Garbage Collection Auto-Tuning for Java MapReduce on Multi-Cores

Jeremy Singer
University of Glasgow
jeremy.singer@glasgow.ac.uk

George Kovoor *
kovoor.george@gmail.com

Gavin Brown Mikel Luján
University of Manchester
firstname.lastname@manchester.ac.uk

Abstract

MapReduce has been widely accepted as a simple programming pattern that can form the basis for efficient, large-scale, distributed data processing. The success of the MapReduce pattern has led to a variety of implementations for different computational scenarios. In this paper we present *MRJ*, a MapReduce Java framework for multi-core architectures. We evaluate its scalability on a four-core, hyperthreaded Intel Core i7 processor, using a set of standard MapReduce benchmarks. We investigate the significant impact that Java runtime garbage collection has on the performance and scalability of MRJ. We propose the use of memory management auto-tuning techniques based on machine learning. With our auto-tuning approach, we are able to achieve MRJ performance within 10% of optimal on 75% of our benchmark tests.

Categories and Subject Descriptors D.3.4 [*Programming Languages*]: Processors—Memory management (garbage collection)

General Terms Experimentation, Performance

Keywords mapreduce, garbage collection, machine learning, Java

1. Introduction

The *MapReduce* programming pattern has its origins in functional programming [14]. However, MapReduce has been popularized by Google since they adopted this pattern as a highly effective means of attaining massive parallelism in compute clusters [13]. Given some input data, the `map` function operates on disjoint portions of the data, potentially in parallel, constructing a set of (key,value) pairs for each portion. The `reduce` function applies an associative, commutative operator to all values with the same key.

A MapReduce framework automatically handles issues like data-partitioning, load-balancing, and thread-scheduling. Thus the application programmer is not required to re-implement these basic threading mechanisms. Instead, the application logic is expressed simply in terms of `map` and `reduce` functions, at a suitably high-level of abstraction.

Over the past five years, MapReduce has attracted significant attention from industry, academia and the open-source community [3]. The pattern still has its detractors [15]. However, it has been demonstrated to give effective parallelism for important parts of the computer applications spectrum e.g. machine learning [10], databases [37], eScience [16].

Our objective is to investigate the MapReduce pattern in the context of multi-core architectures, rather than within compute clusters, as commonly used by Amazon, Facebook, Google and Yahoo, amongst others.

1.1 Motivation for Multi-Core MapReduce

The microprocessor product lines from all major vendors are now firmly entrenched in the multi-core era. Indeed, the industry trend is moving to *many-core*, with next-generation architectures like Intel's Single-chip Cloud Computer [22] which has 48 cores. However while the low-level architecture provides abundant parallel threads of execution, high-level programming models are not sufficiently mature or widespread to target this parallelism effectively. There has been a great deal of research effort in this area. For instance, new programming languages such as Fortress [4] and X10 [9] are under development. However these have not yet been widely adopted. At the same time, new parallelism frameworks have been introduced for existing languages, such as the Java fork/join framework [24], Pervasive DataRush [11], and the Task Parallel Library extensions to .NET [25]. These are complex libraries with significant APIs.

We have implemented our MapReduce for Java system on top of the Java fork/join framework [24]. In essence, we are treating fork/join like the assembly language of parallelism, and we are using MapReduce to provide a higher-level, simpler abstraction for application software developers.

Implementations of MapReduce that do not target clusters have started appearing recently. For example, He et al. [21] and Kruijf et al. [12] have developed MapReduce for GPGPUs and Cell processors, respectively. The Phoenix project [31, 38] is the only previous implementation that focuses on shared memory multi-core architectures. Our implementation targets the same system type as Phoenix, however we are working in Java, whereas Phoenix is based on C/C++. Java gives us certain inherent advantages, described in Section 1.2. However there can be a performance penalty caused by Java's automatic memory management, which Section 4 explores.

1.2 Motivation for MapReduce in Java

The MapReduce pattern is programming language agnostic. However in our opinion, there is plenty of synergy between the Java programming language and the MapReduce programming model. They both have a similar overall aim, in terms of reducing the burden of programming complexity. The Java language provides a runtime system that supports automatic memory management, hot code recompilation, and robust error handling, inter alia. MapRe-

* Work done while author was at Manchester

duce provides a runtime system that handles data distribution, thread scheduling, and error recovery. The philosophy underlying both frameworks can be summarized as: *Let the framework do the work.* Since the programmer is relieved from handling these complex and error-prone tasks manually, then he is free to focus on implementing the application logic of the program.

An explicit motivation for Java is its *platform independence*. MapReduce programs written in Java can be distributed as platform neutral JVM bytecode. In addition, a Java MapReduce framework is straightforward to port to new multi-core architectures, since its only requirement is a suitable virtual machine supporting lightweight parallel thread spawning on different cores.

The major open-source implementation of MapReduce, *Hadoop* [1] is also developed in Java. However Hadoop focuses on optimizing cluster-level parallelism. If a cluster node has n cores, the Hadoop runtime simply spawns n instances of a Java virtual machine on that node. This is a heavyweight approach to multi-core parallelism. We hope that our investigation of lighter-weight multi-core MapReduce using the Java fork/join framework (MRJ) can contribute directly to the evolution of the Hadoop project.

