Comparing Fork/Join and MapReduce

Robert Stewart*[1] and Jeremy Singer†[2]

[1]Mathematical and Computer Sciences, Heriot Watt University, Edinburgh
[2]School of Computing Science, University of Glasgow

Abstract

This paper provides an empirical comparison of fork/join and MapReduce, which are two popular parallel execution models. We use the Java fork/join framework for fork/join, and the Hadoop platform for MapReduce.

We want to evaluate these two parallel platforms in terms of *scalability* and *programmability*. Our set task is the creation and execution of a simple concordance benchmark application, taken from phase I of the SICSA multicore challenge.

We find that Java fork/join has low startup latency and scales well for small inputs (<5MB), but it cannot process larger inputs due to the memory limitations of a single multicore server.

On the other hand, Hadoop has high startup latency (tens of seconds), but it scales to large input data sizes (>100MB) on a compute cluster.

Thus we learn that each platform has its advantages, for different kinds of inputs and underlying hardware execution layers. This leads us to consider the possibility of a *hybrid* approach that features the best of both parallel platforms. We implement a prototype grep application using a hybrid multi-threaded Hadoop combination, and discuss its potential.

1 Introduction

This paper compares the performance and programmability of three parallel paradigms: fork/join, MapReduce, and a hybrid approach. The basis of comparison is the *concordance* application from phase I of the SICSA multicore challenge (described in Section 2) and the *grep* application for the hybrid approach.

We examine two specific implementations, running on diverse parallel platforms:

1. **Fork/Join** targets single node, multicore parallelism. We use the Java fork/join library for our experiments.

2. **MapReduce** targets compute cluster, distributed, parallelism. We use the open-source Hadoop Java-based platform for our experiments.

*R.Stewart@hw.ac.uk
†Jeremy.Singer@glasgow.ac.uk

The investigation of a **hybrid** approach is a combination of these two established technologies.

This comparison, as part of the multicore challenge, is timely. The fork/join model of parallelism [1] is described in Section 3.1. It is appropriate for *shared-memory* architectures, General interest in *distributed-memory* computing platforms is increasing. Section 3.2 describes the MapReduce paradigm [2], which is designed for large-scale clustered architectures. Lastly, we consider a hybrid approach between fork/join and MapReduce, described in Section 7. We study a *hybrid* implementation of the widely used Hadoop *grep* benchmark [3] extended to use multi-threaded tasks.

There is much current interest in hybrid processing models. Attempts to combine eager work distribution and multicore parallelism are emerging for a number of language implementations. The hybrid combination of MPI and OpenMP is investigated in [4] for parallel C programming, and a two layered parallelism model for Haskell is investigated in the implementation of CloudHaskell [5].

Our first implementation of concordance uses Java fork/join (Section 5.1), assessing the performance on an eight-core machine. The second implementation is in Hadoop MapReduce (Section 5.2), running on a 32-node Beowulf cluster. We consider these two off-the-shelf software frameworks and hardware platforms to be realistic implementation technologies for the purposes of comparative evaluation.

We evaluate our concordance implementations on a variety of sizes of textual input data in Section 6. We show that the fork/join platform scales well for smaller inputs, but reaches a performance limit for larger inputs, due to the size restrictions of *shared-memory*, single node architectures. In contrast, the Hadoop platform has significant startup overhead for small size inputs, although it scales well for much larger inputs. Section 8.1 gives an extended discussion of these tradeoffs.

Apart from these performance differences at opposite ends of the spectrum of input size, we note that there is a great deal of similarity between these two Java-based parallel programming frameworks. Our two implementations employ similar data structures, control flow and worker task subdivision.

2 Concordance Benchmark

The exact specification for the SICSA multicore challenge phase I is available on the web [6]. The *concordance* application takes a text file as input, and produces a set of phrases up to length N that occur in the text file, along with a set of locations in the file where each phrase occurs. For example, take the text *"the dog sat on the floor whilst the cat sat on the mat"*. If N=3, then concordance evaluation produces 1 phrase of frequency 4, 5 phrases of frequency 2, and 20 unique phrases (only occurring once). Table 1 summarises these results. The application involves string processing tasks, and is representative of a large class of high-throughput text mining applications. Example use cases might include:

1. A workstation user generates a concordance for a set of literature, such as the works of Shakespeare. Commercial tools such as WordSmith[1] are available to perform this task.

2. A data centre generates an index of a massive text corpus, such as the world wide web. Search engine providers carry out this exercise on a regular basis.

[1] `http://www.lexically.net/wordsmith`

Note that the specification for the *concordance* benchmark states that output can be written to file or standard output. We have chosen to use file output in our implementations. Also, the original specification states that the output is not obliged to include unique phrases from the input text. We have chosen to ignore unique phrases entirely in our output files.

Table 1: Concordance evaluation

Frequency	Sequences
4	*the*
2	*sat; on; sat on; on the; sat on the*
1	*dog; floor; whilst; cat; the dog; dog sat; the floor; floor whilst; whilst the; the cat; cat sat; the dog sat; dog sat on; on the floor; the floor whilst; floor whilst the; whilst the cat; the cat sat; cat sat on; on the mat*

3 Languages and Libraries

3.1 Fork/Join

The Java fork/join framework [7] provides support for fine-grained, recursive, divide-and-conquer parallelism. The framework was originally implemented as part of the Java Specification Request JSR166 experimental language extension providing concurrency utilities[2]. However fork/join has been integrated as part of the standard Java library since Java 7, late 2011 [8].

The *fork* operation creates a new unit of work that may execute in parallel with its parent creator. The *join* operation merges control once the parent (forker) and child (forked) tasks have both completed. The significance of using fork/join parallelism is that it provides very efficient load balancing, if the subtasks are decomposed in such a way that they can be executed without dependencies.

Fork/join parallelism is implemented by means of a fixed pool of worker threads. Each worker thread can execute one task at a time. Tasks waiting to be executed are stored in a queue, which is owned by a particular worker thread. Currently executing tasks can dynamically generate (i.e. fork) new tasks, which are then enqueued for subsequent execution. This is ideal for dynamic task creation, when we cannot determine beforehand how many tasks there will be, e.g. recursive divide and conquer.

When a worker thread has completed execution of a particular task, it fetches the next task from its own queue. If its queue is empty, it can *steal* a tasks from another queue. This work stealing enables efficient load balancing.

The explicit inspiration for the Java fork/join framework comes from Cilk [9, 10]. Intel now support Cilk[3] and have incorporated its parallel programming concepts as language extensions to C and C++. There are recent developments in other languages that offer similar work stealing, fine-grained parallel task functionality, such as the .NET task parallel library [11].

