Chapter 15

A CLOUD COMPUTING PLATFORM FOR LARGE-SCALE FORENSIC COMPUTING

Vassil Roussev, Liqiang Wang, Golden Richard and Lodovico Marziale

Abstract The timely processing of massive digital forensic collections demands the use of large-scale distributed computing resources and the flexibility to customize the processing performed on the collections. This paper describes MPI MapReduce (MMR), an open implementation of the MapReduce processing model that outperforms traditional forensic computing techniques. MMR provides linear scaling for CPU-intensive processing and super-linear scaling for indexing-related workloads.

Keywords: Cluster computing, large-scale forensics, MapReduce

1. Introduction

According to FBI statistics [4], the size of the average digital forensic case is growing at the rate of 35% per year – from 83 GB in 2003 to 277 GB in 2007. With storage capacity growth outpacing bandwidth and latency improvements [9], forensic collections are not only getting bigger, but are also growing significantly larger relative to the ability to process them in a timely manner. There is an urgent need to develop scalable forensic computing solutions that can match the explosive growth in the size of forensic collections.

The problem of scale is certainly not unique to digital forensics, but forensic researchers have been relatively slow to recognize and address the problem. In general, three approaches are available to increase the processing performed in a fixed amount of time: (i) improving algorithms and tools, (ii) using additional hardware resources, and (iii) facilitating human collaboration. These approaches are mutually independent and support large-scale forensics in complementary ways. The first approach supports the efficient use of machine resources; the second permits additional machine resources to be deployed; the third leverages human expertise in problem solving. In order to cope with future forensic collections, next generation forensic tools will have to incorporate all three approaches. This paper focuses on the second approach – supporting the use of commodity distributed computational resources to speedup forensic investigations.

Current forensic tools generally perform processing functions such as hashing, indexing and feature extraction in a serial manner. The result is that processing time grows as a function of the size of the forensic collection. An attractive long-term solution is to deploy additional computational resources and perform forensic processing in parallel. A parallel approach is more sustainable because storage capacities and CPU processing capabilities increase at approximately the same rate as predicted by Moore's law and as observed by Patterson [9]. In other words, as the average collection size doubles, doubling the amount of computational resources maintains the cost of the expansion constant over time. Furthermore, the widespread adoption of data center technologies is making the logistic and economic aspects of the expansion of computational resources even more favorable.

The data center approach is clearly a long-term goal. The first step is to leverage existing computational resources in a forensic laboratory. For example, a group of hosts in a local area network could be temporarily organized as an *ad hoc* cluster for overnight processing of large forensic collections. The main impediment is the lack of a software infrastructure that enables forensic processing to be scaled seamlessly to the available computing resources. This paper describes a proof-of-concept software infrastructure that could make this vision a reality.

2. Related Work

Early applications of distributed computing in digital forensics demonstrated that it is possible to achieve linear speedup (i.e., speedup proportional to the number of processors/cores) on typical forensic functions [11]. Furthermore, 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.

Several efforts have leveraged distributed computing to address digital forensic problems. ForNet [12] is a well-known project in the area of distributed forensics, which focuses on the distributed collection and querying of network evidence. Marziale and co-workers [7] have leveraged the hardware and software capabilities of graphics processing units (GPUs) for general-purpose computing. Their approach has the same goals as this work and is, in fact, complementary to it. Clearly, using

CPU and GPU resources on multiple machines will contribute to more speedup than using CPUs alone.

Recently, AccessData started including support for multi-core processors under its FTK license. It currently offers limited distributed capabilities for specialized tools (e.g., for password cracking). FTK also includes a database back-end that can manage terabytes of data and utilizes CPU and memory resources to perform core forensic functions.

3. MapReduce

MapReduce [3] is a distributed programming paradigm developed by Google for creating scalable, massively-parallel applications that process terabytes of data using large commodity clusters. Programs written using the MapReduce paradigm can be automatically executed in parallel on clusters of varying size. I/O operations, distribution, replication, synchronization, remote communication, scheduling and fault tolerance are performed without input from the programmer, who is freed to focus on application logic.