1.3 Motivation for GC Auto-Tuning

Because MRJ is implemented in Java, it depends on the automatic memory management provided by the underlying Java virtual machine. We find that Java runtime garbage collection interacts with MRJ application performance in non-obvious ways. The case studies in Section 4 demonstrate that this interaction is often benchmark-specific, or may only occur for certain heap sizes.

We argue that the MRJ end-user cannot be expected to perform expert analysis to determine (a) that GC activity is reducing the performance of MRJ, and (b) how to change the JVM configuration to improve the situation. Instead we propose the use of a GC auto-tuning system for MRJ applications. This system would have the following advantages:

1. It could adapt to benchmark-specific or heap-size-specific anomalies much more efficiently than a non-expert user.
2. It could be installed by the system administrator and automatically enabled for users that do not have sufficient permissions to change JVM parameters, e.g. on a utility computing platform [5] like Google AppEngine [18].
3. It would enable rapid deployment of MRJ on new multi-core architecture layouts. This is important due to the rapidly changing multi-many-core processor landscape.

1.4 Contributions

This paper makes several key contributions.

1. Section 3 demonstrates the scalability of our MRJ framework for standard benchmarks, on a commodity multi-core platform.
2. Section 4 highlights the major impact that Java runtime garbage collection (GC) has on MRJ, in a series of case studies.
3. Section 5 presents an auto-tuning approach to optimize GC for MRJ, giving speedups of up to 6x the default GC policy, with a 10% geometric mean speedup over all benchmarks with the largest input data sets.

2. MRJ Implementation

This section gives an overview of the design and implementation choices we made for MRJ, our MapReduce Java framework for multi-core architectures. A full technical description of MRJ is available in our earlier work [23].

```
public class WordCount implements
    StringMapper, StringReducer {

public static void main(String[] args){
  final WordCount wcinstance=new WordCount();
  JobClient jobClient=JobClient.getInstance();
  JobConf jobConf=jobClient.getConf();
  jobConf.setInputtype(InputType.File);
  jobClient.setMapper(wcinstance);
  jobClient.setReducer(wcinstance);
  jobClient.initialise(WordCount.class);
  jobClient.submitApp(wcinstance);
}

public void map(String key, String value,
                StringOutputCollector output) {
  StringTokenizer tkn=new StringTokenizer(value);
  while(tkn.hasMoreTokens()) {
    String word=tkn.nextToken();
    output.putKeyValue(word, ‘‘1’’);
  }
}

public String reduce(String key, String value) {
  StringTokenizer tkn = new StringTokenizer(value
      , ‘‘|’’);
  StringBuilder strb=new StringBuilder (key);
  int sum = 0;
  while (tkn.hasMoreTokens())
    sum += Integer.parseInt(tkn.nextToken());
  return (strb.append(‘‘:’’).append(sum)).
      toString();
}
```

Figure 1. Actual implementation of WordCount in MRJ

2.1 Example MRJ Application

The design of MRJ shares many features with Hadoop at the application interface level (and also, e.g., for job submission, setting the configuration parameters, and initialising the framework) as both frameworks are implemented in Java.

The WordCount application is the canonical MapReduce programming example [13]. It counts the number of occurrences of each word in an input text file. The MRJ implementation requires the programmer to define only two methods: `map` and `reduce` since all other operations including task splitting and output sorting are provided directly by the MRJ runtime system.

Figure 1 shows the actual Java source code for the WordCount application, including a small amount of boilerplate initialization code. It is apparent from this listing that the MRJ API abstracts away all the details of the parallelization, runtime scheduling, etc, so the application programmer is enabled to focus on the application logic. The entire implementation of this simple application takes less than 30 lines of Java code. Other common string processing applications such as Grep, StringMatch and ReverseIndex may be implemented simply by altering the `map` and `reduce` methods in Figure 1.

2.2 MRJ Implementation Basis

The distinguishing feature of MRJ is that it exploits a recursive divide-and-conquer approach. The implementation takes advantage of this by relying on work stealing and the Java fork-join framework (part of pre-release version of `java.util.concurrent` package for JDK1.7) [2, 24].

The execution sequence of fork/join parallelism is analogous to the MapReduce pattern. In fork/join parallelism, a given computational task is divided into new subtasks and each subtask is executed in parallel on a separate core. The instantiation of subtasks in parallel is represented by the *fork* operation. The corresponding *join* operation ensures that the main execution context waits for the completion of all subtasks in a *barrier synchronization* before proceeding to the next stage. 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.

Our MRJ framework builds upon the parallelism constructs provided by the Java fork/join framework to take advantage of the parallelism exposed by the functions `map` and `reduce`. Any MapReduce application has a high degree of parallelism, as the particular input to each `map` and `reduce` function is processed without any dependence on other portions of the overall application input.

We initialize a `ForkJoinPool` of worker threads, which will execute `map` and `reduce` methods as `ForkJoinTask` instances. In order to successfully complete any map or reduce phase, each worker thread needs to wait for other worker threads to finish execution before proceeding. The impact of such overhead is significantly reduced in MRJ framework by using a Cilk-style *work-stealing* technique [7]. This way of scheduling the subtasks minimises the load imbalance due to uneven distribution of the computation associated with each task. Each worker thread has a double-ended queue of map or reduce tasks awaiting execution. If any thread's queue is empty, it may steal a task from another thread's non-empty queue.

Dynamic load balancing is useful since some map/reduce tasks may finish 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 on Intel Core i7 may make some cores execute faster. Indeed, heterogeneous multi-core architectures are an ideal target for a dynamic load-balancing system like MRJ.

