one, Cluster #1, consists of 3 Dell dual-core boxes, whereas Cluster #2 has 3 brand new Dell quad-core machines. Table 1 gives the configuration of the machine in the clusters. (Note: the quad-core machines were run

	Cluster #1	Cluster #2
CPU	Intel Core 2 CPU 6400	Intel Core 2 Extreme QX6850
Clock (GHz)	2.13	3.0
Number of cores	2	4
CPU Cache (KB)	2048	2 x 4096 KB
RAM (MB)	2048	2048

Table 1. Hardware configuration

on a 2GB RAM configuration to make the results comparable–normally they have 4GB.) All machines were configured with Ubuntu Linix 8.04 kernel version 2.6.24-19, *Hadoop*, and *MMR* . The network setup included the standarda, built-in Gbit Ethernet NICs and a CISCO 3750 switch.

Note that the *Hadoop* installation uses the *Hadoop Distributed File System* (HDFS), whereas *MMR* just uses NFS. HDFS separates a file into many chunks and distributes them into many nodes. When a file is requested, each node sends its chunks to the destination node. However the NFS file system uses just one file server; when a file is requested, the server has to send the whole file to the destination node. Therefore, in general, the HDFS performance better than NFS, which means the *MMR* implementaion spends more time on opening a file than the comparable *Hadoop* tool. However, for the purposes of our tests, we virtually eliminated such effect by forcing the caching of the files before the timed execution.

We have rewritten the following three Hadoop samples for *MMR* : `wordcount`, `pi-estimator`, and `grep`. The `wordcount` program calculates the number of occurrence of each word in a text file, the `pi-estimator` program calculates an approximation of PI based on the Monte Carlo estimation method. The `grep` program searches for matched lines in a text file based on the regular expression matching. `wordcount` and `grep` are text processing programs, whereas and `pi-estimator`is computationally intensive with virtually no communication and synchronization overhead. The functionally of *Hadoop* grep is weaker than the regular Linix command–when searching a specific word in a file, *Hadoop* grep just returns how many times this specific word appears in the file. However in Linux the grep command returns the line numbers and the whole lines. *MMR* grep can return the line numbers and the whole lines back like the Linux grep. However, to produce apples-to-apples comparison, we modified some *MMR* grep functions and made it return just the counts.

Evidently, for I/O-bound programs, launching more map processes would not increase the performance of either tool. However, for CPU-bound code, launching more map processes than the number of processing units tends to improve the *MMR* performance; for *Hadoop*, that is not the case. We recommend that, when performing computation programs, setting the number of map processes to 3-4 times of the computation nodes; however, when performing file processing programs, setting the number of map processes to the computation nodes.

3.2 Benchmark Execution Times

We tested `wordcount` and `grep` with 10MB, 100MB, 1000MB and 2000MB files, and `pi-estimator`with 12,000 to 1,200,000,000 points. All our testing results are averages over 10 run at the optimal settings (determined empirically), first and last runs are ignored. All experiments are performed on Cluster #2. For `wordcount`, *MMR* is about 15 times faster than *Hadoop* for the large (1GB+) files and about 23 times for the

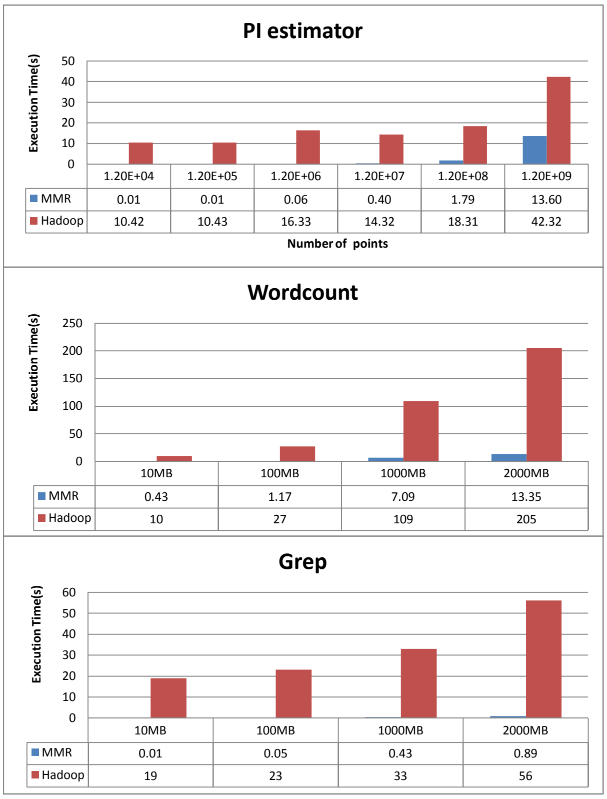


Figure 3. MMR vs Hadoop: execution times for three example applications

small ones. For `grep`, in the worst case MMR, is about 63 times faster than *Hadoop*. For `pi-estimator`, the purely computational workload, *MMR* is just over 3 times faster for the large point set. Overall, it is evident that *Hadoop* has significantly higher startup costs so the times from the longer runs should be considered more representative.