After the early success of MapReduce, Google used the paradigm to implement all of its search-related functions. This is significant because of Google's emphasis on information retrieval, which is also at the heart of most forensic processing. Conceptually, information retrieval involves the application of a set of n functions (parsing, string searching, calculating statistics, etc.) to a set of m objects (files). This yields $n \times m$ tasks, which tend to have few, if any, dependencies and can, therefore, be readily executed in parallel.

Phoenix [10] is an open-source prototype that demonstrates the viability of the MapReduce model for shared memory multi-processor/multi-core systems. It provides close to linear speedup for workloads that are relevant to forensic applications (e.g., word counts, reverse indexing and string searches). The main limitation is that it executes on a single machine and has no facilities to scale it to cluster environments.

Hadoop [1] is an open-source Java implementation of the MapReduce model that has been adopted as a foundational technology by large Internet companies such as Yahoo! and Amazon. The National Science Foundation has partnered with Google and IBM to create the Cluster Exploratory (CluE), a cluster of 1,600 processors that enables scientists to create large-scale applications using Hadoop. One concern about the Java-based Hadoop platform is that it is not as efficient as Google's C-based platform. This may not be a significant issue for large deployments, but it can impact efficiency when attempting to utilize relatively small clusters. Another concern is that Hadoop's implementation requires the deployment of the Hadoop File System (HDFS), which is implemented as an abstraction layer on top of the existing file system. This reduces I/O efficiency and complicates access to raw forensic images.

4. MPI MapReduce

MapReduce is a powerful conceptual model for describing typical forensic processing. However, the Hadoop implementation is not efficient enough for deployment in a forensic laboratory environment. To address this issue, we have developed MPI MapReduce (MMR) and use it to demonstrate that the basic building blocks of many forensic tools can be efficiently realized using the MapReduce framework. Note however that an actual tool for use in a forensic laboratory environment would require additional implementation effort.

Our MMR implementation leverages two technologies, the Phoenix shared-memory implementation of MapReduce [10] and the Message Passing Interface (MPI) distributed communication standard [8]. Specifically, it augments the Phoenix shared-memory implementation with MPI to enable computations to be distributed to multiple nodes.

MPI is designed for flexibility and does not prescribe any particular model of distributed computation. While this is generally an advantage, it is also a drawback because developers must possess an understanding of distributed programming and must explicitly manage distributed process communication and synchronization.

MMR addresses this drawback by providing a middleware platform that hides the implementation details of MPI, enabling programmers to focus on application logic and not worry about scaling up computations. Additionally, MMR automatically manages communication and synchronization across tasks. All this is possible because MMR engages MapReduce as its distributed computation model.

4.1 MapReduce Model

The MapReduce computation takes a set of input key/value pairs and produces a set of output key/value pairs. The developer expresses the computation using two functions, `map` and `reduce`. Function `map` takes an input pair and produces a set of intermediate key/value pairs. The runtime engine then automatically groups together all the intermediate values associated with an intermediate key I and passes them to the `reduce` function. The `reduce` function accepts the intermediate key I and a set of values for the key, and uses them to produce another set of values. The I values are supplied to the `reduce` function via an iterator, which allows arbitrarily large lists of values to be passed.