The Java fork/join framework targets single JVM, shared-memory parallelism, i.e. a multicore machine running a multi-threaded Java application in a single JVM instance. The

[2] `http://jcp.org/en/jsr/detail?id=166`

[3] `http://software.intel.com/en-us/articles/intel-cilk-plus/`

number of worker threads in a fork/join pool is generally upper-bounded by the number of cores in the system.

The dynamic load balancing is useful since some forked tasks may complete execution much quicker than others. For instance, the computation time may depend on values in the input data set, rather than just the size of the input data. Again, there may be architectural reasons for some tasks finishing quicker. Cache locality, or CPU turbo boosting [12] may make some cores execute faster. Indeed, heterogeneous multicore architectures are an ideal target for dynamic load-balancing systems.

3.2 MapReduce

MapReduce [2] is one of a number of popular *distributed execution models* that are designed to exploit the distributed nature of compute clusters. MapReduce exposes a simple programming model that abstracts distributed computing fundamentals, such as communication and synchronisation. In addition it provides support for fault tolerance, locality-based scheduling, and transparent scaling. MapReduce was proposed by Google as "a programming model and an associated implementation for processing and generating large data sets" [2]. The model is designed for coarse-grained parallelism across large datasets. MapReduce is language-agnostic; there are MapReduce libraries for a number of programming languages including C++, Erlang, F#, Perl and Python.

3.2.1 MapReduce Paradigm

In the simplest case, a MapReduce programmer only needs to specify two tasks: *map* and *reduce*. The logical model of MapReduce describes the data flow and types from the input of *key/value* pairs to the output list as follows:

$$\begin{aligned} \textbf{Map} &: (K_1, V_1) \rightarrow \mathit{list}(K_2, V_2) \\ \textbf{Reduce} &: (K_2, \mathit{list}(V_2)) \rightarrow \mathit{list}(K_3, V_3) \end{aligned}$$

Each **map** task receives an input *key/value* pair as an argument and returns a list of intermediate *key/value* pairs. As shown above, the input and output key/value combinations can have different types. An implicit *merge* phase groups together values that share a common key. Each **reduce** task is passed an intermediate *key* and its associated list of values, reducing them to form a possibly smaller list of output values.

3.2.2 Hadoop

[13] is a popular MapReduce implementation, which provides a runtime system for distributed job execution and a distributed file-system. Contributions to the Hadoop software stack come from a variety of sources, including Cloudera, HortonWorks, IBM, and Oracle. There is an active development community driving the project forward. The Hadoop stack is suitably modularised for this purpose, and boasts a number of abstract language implementations, as well as NoSQL datastores and RESTful front-ends to semi-structured database interfaces.

3.2.3 Related work

A number of higher-level language abstractions over the MapReduce model can be found within the Hadoop stack, e.g. to offer an SQL-like interface to the Hadoop distributed file-system [14], or a dataflow language [15]. Such language abstractions have been compared previously [16].

A notable alternative to the MapReduce model is Microsoft's Dryad framework [17]. It is more flexible than MapReduce since data flow follows a directed acyclic graph, whereas MapReduce is restricted to a bipartite graph of map and reduce tasks. CIEL [18] is another comparable execution engine. CIEL supports data-dependent control-flow decisions, which enable iterative and recursive algorithms. These can only be achieved with Hadoop using logical driver programs (outwith MapReduce) to submit jobs recursively. However this approach incurs latency due to consecutive job submissions. Extensions of MapReduce have been introduced in an attempt to bridge this gap, including CGL-MapReduce [19] which caches static (loop-invariant) data in RAM across MapReduce jobs, and HaLoop [20] which extends Hadoop with the ability to evaluate a convergence function on reduce outputs.

3.3 Comparison between Fork/Join and MapReduce

Fork/join and MapReduce are both parallel execution paradigms, affording evaluation of tasks on independent processing elements (PEs). PEs in the fork/join world are cores in a chip on a single compute node. In the MapReduce realm, PEs are independent compute nodes, connected on a network. Both paradigms share the behaviour of splitting a program into parts for partial evaluation, and then combining the results. Despite this commonality, some distinctions can be observed.

Hadoop divides a program into parts, prior to distributing map tasks to be evaluated on remote compute nodes. As such, the number of tasks created is statically determined by the input size, and Hadoop's runtime parameters. In contrast, divide and conquer algorithmic skeletons may be implemented in fork/join, since fork/join tasks can dynamically subdivide if the computation effort exceeds a specific threshold. Another distinction is that fork/join allows inter-task communication. However map tasks in Hadoop are wholly independent until the output of map tasks are reduced.

Both paradigms have some degree of task *redistribution*. The Java fork/join library implements a work stealing algorithm, which has been shown to be an effective load balancing strategy for parallel runtime systems [21]. Each worker thread maintains a double-ended task queue. Should a worker thread exhaust its local queue, it steals a task from the tail of another thread's queue.

Hadoop does not allow dynamic task creation needed by such parallel algorithmic skeletons as divide and conquer [22]. However it does have *speculative execution* built into the framework, when the *mapred.[map/reduce].tasks.speculative.execution* property is enabled [13]. The progress of a task is monitored in the runtime system, relative to the progress of other tasks. Given a job that instantiates millions of tasks, sluggish tasks are likely but unwelcome if they dominate the overall execution time of a job. If a task, after some time of evaluation (usually at least one minute), has not made sufficient progress, then it is duplicated on another compute node. Whenever one of the two identical tasks is fully evaluated, the other is no longer needed so it is killed.

4 Experimental Infrastructure

4.1 Fork/Join platform

For the fork/join experiments, we use a single node from the compute cluster described in Section 4.2. The Java VM is version 1.7.0-02-b13. The Java source code is compiled to bytecode using the javac version 1.6.0-27. We use the jsr166y library, specifically the jarfile version released on 25th January 2012. We use the ConcurrentHashMap implementation provided in the standard Java 7 library.

We set the number of worker threads in the fork/join pool to a maximum of eight, which is the number of cores in a node. This follows the recommendation of the jsr166y documentation[4]. Since all the cores are clocked at the same frequency, and the tasks are doing an equal amount of work to a first approximation, then we do not anticipate work-stealing. Thus for efficiency, we only create one fork/join task per thread. We leave the initial size, resize threshold, and concurrency factor parameters for the ConcurrentHashMap at their default settings. There may be potential to tune these values. We set the starting and maximum JVM heap size to 4 GB. We use the default JVM `-server` settings for the garbage collection algorithm. Again, there may be some significant benefit from tuning these options manually.