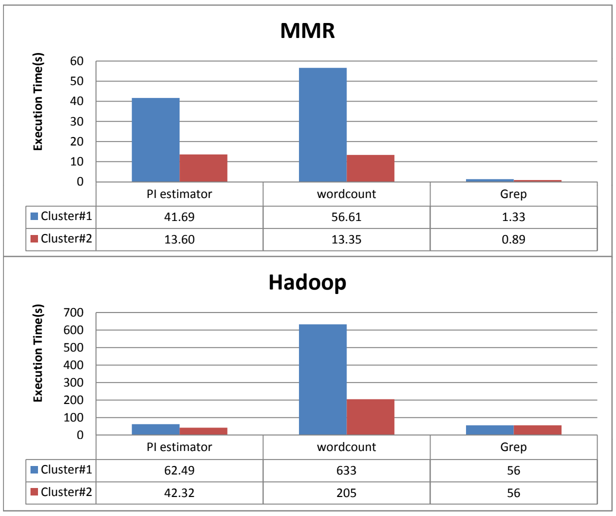
3.3 Scalability

Let us now consider the efficiency with which the two implementations scale up and use available CPU resources. Table 4 compares the execution times for each of the three applications under *MMR* and *Hadoop* on each of the two clusters. `pi-estimator`is a trivially parallelizable application which we would expect to scale up proportionately to hardware improvements of the CPU. We do not expect caching or faster memory to make any difference. Since the two processors are successive designs with virtually the same architecture, it all boils down to number of cores and clock rates. Thus, given twice the number of cores and 50% faster clock, one would expect speedup in the order of 3 times.

Indeed, the actual execution times do show the expected 3 times improvement. By contrast, the *Hadoop* version shows only 50% improvement, which we have no ready explanation for, especially given the 3 times improvement on the `wordcount` benchmark. In `wordcount`, our implementation achieves 4.2x speedup, which exceeds the pure CPU speed up of 3 times due to faster memory access and larger caches. For `grep`, which is a memory-bound application, the speedup is dominated by faster memory access, with minor contribution from the CPU speedup. We could not measure any speedup for the *Hadoop* version.

3.4 Super-linear and sub-linear speedup

So far we have seen that, for CPU-bound applications, *MMR* is able to efficiently take advantage of available CPU cycles and deliver speedup close to the raw hardware improvements. In a realistic setting, we would expect a significant number of applications not to adhere to this model and we now consider the use of *MMR* in such scenarios. Specifically, we compare the speedup of two *MMR* applications–`wordcount` and `bloomfilter`–relative to their serial version, using Cluster #2. In other words, we compare the execution on a single core versus execution on 12 cores.



MMR

	PI estimator	wordcount	Grep
🟦 Cluster#1	41.69	56.61	1.33
🟥 Cluster#2	13.60	13.35	0.89

Hadoop

	PI estimator	wordcount	Grep
🟦 Cluster#1	62.49	633	56
🟥 Cluster#2	42.32	205	56

Figure 4. MMR vs Hadoop: benchmark times on Cluster #1 and Cluster#2

We consider two applications–`wordcount` (same as before) and `bloomfilter`. The `bloomfilter` application hashes a file in 4KB blocks using SHA-1 and inserts them into a Bloom filter [1]. Then, a query file of 1GB is hashed the same way and the filter is queried for matches. If such are found, the hashes triggering them are returned as the result. In the test case, the returned result would be about 16MB. To isolate and understand the effects of networking latency, we created two versions: *MMR* and *MMR Send*. The former completes the computation only sends a total count back, whereas the latter send the actual hash matches back to the master.

Figures 5 and 6 show the performance results. As before, we observe that `wordcount` scales in a super-linear fashion with 15.65x speedup. In fact, the relative speedup factor (speedup/concurrency) is about 1.3, which is very close to the 1.4 number obtained earlier across the two clusters. Again, this is likely due to beneficial caching effects and is