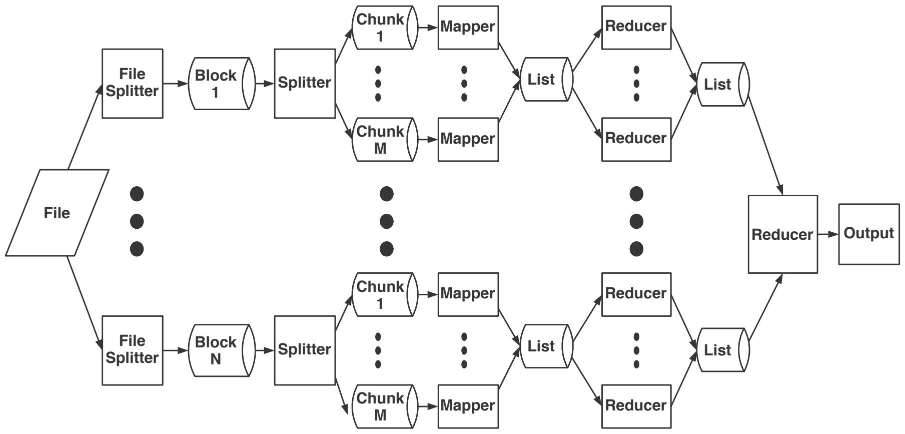


Figure 1. MMR data flow.

As an example, consider the Wordcount problem: given a set of text documents, count the number of occurrences of each word. In this case, the `map` function uses the word as a key to construct pairs of the form (word, 1). For n distinct words in the document, the runtime system creates n pairs and feeds them to n different instances of the `reduce` function. The `reduce` function counts the number of elements in its argument list and outputs the result.

There are several possibilities for data parallelism in this solution. In the case of the `map` function, it is possible to create as many independent instances as there are documents. Large documents could be split into pieces to achieve better load balance and higher levels of concurrency. The `reduce` computation is also parallelizable, allowing for as many distinct instances as there are distinct words. The only major parallelism constraint is that all the `map` instances must complete their execution before the `reduce` step is launched. Note that the programmer does not have to be concerned about the size of the inputs and outputs, and the distribution and synchronization of the computations. These tasks are the responsibility of the runtime environment, which works behind the scenes to allocate the available resources to specific computations.

4.2 MPI MapReduce Details

Figure 1 presents the data flow in an MMR application. A single data file is used as input to simplify the setup and isolate file system influence on execution time. A file splitter function splits the input into N equal blocks, where N is the number of available machines (nodes).

Each node reads its assigned block of data and splits the block into M chunks according to the number of mapper threads to be created at each node. M is typically set to the level of hardware-supported concurrency. This is based on the number of threads that the hardware can execute in parallel and the cache space available for the mappers to load their chunks simultaneously without interference.

After the threads are created, each thread receives a chunk of data and the programmer-defined `map` function is invoked to manipulate the data and produce key/value pairs. If the programmer has specified a reduction function, the results are grouped according to keys and, as soon as the mapper threads complete, 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, which it sends to the master node. The master receives the data and uses a similar `reduce` function to operate on the received key/value pairs and output the final result.

Figure 2 illustrates the MMR flow of execution at each node and the basic steps to be performed by a developer. Note that the functions with an `mmr` prefix are MMR API functions supplied by the infrastructure.

The first step is to invoke the system-provided `mmrInit()` function, which performs a set of initialization steps, including MPI initialization. Next, `mmrSetup()` is invoked to specify arguments such as file name, unit size, number of map/reduce threads at each node and the list of application-defined functions (e.g., key comparison, map and reduce).

Many of the mandatory arguments have default values to simplify routine processing. By default, `Setup()` automatically opens and maps the specified files to memory (this can be overridden if the application needs access to the raw data). After the data is mapped into memory, the `splitData()` function calculates the offset and length of the data block at a node; and `setArguments()` sets all the arguments for map, reduce and MPI communication.

After the initialization steps are done, the application calls `mmr()` to launch the computation. The final result list is generated at the master node and is returned to the application. More complicated processing may require multiple map/reduce rounds. In such an instance, MMR invokes `mmrCleanup()` to reset the buffers and state information, and may setup and execute another MapReduce round using `mmrSetup()` and `mmr()`.