4.2 Hadoop platform

4.2.1 Concordance Measurements

The Hadoop experiments were executed on a 32 node Beowulf cluster. Each node runs CentOS 5.5, Linux kernel 2.6.18 64bit. The hardware specification of each node is 12GB main memory; 8 core Intel Xeon 2Ghz CPU; 1Gbps ethernet. Hadoop version 0.20.3 is used.

Hadoop is highly configurable, allowing administrators to specify parameters for the distributed file-system and the underlying runtime environment. We tune some of these parameters to improve the performance of our concordance implementation.

An influential parameter is *mapred.reduce.tasks*, which dictates the number of reducers for submitted jobs. There are heuristics for optimizing this value, which we employed. However methods for establishing the optimal value for this parameter are imprecise, and setting it incorrectly induces unwelcome performance results [16]. The recommended number of reducers is $0.95 \times$ *number of nodes* $\times$ *mapred.tasktracker.tasks.maximum* [23], which evaluates to using *25* reducers for our system.

The other key parameter is the *[map/reduce].tasks.maximum*, which specifies the maximum number of map and reduce tasks that can be executed in parallel on one node at any given time. The compute nodes host eight cores per CPU, and we set the maximum number of tasks to be eight.

4.2.2 Grep Measurements

The same Beowulf cluster is used for these experiments as in Section 4.2.1. Four compute nodes are used, and the Hadoop parameters are tuned specifically for the grep implementation, as described in Section 7.1.

We implement a text file generator to create the input files for grep. It generates random sequences of words taken from the Unix dictionary. It allows the number of paragraphs; the

[4] `http://gee.cs.oswego.edu/dl/jsr166/dist/docs/java/util/concurrent/ForkJoinPool.html`

number of lines in a paragraph; and the number of words in a line to be configured. The generated input files range from 844MB to 27GB in size. The number of paragraphs dictate the size of the file, since each paragraph has 30 lines of text, with 15 words per line. The files are split such that each map task processes a single line of text.

5 Implementation of Concordance

5.1 Java Fork/Join implementation

This section describes the Java fork/join implementation of the concordance application. The project source code is available from our online repository[5]. We are using svn version `r17` throughout this paper.

5.1.1 Implementation details

We follow the conventional software engineering approach of reusing appropriate Java standard library classes as far as possible, rather than creating our own custom data structures.

We split the program into four consecutive phases:

1. READ: loads the input text file from secondary storage into main memory.
2. SPLIT: divides the text into k fragments, each of which will be analysed by a separate task.
3. ADD: each of the k tasks processes its fragment independently and adds (phrase, index) (key, value) pairs to a global hashtable.
4. PRINT: iterates over the global hashtable and prints out all (key, value) pairs.

Below we summarise the relevant implementation details for each phase.

In the READ phase, the input text is stored in main memory as an `ArrayList` of strings, with one string per element. This data loading operation is a sequential phase of execution. Note that we filter out punctuation marks at this stage. We use a buffered file reader to make the overall I/O more efficient. We use the `ArrayList` structure since the input text is read-only, once it has been stored in memory. There is no need for thread synchronisation, as provided by `Vector`, for this data.

In the SPLIT phase, we decide how many worker tasks to allocate to the problem, and subdivide the input text into that many fragments, say k. The application will have k `ForkJoinTask` instances live at once. In this phase, we compute the start and end index in the input text `ArrayList` for each of the k concurrent tasks, to indicate the extent of the input fragment to process in that particular task.

The ADD phase performs the bulk of the computation. This phase executes in parallel using the `ForkJoinPool` of worker threads. There is an outer loop over phrase lengths i, from 1 to N. For each iteration of this outermost loop, k worker `ForkJoinTask` instances are instantiated. As each worker thread executes, it scans through its associated input fragment to determine all phrases of exact length i. Each phase has an associated start index in the input `ArrayList`. The worker thread stores each (phrase, index) pair in the

[5] `http://sicsaconcordance.googlecode.com`

global hashtable, provided that the phrase does not have a unique prefix phrase[6]. The phrase is a `String` object, which is used as the hashtable key. The index is a boxed integer. The global hashtable is implemented as a `ConcurrentHashMap<String,Vector<Integer>>`. This allows:

1. multiple concurrent writers to update different portions of the hashtable, using distinct locks for the distinct table regions.

2. repeated phrases to be stored with the same key, in a `Vector` of integers.

Figure 1 gives an overview of a single worker thread's activity during the ADD phase.

Because a phrase may span multiple fragments, we allow tasks to read from other tasks' fragments. However the task whose fragment contains the first word of the phrase is responsible for identifying that phrase and storing its location in the hashtable.

Finally, the PRINT phase operates sequentially. It iterates over the keyset of the concordance hashtable, and writes each (key,value) pair out to file. Each key is a phrase string. Each value is a list of input indices that match the corresponding phrase.

5.1.2 Potential inefficiencies

One sequential bottleneck in this concordance implementation is the file handling. We read input text data from a single file sequentially, and we write the concordance information out to a single file sequentially. Since we are using a local file-system on a single OS instance, there is little possibility for parallel optimization here.

The `ConcurrentHashMap` may suffer from contention. Unlike the standard Java `HashMap`, it allows multiple concurrent writers, so long as they are updating values for keys in different regions of the table. The overall logical table is split into physically smaller tables, each of which may be modified concurrently. It is possible to tune the number of separate regions based on expected concurrency level, however we have left this parameter setting at its default value.

Further, the `Vector` stored at each hashtable entry may cause runtime memory bloat [24] since we do not expect many phrases to have lots of repeats. Again, it is possible to supply a parameter to the `Vector` constructor to state its expected length and size increment amount. We also left these settings at their default values.

Runtime tuning is also possible for the garbage collector, however we left the settings at the default JVM server values.

Whenever we are sure that only one thread accesses a data structure at once, we use the cheaper non-synchronised data structures, i.e. `ArrayList` instead of `Vector`, and `StringBuilder` instead of `StringBuffer` to avoid unnecessary synchronisation overhead.