2.3 MRJ Execution Details

Figure 2 presents an overview of the execution stages of the MapReduce pattern. Partitioning of the input data creates subtasks with equal size of partitioned data units. Worker threads in the `ForkJoinPool` execute the generated subtasks, at the map and reduce stages.

The number of worker threads is generally upper-bounded by the number of cores in the system. The particular task scheduling policy may be configured in the MRJ setup. With *static* scheduling, for each map or reduce phase, the number of subtasks created is equal to the number of worker threads in the fork-join pool. On the other hand, *dynamic* scheduling allows for more *fine-grained* parallelism. Many more subtasks are created than there are worker threads in the fork-join pool. The actual number of subtasks can be varied dynamically in the framework. This fine-grained parallelism improves load balance, cache locality and scalability (since it allows greater scope for the work-stealing mechanism to operate). However, generating more subtasks can increase the overhead due to task creation, garbage collection and scheduling. In all our experiments, we find that we can achieve good performance using around eight times as many subtasks as there are available threads. This is the recommended ratio for fork-join tasks to worker threads, in the fork-join library.

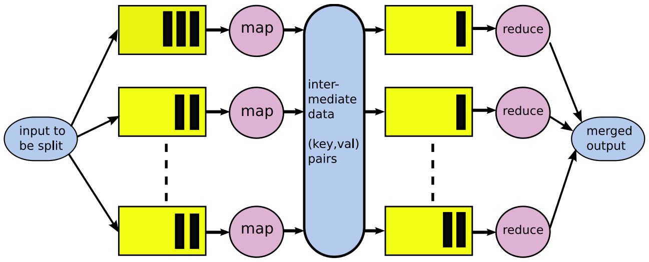

Figure 2. Schematic diagram for MapReduce execution

Vendor	Intel
Codename	Nehalem
Architecture	Core i7
Cores/Contexts	4/2
Per-core L1 i/d	32KB/32KB
Per-core L2	256KB
Shared L3	8MB
Core freq	2.67GHz
RAM size	6GB
OS	Linux 2.6.31
JVM (1.6)	14.0-b16
max fixed heap	4GB

Table 1. Multi-core architecture for evaluating MRJ scalability

In order to improve performance for recursive calls to the MapReduce function, a job-chaining functionality, similar to Hadoop's, has been implemented in MRJ. Job chaining enables multiple calls to map and reduce phases during a single MapReduce execution. This feature is required, for example, to implement the `kmeans` benchmark in which the runtime iterates through the map and reduce stages to compute the final cluster for a given set of coordinates. The benefit of using the job-chaining feature is that all the worker threads and data structures created during the first Map and Reduce stages are *reused*, thus reducing memory management overhead.

3. Scalability Study

In this section, we evaluate the scalability of our MRJ framework on a commodity multi-core architecture described in Table 1. This is a shared-memory, uniform memory access multi-core processor.

All experiments take place using a fixed JVM heap size, that is large enough to minimize any GC activity. (We explore the impact of GC in more detail in Section 4.) Each experiment is run five times, and the arithmetic mean is reported as the result. We use a nanosecond resolution timer. Speedup is calculated as T_1/T_p, where T_1 is the execution time with a single MRJ thread, and T_p is the execution time using p MRJ threads.

Table 2 summarizes the seven MapReduce benchmarks we have ported to MRJ, and their input data sets. The porting process simply involves transliterating the application code from the open-source Phoenix implementation to our Java MRJ framework. (Phoenix is another multi-core MapReduce platform, developed in C [31].)

Figure 3 shows the scalability curves for all benchmarks, for increasing numbers of runtime threads allocated to the Java fork/join pool. The graphs only show the Large input data sets in this section.

The majority of benchmarks do not scale beyond 4 threads on the Core i7 platform, which has 4 cores, each with 2 hyperthread contexts. The intelligent JVM scheduler places the first four MRJ worker threads on separate cores. However hyperthreading does not give any significant performance gain for memory-bound applications. On a cache miss, a hardware thread is context-switched for a different hardware thread. However, if all threads incur frequent cache misses, then they are all stalled and no useful work can be done by the extra contexts. On the other hand, compute-bound applications such as `matrix` and `pca` show some further scaling beyond 4 threads on Core i7, gaining additional benefit from the hy-

benchmark	description	input data
grep	find string occurrences in input text file	S:10MB, M:50MB, L:100MB
kmeans	group 3d points into clusters based on their Euclidean distance	S:100k points, M:250K, L:500K
linearR	compute best-fit line for input data file	S:10MB, M:50MB, L:100MB
matrix	dense integer matrix multiplication	S:1000x1000 values, M:2000x2000, L:3000x3000
pca	principal components analysis on an integer matrix	S:1000x1000 values, M:2000x2000, L:3000x3000
sm	search input text file for a word	S:10MB, M:50MB, L:100MB
wc	count instances of each unique word in input text file	S:10MB, M:50MB, L:100MB