The `mmr()` function invokes `mmrMapReduce()` to perform map/reduce at a node. If the node is not the master, it packs the result (key/value pairs) into an MPI buffer and sends it to the master node. Since MMR does not know the data types of the keys and values, the developer must

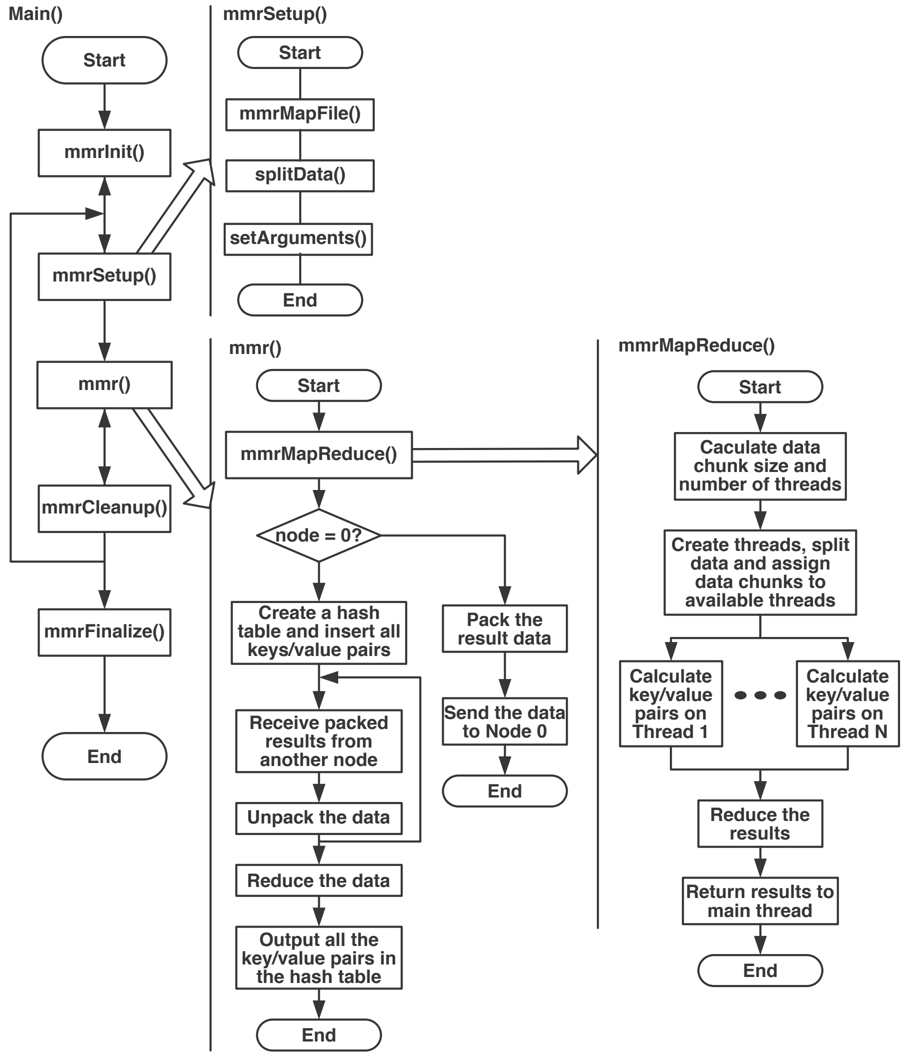

Main()
Start
mmrInit()
mmrSetup()
mmr()
mmrCleanup()
mmrFinalize()
End
mmrSetup()
Start
mmrMapFile()
splitData()
setArguments()
End
mmr()
Start
mmrMapReduce()
node = 0?
Create a hash table and insert all keys/value pairs
Receive packed results from another node
Unpack the data
Reduce the data
Output 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.

write functions to pack and unpack the data. If the node is the master, after `mmrMapReduce()` is invoked, a hash table is generated to store hashed keys. The hash table is used to accelerate the later reduction of values on the same keys. The master receives byte streams from each node, unpacks them into key/value pairs, aggregates 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 short

Table 1. Hardware configuration.

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

circuit the computation by calling `mmrMapReduce()` instead of `mmr()`. The `mmrMapReduce()` function performs map/reduce at 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 uses its own partial list.

To use MMR, the user chooses one of the available nodes as the head node and starts MMR on it. The remaining nodes are rebooted and use PXE (Preboot Execution Environment) [6] to remotely boot from the head node. The head node is responsible for running an NFS server, which stores the forensic collection and output data.