5.2 Hadoop Implementation

The canonical MapReduce example is wordcount, shown in Figure 2, which illustrates mapping over large text files in parallel, and reducing the sum of every word in the file. Each map task processes a line of text from an input file and iterates over each word w outputting `<w,1>`. The Hadoop runtime system collects together the output key/value pairs

[6]Every unique phrase of length i starting at index x will also generate a unique phrase of length $i+1$, provided $x+i+1$ does not reach the end of the file. We only need to store the shortest unique phrase rooted at index x, to save memory.

```
1  while (index < endIndex) {
2    // find phrase of length N at this index
3    // if phrase does not have a uniquely occurring
4    // prefix phrase, add <phrase, index> to concordance
5
6    StringBuilder phrase = new StringBuilder("");
7    int i;
8    boolean uniquePrefix = false;
9    for (i=0; i<N-1; i++) {
10     // check if any phrase prefix has
11     // a unique occurrence
12     String prefix = phrase.toString();
13     AbstractCollection<Integer> v = concordance.get(prefix);
14     if (v.size()==1) {
15       uniquePrefix=true;
16       break;
17     }
18     if (i>0) {
19       phrase.append(" ");
20     }
21     phrase.append(text.get(index+i));
22   }
23   // finished checking prefixes
24   if (!uniquePrefix) {
25     if (i>0) {
26       phrase.append(" ");
27     }
28     // create the full-length phrase of N words
29     phrase.append(text.get(index+i));
30     String concordanceEntry = phrase.toString();
31     AbstractCollection<Integer> v = concordance.get(concordanceEntry);
32     if (v == null) {
33       // set up new list for this entry, since it
34       // is the first time we have seen it
35       v = new Vector<Integer>();
36       AbstractCollection<Integer> tmp = concordance.putIfAbsent(
             concordanceEntry,v);
37       if (tmp != null) {
38         v = tmp;
39       }
40     }
41     v.add(index); // put the start-index for this phrase
42   }
43   index++;
44 }
```

Figure 1: Java fork/join code snippet that adds phrases to the concordance hashtable

```
1  map( line ) {
2    Tokenizer tok = line.tokenize();
3    while (tok.hasMoreTokens) {
4      output(tok.next(), 1);
5    }
6  }
7
8  reduce( word, values ) {
9    int sum = 0;
10   while(values.hasNext()) {
11     sum += values.next();
12   }
13   output(word,sum);
14 }
```

Figure 2: Hadoop wordcount pseudo-code

from all map tasks. It merges together sets of values associated by the same key (in this case, the same word). Finally each reduce task sums the values for each word to establish the frequency of each word in the input file.

The concordance program outlined in Section 2 is generally similar in structure to wordcount, however there is an important alteration. The wordcount example splits map inputs by dividing text files by lines, which is not possible when concordance evaluation mandates the consideration of sequences spanning multiple lines in the input file. We must address the common problem of *tiling* the input data [25] when designing our implementation.

We produce a MapReduce implementation of concordance by modifying the wordcount example in several ways. The simplest approach would be to evaluate the whole text file in one map task, thus avoiding the tiling problem. However the performance of such a solution would be extremely poor given given its lack of map parallelism.

In order to allow tiling of data from one line to the next, we add an implementation of the *RecordReader* interface. This prepends the line sequence input to a map task with the tail of the previous line, where the tail length is governed by the concordance maximum phrase length parameter, N.

Another requirement of the specification is to maintain a list of pointers identifying the position of each occurrence of a phrase in the input text file. This requires a trivial extension to the Hadoop wordcount implementation. The custom record reader keeps track of word pointers whilst it scans the text file, splitting it into records. The input key/value pairs to the map tasks contain the relative position of a sequence as the key, and each text sequence (of length N) as the value. The output *key/value* pair from the map task is the sequence as the key (up to length N), and its position in the text as the value, e.g. for the example given earlier in Section 2 output pairs would include `<dog sat,2>`, `<on the floor,4>`, `<the cat sat,8>`.

The Hadoop framework collects together identical keys to reduce their collective values. This process is opaque to the MapReduce application, highlighting the simplicity of the MapReduce programming model. The effect of this merging process is that each reduce task receives the word sequence (of length up to N) as a key, and a list of that phrase's positions in the text as values. Reducers therefore have the simple task of concatenating these position values together, before the result is written to the Hadoop file-system. An important code refinement is to use `StringBuilder` objects for this concatenation process, rather

than simple `String` appends. This adjustment yields radical performance improvements. Since `String` objects are immutable, each append operation creates a new `String` object: In contrast, `StringBuilder` encapsulates a mutable character array. We also turned on compression for the outputs of the map tasks, though consequential improvements to runtime were negligible.

5.3 Comparing Fork/Join and Hadoop Implementations

Whereas in the Hadoop implementation, the control-flow and data structures for the concordance are mostly implicit, we have to specify these things explicitly in Java fork/join. In that sense, fork/join is a *lower-level* parallel programming dialect than MapReduce. This explicit specification of control-flow and data structures may be an advantage (in that the developer can specify her algorithm more precisely) and also a disadvantage (in that the code becomes more complex and there are non-obvious and multi-dimensional tuning options).

6 Evaluation

6.1 Fork/Join Performance

A number of sample input text files were suggested for the concordance multicore challenge, which we use to assess implementation performance. They are ASCII files of size *0.003MB, 0.06MB, 1.49MB, 4.79MB*, and *104.3MB* [7].

We report timings on a single eight-core node, as described in Section 4.1. For each test, we set the maximum phrase length parameter N to 10 words. The concordance program fails with an *OutOfMemory* error while processing the largest file of size 104.3MB. In each experiment, the reported time measures the generation of the concordance in memory and the output of the unsorted results to a single file. Note that we do not consider the time taken to read the input text file from the filesystem. This corresponds exactly to the timing regime used for the Hadoop measurements in the next section. All reported timing results are arithmetic means of 30 runs.

Java fork/join concordance execution times are in the range of 0.1–10 seconds when using eight worker threads in the threadpool, as can be seen from Figure 4. Figure 3 shows the runtime speedup for the four smallest files. Increasing the number of worker threads from one to eight improves runtime performance for the 1.49MB and 4.79MB files, with the biggest speedup of 2.35 for the 1.49MB file using six threads. Speedup performance for the two larger files correlate closely. There is a negligible slowdown for these inputs when increasing the number of worker threads from six to eight, perhaps due to interference with OS and background threads.

For the 0.003MB file, the average write-to-file ratio relative to the the total runtime is 69%, and for the 0.06MB file, the ratio is 37%. There is a larger variation in the measurements for the smallest files, due to IO latency and noisiness. As such, runtime improvements are diminished when increasing the threadpool size for the two smallest files, as (i) the sequential write-to-file time dominates the overall concordance execution, and (ii) the overhead of thread management is not mitigated by the small amount of parallel work achieved.