Table 2. MapReduce Benchmarks evaluated in the MRJ framework

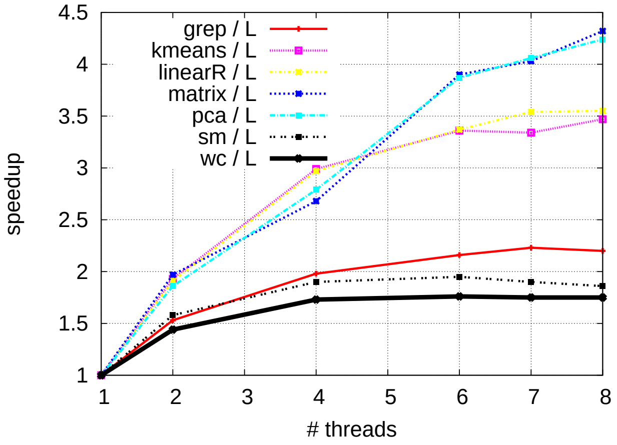

Figure 3. Scalability of MRJ benchmarks on Intel Core i7

benchmark	*L3 miss rate*	*σ*	*DLTB miss rate*	*σ*
grep	2283	90	556	41
kmeans	1547	30	38	2
linearR	2430	244	254	14
matrix	175	5	52	2
pca	231	5	48	1
sm	2377	161	251	14
wc	2271	89	617	26

Table 3. Oprofile samples for miss rates on Core i7, means and standard deviations for five runs of each benchmark

perthreading contexts. These benchmarks have more regular memory access patterns, which gives them relatively good cache locality.

Similar trends in behaviour for compute-bound and memory-bound benchmarks on simultaneous multi-threaded (SMT) multicore architectures has been observed for the PARSEC benchmark suite [6].

In order to characterize which MRJ benchmarks are memory-bound and which are compute-bound, we run a set of simple profiling experiments. We use the Linux *oprofile* tool, to sample the `LLC_MISSES` and `DTLB_MISSES` hardware performance counters on the Intel Core i7 platform. We sample every 6000th retired memory load event that misses the L3 cache, and every 6000th memory access that incurs a DTLB miss, during MRJ benchmark execution. In order to make sure all cores are roughly equally loaded, we profile with 8 MRJ threads running. If we divide each sample count by the benchmark execution time, we should obtain values that correlate roughly with the L3 cache miss rate and DTLB miss rate for this benchmark. We say that MRJ benchmarks with high cache miss rates are memory-bound, conversely those with low cache miss rates are CPU-bound. Benchmarks with a low DTLB miss rate exhibit good locality of reference, whereas those with a high DTLB miss rate do not have good locality. Table 3 reports the mean of 5 measurements for each benchmark using 8 threads with large heap sizes to minimize GC.

We note that `matrix` has comparatively low miss rates, which account for its near-linear scalability. Since matrix multiplication is implemented as simple linear array traversals, it exhibits excellent cache locality. The `kmeans` benchmark has *higher* miss rates, since it has little spatial locality. In the `kmeans` data, close points in the 3-d co-ordinate space are randomly distributed in the Java heap. The `grep` and `wc` benchmarks are operating on `String` based data, and have high miss rates.

4. Impact of Garbage Collection

Since MRJ is implemented in Java, it relies on the underlying JVM to provide garbage collection (GC) services. The impact of GC is more significant for execution with relatively small heap sizes: In the earlier experiments in Section 3, heap space is explicitly fixed at 4GB to minimize GC effects.

The Java 1.6 runtime provides three standard GC algorithms: serial, parallel, and concurrent. The *serial* collector is a stop-the-world GC (i.e. all application threads must be paused before GC takes place) which uses a single thread to perform all collection. Thus there is no inter-thread communication overhead for serial GC. The *parallel* collector is also a stop-the-world GC. However it uses multiple threads to perform the collection, thus it can outperform the serial collector for applications with large data sets, running on architectures that provide multiple hardware threads. The *concurrent* collector [29] performs most of its work while the application threads are running. This minimizes GC pause time, which improves response time for interactive applications. However there is a runtime overhead to support concurrent GC, which means parallel GC generally gives better overall execution times.

When we run an MRJ application with n threads, we allow the parallel and concurrent GCs to use n threads also. (The number of threads to use for GC can be specified as a JVM command-line parameter.)

For MRJ application execution, there are two major causes of heap memory usage. (i) The size of the input data set affects the heap space requirements. (ii) The specified number of map and reduce threads, and the granularity of the tasks, affects heap space requirements. This is because each individual task resolves to a

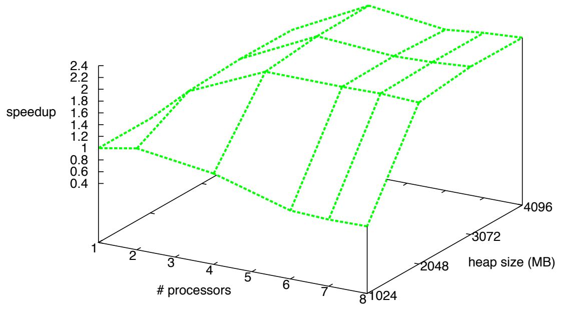

Figure 4. Scalability of grep degrades with increasing numbers of processors, for small heap sizes

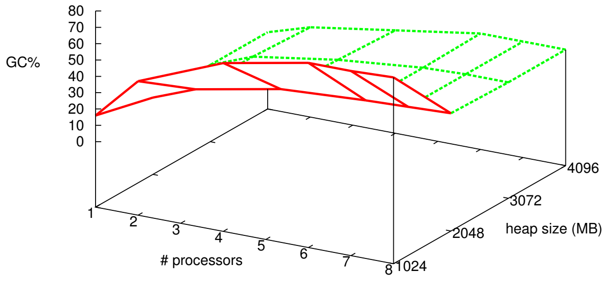

Figure 5. GC overhead increases with the number of processors, more significantly for small heap sizes