5. Performance Evaluation

This section compares the performance of MMR and Hadoop with respect to two criteria, relative efficiency and scalability. The relative efficiency is evaluated by comparing the performance for three representative applications. Scalability is evaluated by computing the speedup vs. serial execution times and comparing them with the raw increase in computational resources.

5.1 Test Environment

The test environment included a cluster of networked Linux machines configured with a central user account management tool, `gcc 4.2`, `ssh`, `gnumake`, OpenMPI and a network file system. The experiments employed two *ad hoc* clusters of laboratory workstations, Cluster 1 and Cluster 2. Cluster 1 comprised three Dell dual-core machines while Cluster 2 comprised three Dell quad-core machines.

Table 1 presents the configurations of the machines in the clusters. Note that the quad-core machines were run with a 2 GB RAM configuration to permit the comparison of results. All the machines were configured with Ubuntu Linux 8.04 kernel version 2.6.24-19, Hadoop and MMR. The network setup included gigabit Ethernet NICs and a Cisco 3750 switch.

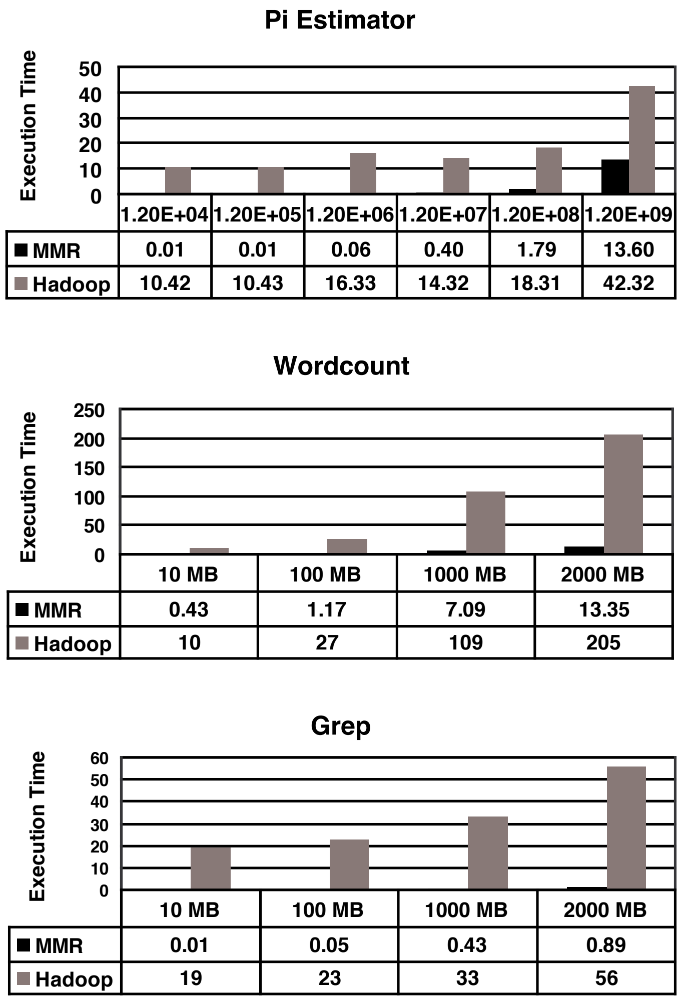
Note that the Hadoop installation uses HDFS whereas MMR uses NFS. HDFS separates a file into chunks and distributes them among nodes. When a file is requested, each node sends its chunks to the destination node. Wherever possible, we discounted I/O times and focused on CPU processing times. However, we found this to be somewhat difficult with Hadoop because of the lack of control over data caching in HDFS.