[7] Text files available for download at `http://www.macs.hw.ac.uk/~rs46/multicore_challenge1`

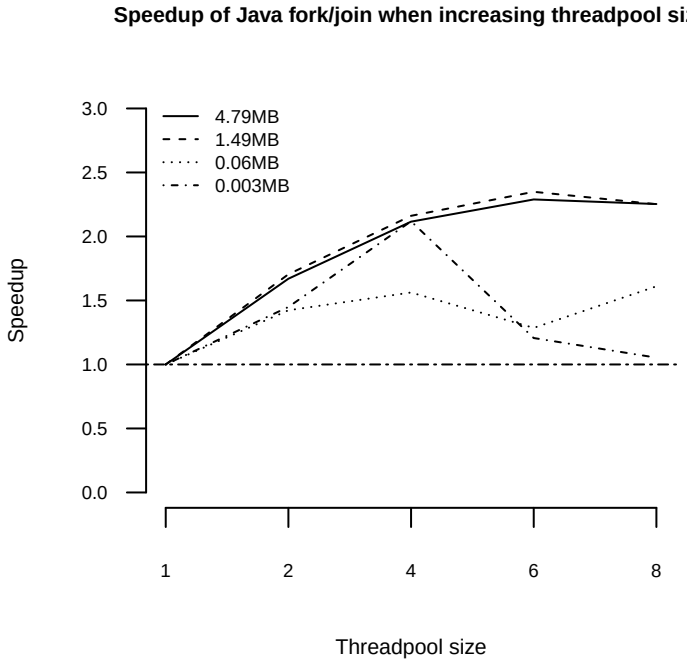

Figure 3: Runtime speeds for Java fork/join, scaling threadpool size

6.2 Hadoop Performance

As stated in Section 3.2, the MapReduce paradigm is well suited for high-throughput computation on very large datasets. Examples of the scalability of Hadoop are often given when evaluating large datasets in orders of gigabytes or even terabytes [26] of data. Hadoop is *not* a low latency execution engine; for small inputs to our concordance implementation, the runtime measurements are very poor. In these cases, most of the time is spent scheduling the job, coordinating the nodes and task trackers. Hadoop latency issues are discussed in more detail in Section 8.1. We report these observations in this section, and go on to show that Hadoop scales amicably for larger inputs. All Hadoop experiments are executed three times on their respective platforms, and the mean values of these are reported in this study.

The same input files were used as for the Java fork/join experiments. A number of additional implementations are compared: sequential C, sequential Haskell, Java fork/join (using eight threads) on one node; and Hadoop on 32 nodes, again setting the phrase length N at 10.

The results are shown in Figure 4. For the two smallest inputs, C, Java fork/join and Haskell all fully evaluate concordance in under 1 second, where as Hadoop, with its high latency submission process, takes 38 seconds for both files. Increasing input size to 1.49MB, C takes 2 seconds, Java fork/join 3, Haskell 8, and Hadoop, 39 seconds. For the 4.79MB file, there is little movement in Hadoop's runtime speed, at 39 seconds, whereas Haskell takes 32 seconds, Java fork/join 8, and C, 4 seconds.

The largest input file is of size 104.3MB—still a small file in terms of the scope of Hadoop, though much larger than previous inputs. Neither the sequential Haskell or Java fork/join implementations are able to evaluate the file, running out of local memory before

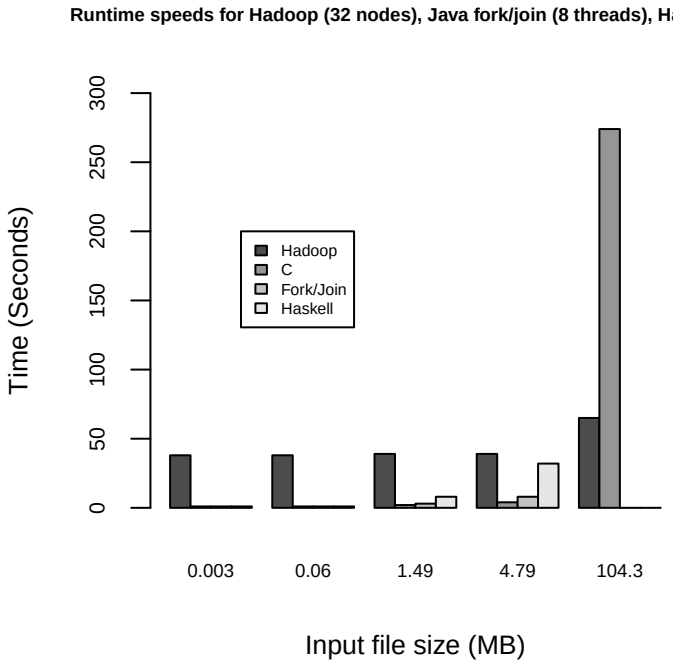

Figure 4: Runtime speeds for Hadoop, Java fork/join, Haskell & C

they are able to do so. The sequential C implementation took 274 seconds to evaluate the concordance, and the Hadoop implementation required 65 seconds on 32 nodes. Contrast this result with the 39 seconds Hadoop took to evaluate the 4.79MB file. This is a 21x input size increase, at the expense of only a 66% increase in runtime speed.

6.3 Programmability Comparison

Section 3.3 describes the similarity between the fork/join and MapReduce models in general. Section 5 presents particular similarities in the implementation of parallel concordance generation. In both frameworks, the input text file is processed in independent sub-units and the results of these parallel analyses are aggregated. This section now briefly explores the *program size* of the two implementations, comparing the conciseness of map and reduce operations with fork and join. We also detail the amount of boilerplate code required for each framework. For completeness, we also examine the sizes of the sequential C and Haskell implementations.

Table 2 shows the source lines of code metrics derived from the CLOC reporting tool [27]. We choose to ignore blank lines and comments. The separation of concerns when using the MapReduce and fork/join models is vindicated, with 62 lines of map and reduce code, and 74 lines of fork and join code. The remaining 152 lines of Java code in the Hadoop implementation mostly consist of (i) setting up the Hadoop job, and (ii) implementing a custom record reader required for the concordance application, described in Section 5.2. The remaining 179 lines of Java code in the fork/join implementation involve (i) reading the input file, and (ii) iterating over the text vector inserting phrases into the concordance hashtable, described in Section 5.1.