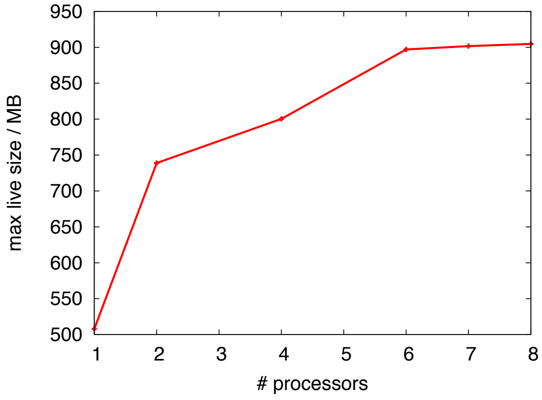

Figure 6. Max live size increases with the number of fork/join threads

`ForkJoinTask` instance, which requires its own book-keeping data structures in memory.

In the remainder of this section, we present some results that appear to be MRJ performance anomalies, but can be explained by considering the GC interaction with MRJ.

4.1 Inverse Scaling for Small Heaps

The first potentially surprising result is that the grep benchmark performance degrades with increasing numbers of threads, for small heap sizes. This is the case for all GC algorithms; Figure 4 illustrates the point for the serial GC, using the large input data set. When the heap size is 1024MB, the speedup drops below one when the number of processors is above two. However at all larger heap sizes, the speedup increases until the number of processors reaches four, then it hits a plateau (as shown earlier in Figure 3).

The reason for this slowdown at 1024MB, as the number of threads increases, is *heap space pressure* due to increasing memory consumption from the Java fork-join threads.

From an analysis using the Java *hprof* heap profiling tool, we see that the *total* amount of memory allocated in grep is generally invariant, no matter how many worker threads are allocated to the Java fork/join pool. However the amount of *live* data on the heap is proportional to the number of concurrently executing worker threads. This is because each thread has its own thread-local data structures, such as `StringBuffer` objects and backing `char` arrays. Thus with more concurrent worker threads, there is a higher proportion of live data on the heap, which increases heap space pressure. This causes a greater number of garbage collections, hence the slowdown in overall application execution time. Figure 5 shows how the proportion of application execution time spent in GC increases with the number of threads, for the 1024MB heap size with serial GC. With a single fork/join worker thread, the GC time is 16%. This rises to 74% with eight fork/join worker threads. At larger heap sizes, the proportion of time spent in GC is much less significant.

Figure 6 shows how the max live size statistic varies with the number of fork/join threads, for grep with the Large input in a fixed 1024MB heap. The max live size measure is an approximation of the minimum memory requirement for a program, based on the high-water-mark of live data over all GC events throughout the program execution. Note how this high-water-mark approaches the fixed heap capacity for larger numbers of fork/join threads. We observe similar behaviour with the wc benchmark. At 1024MB it fails to execute the Large input data due to `OutOfMemory` errors for any more than two worker threads.

4.2 Relative GC Performance is Input Dependent

It is apparent that the behaviour of applications is often highly input dependent. Mao et al. [27, 33] establish the influence of Java program inputs on the minimum heap size required for successful execution, and the relative performance of various garbage collection algorithms.

We have three input data sets for each MRJ benchmark, classified as Small, Medium and Large. Using Small inputs, MRJ application performance is generally similar irrespective of the selected GC algorithm. This is because there is relatively little allocation, so the GC does not have much work to do. On the other hand, with larger inputs (or equivalently, smaller heap sizes [1]), there are significant differences in performance between the various GC algorithms. As a general rule, parallel GC outperforms serial GC on larger inputs, when the number of threads is above two. Since parallel GC uses multiple threads to perform collection, it reduces the GC pause time relative to serial GC.

Table 4 illustrates the point that the relative performance of each GC algorithm varies with application input. For the wc MRJ application, we evaluate each input data set with each GC algorithm, in a variety of heap sizes ranging from 1 to 4GB. So for each `(inputsize, heapsize, numthreads)` combination, we have three execution times, one per GC algorithm. We define a GC's performance as *good* if it gives an execution time within 10% of the optimal time from any GC algorithm for this `(inputsize, heapsize, numthreads)` case.

Table 4 enumerates all the cases: each table cell corresponds to a single case. For each case, the cell label indicates which GC algorithms are *good*. (Note that between one and three GC algorithms can be *good* for a particular case[2].)

For the Small input, the serial GC appears to be *good* for the majority of cases. On the other hand, for the Large input, the parallel and concurrent GC algorithms appear to outperform the serial GC for many cases.

Other benchmarks, such as matrix and kmeans, have negligible GC overhead at all the heap sizes we tested. These benchmarks do not exhibit significant GC performance variation with input.

4.3 Relative GC Performance is Application Dependent

It is a well-established fact that the relative performance of different GC algorithms is dependent on the characteristics of the application being executed [17, 28, 35] particularly at smaller heap sizes.

As one might expect, parallel GC (which is the default option on server class JVMs) generally outperforms serial and concurrent, especially for larger numbers of threads. However this is not always the case. For instance, the sm benchmark performs better with concurrent instead of parallel GC. For example in a 2GB heap, the sm benchmark with Large input executing on 8 threads takes an average of 2.26s with parallel GC, but only 0.61s with concurrent GC.