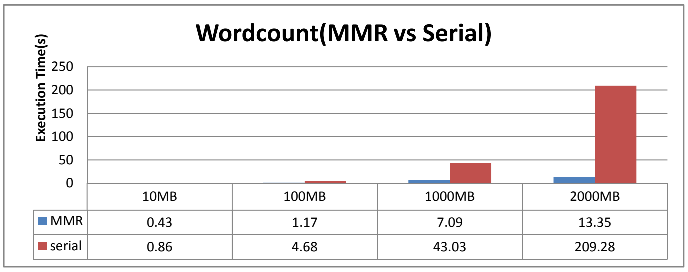


Figure 5. Wordcount: MMR vs serial execution

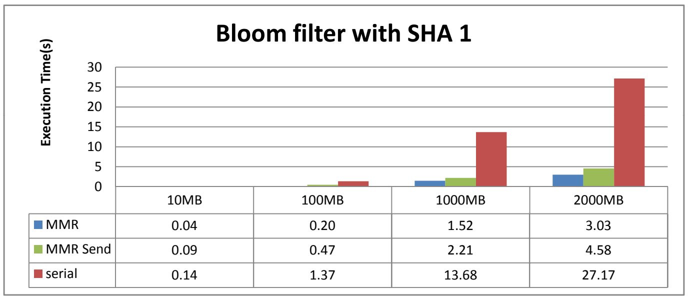


Figure 6. Bloom filter: MMR vs serial execution

great news because the core `wordcount` computation is closely related to the computations performed during indexing.

In the `bloomfilter` case, because the computation is relatively simple and the memory access pattern is random (no cache benefits), memory and I/O latency become the bottleneck factor to speedup. Looking at the 1 and 2GB cases, we see that the *MMR* version gets consistently a speedup factor of 9, whereas the *MMR Send* version achieves only 6. For longer computations, overlapping network communication with computation would help hide some of the latency but, overall, this type of workload cannot be expected to scale as well as the previous one.

4. Conclusions and Future Work

Digital forensic tool developers are in critical need of scalable development platforms that can automatically take advantage of distributed compute resources. The MapReduce model has recently gained popularity and the backing of major industry players as a viable option to quickly and efficiently build scalable distributed applications and has been adopted as a core technology by the top two web search companies–Google and Yahoo!.

In this work, we introduced a new open implementation–*MPI MapReduce* (*MMR*)–which significantly outperforms the leading open-source solution *Hadoop*. We showed that, unlike *Hadoop*, we can efficiently and predictably scale up a MapReduce computation. Specifically, for CPU-bound processing, our platform provides linear scaling with respect to the number and speed of CPUs provided. For the `wordcount` application, which is closely related to common indexing tasks, we demonstrated super-linear speedup, whereas for I/O-bound and memory-constrained tasks the speedup is substantial but sub-linear.

Based on this initial evaluation, and our development experience, we can confidently conclude that the proposed *MMR* platform provides a promising basis for the development of large-scale forensic processing tools. Our next short term goals are to scale up the experiments to tens/hundreds of cores, and develop specific tools that can deal with actual terabyte-size forensic targets in real-time.

References

[1] A. Broder, M. Mitzenmatcher. "Network applications of Bloom filters: a survey". In Annual Allerton Conference on Communication, Control, and Computing, Urbana-Champaign, Illinois, USA, October 2002.

[2] http://hadoop.apache.org/core/.

[3] Federal Bureau of Investigation, Regional Computer Forensics Laboratory Program annual report FY2007, http://www.rcfl.gov.

[4] Dean, J., Ghemawat, S., "MapReduce: Simplified Data Processing on Large Clusters." In Proceedings of the Sixth Symposium on Operating System Design and Implementation, OSDI'04, San Francisco, CA, December, 2004.

[5] Ghemawat, S., Gobioff, H., and Leung, S. "The Google file system." In 19th ACM Symposium on Operating Systems Principles, Proceedings, pages 2943. ACM Press, 2003.

[6] Roussev, V., Richard, G. Breaking the Performance Wall: The Case for Distributed Digital Forensics. In Proceedings of the 2004 Digital Forensics Research Workshop (DFRWS). Aug 2004, Baltimore, MD.

[7] Marziale, L., Richard, G., Roussev, V., "Massive Threading: Using GPU to Increase the Performance of Digital Forensic Tools." In Pro-

ceedings of the 2007 DFRWS Conference Elsevier, pp. 73-81, Pittsburgh, PA, Aug 2007.

[8] D. Patterson, Latency Lags Bandwidth, Communications of the ACM, Vol. 47(10), 2004.

[9] Ranger, C., Raghuraman, R., Penmetsa, A., Bradski, G., and Kozyrakis, C., "Evaluating MapReduce for Multi-core and Multiprocessor Systems." Proceedings of the 13th Intl. Symposium on High-Performance Computer Architecture (HPCA), Phoenix, AZ, February 2007.

[10] Roussev, V., Richard, G. "Breaking the Performance Wall: The Case for Distributed Digital Forensics." In Proceedings of the 2004 Digital Forensics Research Workshop (DFRWS). Aug 2004, Baltimore, MD.

[11] Shanmugasundaram, K., Memon, N., Savant, A., and Bronnimann, H., "ForNet: A Distributed Forensics Network." The Second International Workshop on Mathematical Methods, Models and Architectures for Computer Networks Security, 2003, St. Petersburg, Russia.