Table 2: Source Lines of Code comparison

Language	Lines of code
Hadoop MapReduce *(map+reduce code only)*	214 *(62)*
Java fork/join *(fork+join code only)*	253 *(74)*
Sequential C	288
Sequential Haskell	66

Compare these counts with sequential C, which is 14% larger than Java fork/join and 35% larger than Java Hadoop MapReduce. However this is not an entirely fair comparison since the C version pretokenizes the input text in order to process and store phrases more efficiently. The sequential Haskell implementation [28] is far more concise than the other three comparisons, consisting of only 66 lines of code. Neither the C or Haskell sequential implementations contain constructs for parallel evaluation, and do not scale for larger inputs, as shown in Section 6.2.

7 Investigating a Hybrid Model

Section 6 evaluates the performance of two implementations of concordance, one for shared-memory systems with multicore chips and the other for distributed-memory clusters. This section explores the combination of the two, investigating the feasibility of a two-tier parallel architecture.

The combination of multi-threading with MapReduce appears in principle to be a union of two complementary technologies. Hadoop offers a number of configurable parameters for tuning performance on multicore nodes. In addition, a *MultithreadedMapper* class is provided in the Hadoop libraries for convenience, which is used in one implementation described in this section to assess the efficacy of the hybrid approach.

7.1 Hadoop parameters

The most commonly used parameter for multicore deployment is *mapreduce.tasktracker.[map/reduce].tasks.maximum*[8]. An important characteristic of Hadoop is that by default, a new JVM is spawned on a Hadoop node for each task to be evaluated. As such, this parameter dictates how many concurrent JVMs may be instantiated on each node at any point in time. To achieve multicore parallelism, this parameter may be set to the number of cores on a node.

Another parameter that can be used to tune node efficiency is *mapreduce.job.jvm.numtasks*, although this is not directly related to achieving multicore parallelism. The default value is `1`, which entails that a JVM is instantiated to execute a single task and is then terminated. Setting this parameter to a value greater than one means that the existence of a JVM will be prolonged to evaluate multiple tasks in sequence. This can useful to avoid the overheads of JVM instantiation. If the value is set to `-1`, then a JVM can be reused for an unlimited number task evaluations.

[8]In the new Hadoop APIs, since v0.20.3, used in these experiments

JVM reuse for CPU-bound tasks may be able to take advantage of adaptive recompilation technology like HotSpot [29]. After execution for long periods of time, the HotSpot JVM detects performance-critical sections of code, and dynamically compiles these hot code regions into highly optimized native machine code [13]. Enabling JVM reuse for long running tasks may enable the HotSpot JVM to improve Hadoop performance.

7.2 Multi-threaded JVMs

The Hadoop library offers a *MultithreadedMapper* class, which extends the default *Mapper* class. A maximum number of threads is specified for the thread pool to evaluate a map task. In terms of performance, the critical behaviour of multi-threaded maps is that IO, e.g. reading from the Hadoop distributed filesystem, is done *prior* to the threads being created [30]. As such, the multi-threaded mapper class is more suitable for CPU intensive map tasks, or *non-Hadoop* IO like web crawlers.

7.3 Comparison

These parameters offer a design choice for implementing multicore parallelism in Hadoop programs, i.e. should we (i) configure Hadoop to instantiate multiple (single-threaded) JVMs on each node, or (ii) throttle the number of JVMs and instead permit multi-threaded parallel task evaluation. The contrasting approaches have been widely debated [31], and it is generally agreed that adhering entirely to the Hadoop model is safer, has fewer overheads, and achieves optimal performance in most cases. It is a single-threaded model (Hadoop is *not* thread safe), which imposes JVMs as hosts for the smallest grain of computation—a *task*.

Latency is an unfortunate side effect of JVM instantiation. The more map tasks in a job, the higher the overall latency. However, Hadoop is a *high-throughput* execution engine for large datasets, and it is assumed that tasks will take a reasonable amount of time to execute (at least a few minutes), dominating JVM instantiation overheads (at most a few seconds).

A key benefit to the multiple JVM approach is that IO throughput is much higher to separate OS processes than it is to only one JVM. This is the case for the multi-threaded mapper that Hadoop provides, as map threads are created only once IO to the mapper has taken place.

7.4 Experiment Details

We use the standard *grep* benchmark [32] , which is common for high-throughput computation comparisons. We have a proof of concept implementation for each approach outlined above. We start from the published Hadoop implementation of grep [3], and extend this to evaluate performance improvements from multi-threaded tasks and JVM reuse. The experimental platform consists of four compute nodes from the platform described in Section 4.2.

Firstly, the published implementation is measured without any modifications, and the *mapreduce.tasktracker.map.tasks.maximum* parameter is set at eight. The mapper class distributes input records (lines of text) to the available maps residing in separate JVMs, in a round robin fashion. Next, we consider JVM reuse, setting this value to -1 to allow an unlimited number of tasks to be executed sequentially on each JVM. The third approach is

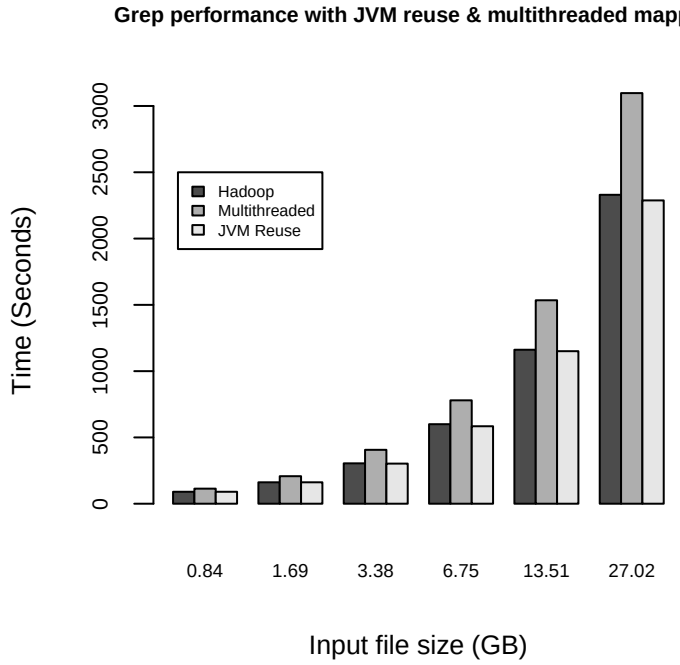

Figure 5: Grep results for Hadoop, enabling JVM reuse and using multi-threaded mappers

to set *MultithreadedMapper* as the mapper class in the application. The number of threads in the threadpool is set to eight, and the maximum number of tasks set to one. The multi-threaded mapper provided by the Hadoop libraries cycles through the input records, distributing them to maps residing in threads. This offers a direct comparison between using multiple threads for multicore parallelism as opposed to using multiple JVMs as OS processes.