The unusual behaviour of sm can be understood by referring to its high object death rate. Figure 7 shows the death rates for all MRJ benchmarks. We compute the death rate as the total amount of collected garbage during the program execution divided by the total execution time of the program. The bar chart shows that sm has a significantly higher death rate than the other benchmarks.

Concurrent GC identifies and collects dead objects in the background, while the application is executing. This avoids long pauses for full-heap scans. We suggest that since the sm benchmark creates large numbers of short-lived objects, these are collected very shortly after their allocation, reducing overall heap consumption and increasing data locality. This has a significant impact on overall application performance.

As earlier, we use oprofile to quantify cache locality. We measure the L3 cache miss rates for sm with different GC algorithms, at a sampling rate of 6000, for 8 threads, Large inputs, 2GB heap. Table 5 shows this data, demonstrating that the cache locality is much better for concurrent GC.

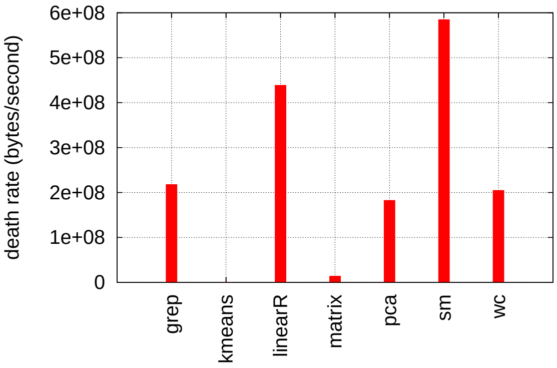

Figure 7. Object death rates for the MRJ benchmarks, running with 1 thread, serial GC, large inputs, 2048MB heap

GC	L3 cache miss rate	σ
serial	1973	57
parallel	3540	86
concurrent	1054	96

Table 5. Mean and standard deviation of L3 cache miss sample rates from oprofile over 10 runs of sm, for each GC algorithm

5. Auto-Tuning Garbage Collection for MRJ

In this section, we investigate the use of machine learning to select a suitable GC policy for each MRJ program. Previous sections have demonstrated that the GC performance is dependent on a number of factors, including benchmark characteristics, JVM heap size, and number of threads. In general, it is not straightforward to select the optimal GC policy without exhaustively testing all candidate policies. We show that it is *not* the case that a single GC policy is uniformly good for all programs.

Our objective is to implement a *GC auto-tuning framework* for MRJ. Given a new benchmark, input dataset, and system constraints on heap size and number of threads, the auto-tuning framework predicts a suitable GC policy. For this initial investigation, the GC policy consists of the GC algorithm (serial, parallel or concurrent) and the heap space layout (young:old generation ratio of 1:2 or 1:8). Thus there are six different GC policies, based on the various combinations of these configuration options. Other aspects of memory management could be incorporated into this auto-tuning framework, but these seem to be the most significant concerns relating to GC performance.

5.1 Motivation

Why should we have a specific GC tuning framework for MRJ? We feel there are several compelling reasons:

1. MRJ application developers will not be interested in the mechanics of GC. It is a low-level cross-cutting concern, to be addressed by the software platform architects, rather than application developers or users.
2. GC is crucial since it has effects on runtime behaviour, including thread scheduling, cache locality, response time, and overall execution time.
3. We can make use of MRJ-specific information as features to characterize applications, in order to predict appropriate GC

[1] Although this is a similar point to earlier, there is a distinction. Section 4.1 showed that small heaps magnify the impact of GC. Section 4.2 shows that in relatively small heaps, some GC variants are noticeably more effective than others.

[2] In a few cases the benchmark throws an `OutOfMemory` error for all GC algorithms, so no GC algorithm is good.

	input	Small			Medium			Large		
	heap/GB	1	2	4	1	2	4	1	2	4
#threads	8	P	S,P	S,P	C	P	P		C	P
	7	P	S,P	S,P	C	P	P		C	P
	6	P	S,P	S,P	C	P,C	P		C	P,C
	4	P	S,P	S,P	S,P,C	P	P		S,P,C	P,C
	2	P	S,P	S,P	P	P	P	C	P	P
	1	S	S,P	S,P	S,P,C	S,P,C	S,P,C	C	S,P	S,P

Table 4. (S)erial, (P)arallel and (C)oncurrent benchmarks exhibit 'good' performance for different scenarios depending on input for the wc benchmark

policies. This might include allocated object types and sizes. For example, large `int` arrays tend to last for the lifetime of the application (consider the matrices in matrix multiplication or pca). On the other hand, smaller objects are more short-lived, such as `StringBuffer` objects in the wc or grep applications.

4. An accurate GC auto-tuning framework ought to give a time saving. When a new MapReduce application arrives, the framework will predict an appropriate GC policy for its execution given the system constraints. Without this auto-tuning, we would have to do exhaustive profiling, i.e. evaluate each individual GC policy. With auto-tuning, we can gather static features from the benchmark code and system parameters, and perhaps gather dynamic features from a small number of trial executions.

5.2 Data Set Generation

This section outlines the *features* that we use to characterize individual MRJ benchmark executions with specific constraints. These features will be used as inputs to the learning algorithm, which will predict a good GC policy.