Our first experiment was designed to compare three Hadoop applications (`wordcount`, `pi-estimator` and `grep`) to their functional equivalents written in MMR. No changes were made to the Hadoop code except to add a timestamp for benchmarking purposes. The three applications are representative of the processing encountered in a typical forensic environment. The `wordcount` program calculates the number of occurrences (frequency) of each word in a text file. Word frequency calculations are of interest because they are the basis for many text indexing algorithms. The `pi-estimator` program calculates an approximation of π using the Monte Carlo estimation method; it involves a pure computational workload with almost no synchronization/communication overhead. Many image processing algorithms are CPU-bound so this test provides a baseline assessment of how well the infrastructure can utilize computational resources. The `grep` program searches for matched lines in a text file based on regular expression matching and returns the line number and the entire line that includes the match. It is one of the most commonly used tools in digital forensics. Note that Hadoop `grep` only returns the number of times the specified string appears in the file, which is weaker than the Linux `grep` command. MMR `grep` can return the line numbers and the entire lines (like the Linux `grep`); however, in order to permit comparisons, it only returns the counts as in the case of Hadoop.

5.2 Benchmark Execution Times

We tested `wordcount` and `grep` with 10 MB, 100 MB, 1,000 MB and 2,000 MB files, and `pi-estimator` with 12,000 to 1,200,000,000 points. The test results were averaged over ten runs with the total number of map/reduce threads equal to the hardware concurrency factor (note that the first and last runs are ignored). All the execution runs were performed on Cluster 2 and the results (in seconds) are shown in Figure 3 (lower values are better).

In the case of `wordcount`, MMR is approximately fifteen times faster than Hadoop for large (1 GB or more) files and approximately 23 times faster for small files. For `grep`, MMR is approximately 63 times faster



	1.20E+04	1.20E+05	1.20E+06	1.20E+07	1.20E+08	1.20E+09
■ MMR	0.01	0.01	0.06	0.40	1.79	13.60
■ Hadoop	10.42	10.43	16.33	14.32	18.31	42.32

	10 MB	100 MB	1000 MB	2000 MB
■ MMR	0.43	1.17	7.09	13.35
■ Hadoop	10	27	109	205

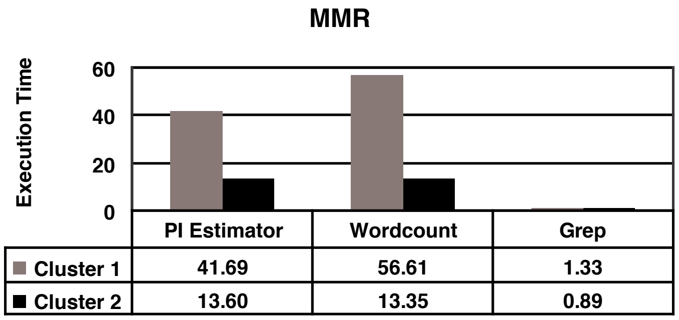
	10 MB	100 MB	1000 MB	2000 MB
■ MMR	0.01	0.05	0.43	0.89
■ Hadoop	19	23	33	56

Figure 3. Execution times for three applications.

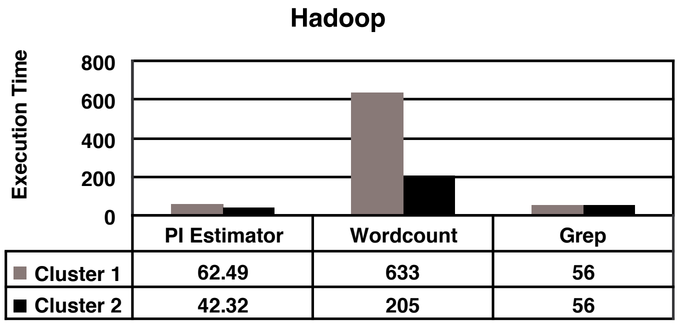
than Hadoop in the worst case. For `pi-estimator` (which measures the purely computational workload), MMR is just over three times faster for the largest point set. Overall, it is evident that Hadoop has higher start-up costs so the times for the longer runs are considered to be more representative.