7.5 Performance Results

The grep results are shown in Figure 5. For each input size, the JVM reuse policy gives a modest performance improvement in comparison to the unmodified implementation. The greatest speedup is 1.9%, seen for the largest input which contains 6.4 million paragraphs, with a file size of approximately 27GB. The most notable observation from these results is that the multi-threaded implementation has the slowest runtime for all input sizes. The poorest runtime speed relative to the unmodified version is a 32.9% slowdown, observed with the largest input file. For the three largest input files, the relative slowdown in ascending input size is 30%, 32.1% and 32.9%.

The poor performance of the multi-threaded mapper can be attributed to the single-threaded IO that each threaded mapper must perform through their JVM host, discussed in Section 7.3. The other two implementations benefit from multiple JVMs residing on each node, doing IO to read in the input file, *in parallel*. The JVM reuse policy offers only very slight performance improvements. Section 7.1 outlines the potential benefit of JVM reuse, which is to take advantage of adaptive optimization for CPU intensive tasks; however the grep application does not fit into this category.

8 Conclusion

8.1 Discussion

We have presented two implementations of the concordance application, one using using Java fork/join on a multicore server, and the other using Hadoop MapReduce on a compute cluster. The implementations are similar in design, which reflects the closeness of the two paradigms for parallel computation. The ease of use for each model is demonstrated in the lines of code analysis, where both are smaller than the sequential C for concordance: Hadoop MapReduce by 35% and Java fork/join by 14%. Moreover, the map and reduce task code amounts to approximately 28% of the Hadoop code, and the fork/join code approximately 29% coverage of the Java program.

For the two larger of the four input files to the Java fork/join concordance implementation, runtime is improved with multi-threaded execution. We observe significant runtime improvements in comparison to a threadpool size of 1. The biggest speedup is achieved when using a threadpool size of 6, with speedup's of 2.35 and 2.29 for the 1.49MB and 4.79MB files, respectively.

The Java fork/join implementation scales well for the smaller input text files (up to 4.79MB). However when we scale up to the 104.3MB input file, Hadoop's utility is clearly demonstrated. Evaluating concordance on the 4.79MB text file takes 34 seconds, whereas on the 104.3MB file (21x larger) it takes 65 seconds. The MapReduce implementation exhibits high startup latency, meaning that it spends a significant proportion of time initialising the job for smaller input file sizes. Both the Java fork/join and sequential Haskell implementations were unable to evaluate the 104.3MB file; local memory is exhausted on a single node before completing the concordance generation.

There are many reasons why Hadoop has latency problems, which degrade its performance for small input sizes [33]. For instance, Hadoop implements a work stealing algorithm, whereby *TaskTrackers* on slave nodes ping for work from *JobTrackers* on the master node, and the latency of this ping is a few seconds. Another factor is the time overhead to transfer Java binary executable files via Hadoop's distributed filesystem.

The Hadoop development community is actively engaged in addressing these latency issues, e.g. through the *NextGen Hadoop MapReduce* project [34]. Other priorities include scaling to 10,000+ compute nodes and 200,000+ CPU cores, and interestingly, to offer a platform to facilitate other programming paradigms apart from MapReduce, such as MPI, master-worker, and iterative models. There are many aspects of the MapReduce implementation in Hadoop, that were not in the scope of this paper. For instance, our concordance MapReduce implementation is implicitly *fault tolerant*, adopting the use of heartbeats and task reallocation when node failure is detected.

A number of Hadoop implementations of grep were also measured for performance on the same hardware. JVM reuse was explored, in addition to a multi-threaded approach for parallelising map tasks. The results show that Hadoop's model of one single-threaded JVM per task is appropriate for IO intense tasks such as the *grep* benchmark, achieving a 32.9% quicker runtime than the multi-threaded equivalent for the largest input size. Hadoop JVM reuse attempts to improve the execution of CPU intensive jobs, by means of JVM adaptive recompilation technology. However our experiments were not able to assess this feature, since map tasks in the grep implementation were extremely small, lacking any complex code that would otherwise be optimized.

8.2 Future Work

We have investigated the feasibility of combining Java fork/join with Hadoop MapReduce. We identify a number of platform and system parameters as potentially exploitable for tuning programs on multicore compute nodes. Such parameter tuning may require expert analysis, or may be amenable to automatic exploration and optimization, e.g. [35]. We discover that, despite the apparently suitable marriage of Hadoop and fork/join technologies, the Hadoop implementation of MapReduce is not thread-safe: The occasional use of the *MultithreadedMapper* class is found where map tasks are not *CPU bound*, with threads interacting with *non-Hadoop* IO systems such as web crawlers.

This leads us to contemplate the tantalising goal of a non-Hadoop hybrid distributed/shared memory solution. For instance, we might compare (i) Hadoop MapReduce with its single-threaded model of multicore parallelism, against (ii) Java RMI [36], which provides polymorphic object serialisation, in combination with Java fork/join. A feasibility study concerning the integration of Java RMI for task distribution and Java fork/join for multicore parallelism would be worthwhile.

This paper has explored the efficacy of using multi-threaded mappers, and JVM reuse within Hadoop. Future investigations should characterise precisely the conditions necessary to exploit these tools, and quantify the benefit from using them. Both JVM reuse and multi-threaded mappers are designed for CPU intensive tasks, and as such a suitable application workload would involve long running tasks, with critical code sections, involving little IO interaction. Demonstrable performance improvements under these conditions conditions would attract significant interest in exploiting multicore parallelism in distributed MapReduce frameworks.

Acknowledgements We would like to thank Harsh Chouraria for his valuable Hadoop knowledge, and to Phil Trinder and Patrick Maier for providing stimulating feedback on this paper. This work was in part supported by the Scottish Funding Council, through the SICSA programme.

References

[1] Maier D, Grcevski N, Sundaresan V. An Introduction to Java Development Kit 7. *Proceedings of the Conference of the Center for Advanced Studies on Collaborative Research*, 2011; 366–367. URL `http://dl.acm.org/citation.cfm?id=2093889.2093952`.

[2] Dean J, Ghemawat S. Mapreduce: Simplified Data Processing on Large Clusters. *Commun. ACM* 2008; **51**(1):107–113, doi:http://doi.acm.org/10.1145/1327452.1327492.

[3] Hadoop Implementation of the Grep application. `http://goo.gl/pAKGj`.

[4] Lusk EL, Chan A. Early Experiments with the OpenMP/MPI Hybrid Programming Model. *Proceedings of the International Workshop on OpenMP*, 2008; 36–47, doi:10.1007/978-3-540-79561-2_4.