The MRJ system constraints specified by the user are the JVM fixed heap size and the number of threads. In general, a larger heap and more threads should improve the overall performance of the application. However there may be other system-level restrictions that determine a particular MRJ application's resource usage.

Other features are dynamic characteristics of the individual application and its input data. We record the number of minor and major GCs that take place on a single run through the program, along with the time spent in GC as a proportion of the overall execution time. We measure the total amount of memory dynamically allocated by the application during its execution, and the proportion of this allocated memory occupied by `String` objects, and `int` arrays. These two are the most common datatypes manipulated by our MRJ benchmarks.

All of these dynamic features are collected on a single run of an MRJ program, using serial GC with 1 MRJ thread. In fact, we have two trial executions, one with a young:old heap ratio of 1:2, and the other 1:8. This gives us twice as many features for the GC-specific data. We make use of the extensive JVM profiling features, such as the `hprof` agent library and verbose GC logging. Table 6 presents a summary of all the features collected for each (benchmark,input) combination.

Next we collect the execution times on the Intel Core i7 machine. These execution times form the basis for the target class in our data set. For each `(bm,input,heapsize,numthreads)` combination, we run 5 experiments for each GC policy, and compute the arithmetic mean execution time.

feature	*type*	*how collected*
heap size (MB)	integer	system parameter
# MRJ worker threads	integer	system parameter
# minor GCs (x2)	integer	trial execution
# major GCs (x2)	integer	trial execution
% GC time (x2)	real	trial execution
bytes allocated	integer	trial execution
% String alloc'd	real	trial execution
% int array alloc'd	real	trial execution

Table 6. Summary of features collected for each (benchmark,input) combination

From this information, we generate the data set for the machine learning algorithm as follows. For each experiment, there will be an *optimal* GC policy, i.e. the policy that gives the lowest execution time. We say that a GC policy is *good* for an application execution if the time is shorter than 95% of the time with the default policy, and if the time is no longer than 110% of the optimal time. (These thresholds require some experimentation, informed by the standard deviations of the execution time distributions for each application.)

So, for each application execution, we can say which GC policies are good, and which are not. We build a distinct classifier for each policy, to predict whether or not that policy is good for a particular experiment.

We create classifiers using leave-one-out cross-validation (LOOCV). We set up a training set to include experimental data for all benchmarks except one, and then *train* the classifiers using this data. Subsequently we *test* the generated classifiers on the missing benchmark to obtain fair predictions. (For n benchmarks, we have n rounds of LOOCV, each time eliminating one of the n benchmarks from the training set.)

The following section gives details of the classification technique we use.

5.3 Prediction Technique

We instantiate a separate classifier for each GC policy p, that predicts whether p enables good performance for an MRJ benchmark and its input, given their set of features. This is a variant of one-versus-all prediction [32], which is a commonly accepted way of decomposing a multi-class problem.

Each individual classifier is a *random forest*, which is an ensemble predictor consisting of many decision trees [8]. Decision trees automatically select the most relevant features for the classification problem, and discard less relevant features [30]. The ensemble technique introduces a small amount of randomness into the selected features to make the overall classifier more robust. We use the Weka [19] implementation of random forests, with all the parameters set to default except that we have 20 trees for each forest.

It may be the case that multiple individual classifiers predict that a program/input will be good with this GC policy. We apply the predictors in a simple *cascade* of classifiers [36]. This means that we impose an order on the policies, then apply the predictors in this order. If at any stage in the cascade, the predictor indicates good performance, then we go with this policy, otherwise we try

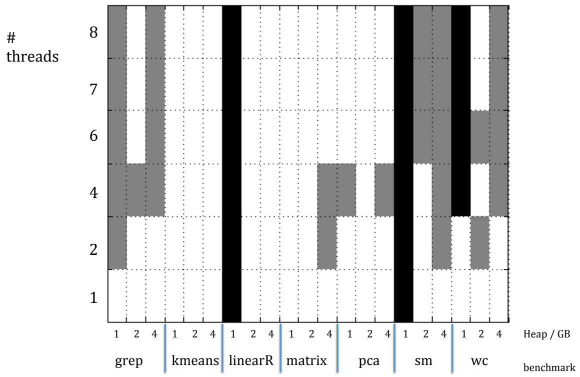

Figure 8. Heatmap showing GC auto-tuning performance relative to optimal policy, for each benchmark execution, at varying heap sizes (1,2,4 GB) and numbers of threads, with Large input data sets. White cells (around 75% of complete experiments) indicate that the predicted GC policy was close to the the optimal policy. Grey cells indicate that the selected GC policy gave clearly sub-optimal performance. Black cells indicate that the particular experiment did not complete with any GC policy due to OutOfMemory errors.

the next predictor in the cascade. If no good policy is predicted, then we use the *default* HotSpot server GC policy, which is parallel GC with a 1:2 young:old heap layout.

The actual order of classifiers in our cascade, in terms of their predicted GC policies expressed as *algorithm-ratio* is: concurrent-1:8, concurrent-1:2, serial-1:8, serial-1:2, parallel-1:8, parallel-1:2. This ordering is another part of the learning that could be tuned directly. However to avoid complications we settled on this fixed order that gives good overall results.

5.4 Evaluation of Performance Tuning

We test the performance of this cascade of classifiers by using it to predict suitable GC policies for the (benchmark,input) combinations that we excluded from the LOOCV training data. We only give results for Large input data sets. For smaller inputs, there is less variation between the GC algorithms' performance.