5.3 Scalability

Figure 4 compares the execution times for the three applications under MMR and Hadoop for each cluster. Since `pi-estimator` is easily



	PI Estimator	Wordcount	Grep
▨ Cluster 1	41.69	56.61	1.33
■ Cluster 2	13.60	13.35	0.89

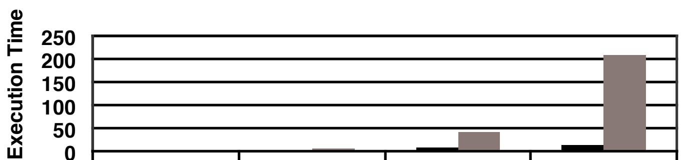


	PI Estimator	Wordcount	Grep
▨ Cluster 1	62.49	633	56
■ Cluster 2	42.32	205	56

Figure 4. Benchmark times on Cluster 1 and Cluster 2.

parallelizable and scales proportionately to hardware improvements of the CPU, one would not expect caching or faster memory to make any difference. Note that the processors in Cluster 2 are newer designs compared with those in Cluster 1, and the expected raw hardware speedup is a function of the differences in the clock rates and the number of cores. Since Cluster 2 has a 50% faster clock and twice the number of cores, one would expect a speedup factor of 3 (i.e., three times faster).

The execution times for MMR show the expected improvement factor of 3 for the `pi-estimator` benchmark. In contrast, the Hadoop version only has an improvement factor of 0.5. This is a curious result for which we have no explanation, especially given the factor of 3 improvement for the `wordcount` benchmark. MMR produces a speedup factor of 4.2 in the case of `wordcount`, which exceeds the pure CPU speedup factor of 3 due to faster memory access and larger caches. In the case of `grep`, which is a memory-bound application, the speedup is dominated by faster memory access with a minor contribution from the CPU speedup. Unfortunately, we could not measure the speedup for the Hadoop version.



	10 MB	100 MB	1000 MB	2000 MB
■ MMR	0.43	1.17	7.09	13.35
▨ Serial	0.86	4.68	43.03	209.28

Figure 5. Wordcount: MMR vs. serial execution.

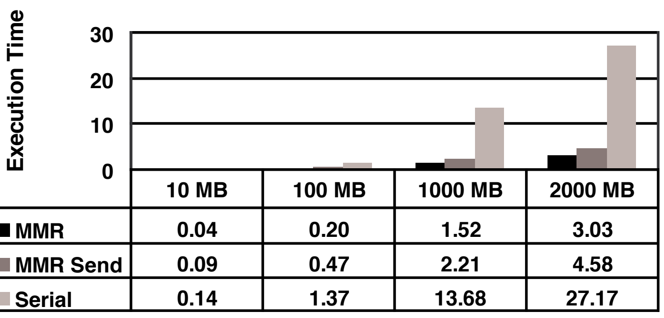
5.4 Super-Linear and Sub-Linear Speedup

In the case of CPU-bound applications, MMR efficiently leverages the available CPU cycles and delivers speedup close to the raw hardware improvements. In a real-world setting, we would expect a significant number of applications not to adhere to this model; consequently, we must consider the use of MMR in such scenarios. Specifically, we use Cluster 2 to compare the speedup of two MMR applications, `wordcount` and `bloomfilter`, relative to their serial versions.

The `bloomfilter` application hashes a file in 4 KB blocks using SHA-1 and inserts them into a Bloom filter [2]. Then, a query file of 1 GB is hashed in the same way and the filter is queried for matches. If matches are found, the hashes triggering them are returned as the result. In the test case, the returned result is about 16 MB. To isolate and understand the effects of networking latency, we created two versions: MMR and MMR Send. MMR completes the computation and only returns a total count. On the other hand, MMR Send passes the actual hash matches to the master node.

Figure 5 compares the results of MMR and serial execution for the `wordcount` application. Note that `wordcount` scales in a super-linear fashion with 15.65 times the speedup. In fact, the relative speedup factor (speedup/concurrency) is approximately 1.3, which is close to the 1.4 value mentioned above.