[5] Epstein J, Black AP, Jones SLP. Towards Haskell in the Cloud. *Proceedings of the Haskell Symposium*, 2011; 118–129, doi:10.1145/2034675.2034690.

[6] SICSA multicore challenge phase i: Concordance application. `http://goo.gl/X4BQO` 2011.

[7] Lea D. A Java fork/join framework. *Proceedings of the Conference on Java Grande*, 2000; 36–43, doi:10.1145/337449.337465.

[8] Ponge J. Fork and Join: Java Can Excel at Painless Parallel Programming Too! `http://blogs.oracle.com/java/entry/understanding_the_java_7_fork` 2011.

[9] Blumofe RD, Joerg CF, Kuszmaul BC, Leiserson CE, Randall KH, Zhou Y. Cilk: An Efficient Multithreaded Runtime System. *Journal of Parallel and Distributed Computing* 1996; **37**:55–69, doi:10.1006/jpdc.1996.0107.

[10] Blumofe RD, Leiserson CE. Scheduling Multithreaded Computations by Work Stealing. *Journal of the ACM* 1999; **46**:720–748, doi:10.1145/324133.324234.

[11] Leijen D, Schulte W, Burckhardt S. The Design of a Task Parallel Library. *Proceedings of the ACM SIGPLAN conference on Object oriented programming systems languages and applications*, 2009; 227–242, doi:10.1145/1640089.1640106.

[12] Charles J, Jassi P, Ananth N, Sadat A, Fedorova A. Evaluation of the Intel Core i7 Turbo Boost feature. *Proceedings of the IEEE International Symposium on Workload Characterization*, 2009; 188–197.

[13] White T. *Hadoop - The Definitive Guide: Storage and Analysis at Internet Scale*. 2nd edn., O'Reilly, 2011.

[14] Thusoo A, Sarma JS, Jain N, Shao Z, Chakka P, Anthony S, Liu H, Wyckoff P, Murthy R. Hive: A Warehousing Solution Over a Map-Reduce Framework. *Proceedings of the Very Large Data Base Endowment* 2009; **2**(2):1626–1629.

[15] Olston C, Reed B, Srivastava U, Kumar R, Tomkins A. Pig Latin: A Not-So-Foreign Language for Data Processing. *Proceedings of the 2008 ACM SIGMOD International Conference on Management of Data*, 2008; 1099–1110, doi:10.1145/1376616.1376726.

[16] Stewart RJ, Trinder PW, Loidl HW. Comparing High Level MapReduce Query Languages. *Proceedings of the Advanced Parallel Processing Technology Symposium*, *Lecture Notes in Computer Science*, vol. 6965, 2011; 58–72, doi:10.1007/978-3-642-24151-2_5.

[17] Isard M, Budiu M, Yu Y, Birrell A, Fetterly D. Dryad: Distributed Data-Parallel Programs from Sequential Building Blocks. *EuroSys*, 2007; 59–72, doi:10.1145/1272996.1273005.

[18] Murray DG, Schwarzkopf M, Smowton C, Smith S, Madhavapeddy A, Hand S. CIEL: A Universal Execution Engine for Distributed Data-Flow Computing. *Proceedings of the 8th USENIX conference on Networked Systems Design and Implementation*, 2011; 113–126. URL `http://static.usenix.org/event/nsdi11/tech/full_papers/Murray.pdf`.

[19] Ekanayake J, Pallickara S, Fox G. MapReduce for Data Intensive Scientific Analyses. *eScience*, 2008; 277–284, doi:10.1109/eScience.2008.59.

[20] Bu Y, Howe B, Balazinska M, Ernst MD. HaLoop: Efficient Iterative Data Processing on Large Clusters. *Proceedings of the Very Large Data Base Endowment* 2010; **3**(1):285–296. URL `http://dl.acm.org/citation.cfm?id=1920841.1920881`.

[21] Marlow S, Maier P, Loidl HW, Aswad M, Trinder PW. Seq No More: Better Strategies for Parallel Haskell. *Proceedings of the Haskell Symposium*, 2010; 91–102, doi:10.1145/1863523.1863535.

[22] Cole M. Bringing skeletons out of the closet: A pragmatic manifesto for skeletal parallel programming. *Parallel Computing* 2004; **30**(3):389–406.

[23] Partitioning Your Job into Maps and Reduces. Hadoop wiki. URL `http://goo.gl/tpU23`.

[24] Mitchell N, Schonberg E, Sevitsky G. Four trends leading to Java runtime bloat. *IEEE Software* 2010; **27**(1):56–63.

[25] Guo J, Bikshandi G, Fraguela BB, Garzaran MJ, Padua D. Programming with Tiles. *Proceedings of the 13th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, 2008; 111–122, doi:10.1145/1345206.1345225.

[26] O'Malley O. Terabyte Sort on Apache Hadoop. `http://www.hpl.hp.com/hosted/sortbenchmark/YahooHadoop.pdf` 2008.

[27] Danial A. CLOC - Count Lines of Code. Open source 2009. URL `http://cloc.sourceforge.net`.

[28] Horstmeyer T. SICSA multicore challenge phase i: Concordance. Haskell: Sequential implementation. `http://goo.gl/qtmzD` 2010.

[29] The Java HotSpot Virtual Machine White Paper. `http://goo.gl/HVUng` 2001.

[30] Bejoy K. Enable Multiple Threads in a Mapper aka MultithreadedMapper. `http://goo.gl/qal5q` 2012.

[31] Discussion on JVM reuse and multi-threading in Hadoop. Hadoop issue tracking system: HADOOP-249 May 2006.

[32] Pavlo A, Paulson E, Rasin A, Abadi DJ, DeWitt DJ, Madden S, Stonebraker M. A Comparison of Approaches to Large-Scale Data Analysis. *Proceedings of the SIGMOD Conference*, 2009; 165–178, doi:10.1145/1559845.1559865.

[33] Discussion on high latency Hadoop initialization. Hadoop mailing list - `http://goo.gl/ocmeA` July 2009.

[34] The Next Generation of Apache Hadoop MapReduce. Yahoo Developer Network - `http://goo.gl/5LARc` February 2011.

[35] Singer J, Kovoor G, Brown G, Luján M. Garbage collection auto-tuning for Java MapReduce on multi-cores. *Proceedings of the International Symposium on Memory Management*, 2011; 109–118, doi:10.1145/1993478.1993495.

[36] Java Remote Method Invocation-Distributed Computing for Java 2010. URL `http://goo.gl/3xaVm`.