Figure 6 compares the results of MMR and serial execution for the `bloomfilter` application. In the case of the `bloomfilter` application, because the computation is relatively simple and the memory access pattern is random (no cache benefits), memory and I/O latency become the bottleneck factors for speedup. In the 1 GB and 2 GB experiments, the MMR version achieves a speedup factor of 9 whereas MMR Send has a speedup factor of only 6. For longer computations, overlapping network communication with computation would reduce some of the



	10 MB	100 MB	1000 MB	2000 MB
■ MMR	0.04	0.20	1.52	3.03
▨ MMR Send	0.09	0.47	2.21	4.58
□ Serial	0.14	1.37	13.68	27.17

Figure 6. Bloom filter: MMR vs. serial execution.

latency, but, overall, this type of workload cannot be expected to scale as well as it does for the MMR version.

In summary, the experiments demonstrate that the proof-of-concept implementation of MMR has superior performance relative to Hadoop, the leading open-source implementation. MMR also demonstrates excellent scalability for all three types of workloads, I/O-bound, CPU-bound and memory-bound workloads.

6. Conclusions

Digital forensic tool developers need scalable development platforms that can automatically leverage distributed computing resources. The new MPI MapReduce (MMR) implementation provides this capability while outperforming Hadoop, the leading open-source solution. Unlike Hadoop, MMR efficiently and predictably scales up MapReduce computations. Specifically, for CPU-bound processing, MMR provides linear scaling with respect to the number of CPUs and CPU speed. For common indexing tasks, MMR demonstrates super-linear speedup for common indexing tasks. In the case of I/O-bound and memory-constrained tasks, the speedup is sub-linear but nevertheless substantial.

Our experimental results indicate that MMR provides an attractive platform for developing large-scale forensic processing tools. Our future research will experiment with clusters containing tens to hundreds of cores with the ultimate goal of developing tools that can deal with terabyte-size forensic collections in real time.

References

[1] Apache Software Foundation, Apache Hadoop Core, Forest Hill, Maryland (hadoop.apache.org/core).

[2] A. Broder and M. Mitzenmacher, Network applications of Bloom filters: A survey, *Internet Mathematics*, vol. 1(4), pp. 485–509, 2005.

[3] J. Dean and S. Ghemawat, MapReduce: Simplified data processing on large clusters, *Communications of the ACM*, vol. 51(1), pp. 107–113, 2008.

[4] Federal Bureau of Investigation, Regional Computer Forensics Laboratory (RCFL) Program Annual Report for Fiscal Year 2007, Washington, DC (www.rcfl.gov/downloads/documents/RCFL_Nat_Annual07.pdf), 2008.

[5] S. Ghemawat, H. Gobioff and S. Leung, The Google file system, *Proceedings of the Nineteenth ACM Symposium on Operating Systems Principles*, pp. 29–43, 2003.

[6] Intel Corporation, Preboot Execution Environment (PXE) Specification, Version 2.1, Santa Clara, California (download.intel.com/design/archives/wfm/downloads/pxespec.pdf), 1999.

[7] L. Marziale, G. Richard and V. Roussev, Massive threading: Using GPU to increase the performance of digital forensic tools, *Digital Investigation*, vol. 4(S1), pp. 73–81, 2007.

[8] Message Passing Interface Forum, MPI Forum, Bloomington, Indiana (www.mpi-forum.org).

[9] D. Patterson, Latency lags bandwidth, *Communications of the ACM*, vol. 47(10), pp. 71–75, 2004.

[10] C. Ranger, R. Raghuraman, A. Penmetsa, G. Bradski and C. Kozyrakis, Evaluating MapReduce for multi-core and multiprocessor systems, *Proceedings of the Thirteenth International Symposium on High-Performance Computer Architecture*, pp. 13–24, 2007.

[11] V. Roussev and G. Richard, Breaking the performance wall: The case for distributed digital forensics, *Proceedings of the Fourth Digital Forensic Research Workshop*, 2004.

[12] K. Shanmugasundaram, N. Memon, A. Savant and H. Bronnimann, ForNet: A distributed forensics network, *Proceedings of the Second International Workshop on Mathematical Methods, Models and Architectures for Computer Network Security*, pp. 1–16, 2003.