Figure 8 presents the results of the GC auto-tuning, as a heatmap. For each benchmark, with a particular heapsize and number of threads, we compare the overall execution performance with the predicted GC policy against the best performance (optimal) of any of the six specified GC policies. In 82 cases out of 110 completed experiments, the predicted policy gives comparable performance with the optimal policy.

Even in cases that do not perform as well as the optimal time, we may still outperform the default policy, and thus get some improvement. For instance, this scenario occurs for the sm benchmark when using larger numbers of threads.

Figure 9 shows another map over the same experiments. Here, we report whether our predictive GC policy gives performance at least as good as the default GC policy. White cells indicate that the corresponding experiments are not more than 10% slower than the default execution time. In only 11 cases (out of 110) is the predicted performance significantly worse than default. These cases are indicated by grey cells in the heatmap.

The overall worst case is a 30% slowdown[3]. Across all 110 experiments, the maximum speedup of the predicted policy over the default policy is 6.0 times[4], and the geometric mean speedup over all experiments is 1.1 times.

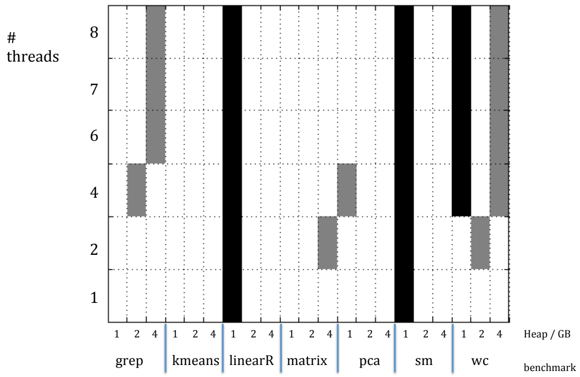

Figure 9. Heatmap showing GC auto-tuning performance relative to default GC policy, on benchmarks with Large inputs. White cells (around 90% of complete experiments) indicate that the predicted GC policy was at least as good as the default policy. Grey cells indicate that the selected GC policy gave worse than default performance. Black cells indicate that the particular experiment did not complete with any GC policy due to OutOfMemory errors.

The predictor suggests using the default policy in only 5 out of 110 cases. However note that GC policy does not make much overall difference for benchmarks that do little dynamic memory management, such as matrix and pca.

6. Related Work

The original work on MapReduce [13, 14] applies to compute-clusters. Ranger et al. describe the first application of MapReduce to multi-core processors [31]. Yoo et al. later extend this to non-uniform memory access architectures [38]. They discuss the scalability of memory management in terms of `malloc` and `mmap` functions. Since their work is in C/C++, they do not have automatic memory management overhead.

Since the Hadoop MapReduce system [1] is implemented in Java, its performance can be affected by runtime garbage collection like our MRJ framework. The Hadoop developers give a limited amount of advice[5] on performance tuning for GC with Hadoop. However they do not appear to consider tuning for application-specific behaviour. Their chief concern is to minimise pause time by using a concurrent GC algorithm with a small nursery space.

The techniques for understanding interactions between an application and a runtime system are inspired by Hauswirth's work on vertical profiling [20]. He considers all levels from architecture through to application, to explore reasons for apparent anomalous behaviour. In the present paper, we have attempted to follow his holistic approach by considering and relating performance data from hardware counters, through JVM level statistics, to application performance.

A recent paper analysing the behaviour of multi-threaded Java workloads on multi-core systems shows that conventional memory management techniques do not scale to large multi-core envi-

[3] For execution of grep with 4GB, 8 threads, using the serial-1:8 GC policy.

[4] For execution of sm with 2GB, 2 threads, using the serial-1:8 GC policy. (Note that although both examples given use the serial-1:8 policy, all the other GC policies are predicted for various examples in the space of experiments.)

[5] `http://wiki.apache.org/hadoop/PerformanceTuning`

ronments, and require adaptation [40]. They identify an *allocation wall*, which refers to the maximum rate of allocation from concurrent JVM threads. In our limited experience with MRJ benchmarks, we have not hit the allocation wall. Most MRJ programs create major data structures up-front, then operate on these throughout the map or reduce phases, only allocating small objects (`String` instances, etc) during MapReduce computation. We have not hit a corresponding *de-allocation wall*, except as in Section 4.1 when the heap size is relatively small.

Singer et al. investigate the automatic selection of garbage collection algorithms for a set of standard Java benchmarks, using different feature sets and learning algorithms [34]. In the specific context of MapReduce for Java, we feel that the current set of features and learning strategies fit better.

We note that, in general, the application of machine learning to Java runtime performance auto-tuning is a growing trend [26, 39].

7. Conclusions

This paper presents MRJ: a Java-based framework for MapReduce parallelism that targets conventional multi-core architectures. We have demonstrated its scalability of performance, with increasing numbers of threads allocated to the underlying Java fork/join pool. We have highlighted the interactions between MapReduce benchmarks and the garbage collector, and shown how a machine-learning GC auto-tuning policy can improve runtime performance.

We intend to release MRJ shortly as an open-source project, since we hope it will be useful to a wider community.

Acknowledgements

Thanks to Angela Demke Brown for her careful and considerate shepherding of this paper. We are also grateful to EPSRC for supporting this research, under grant EP/G000662/1. Mikel Luján holds a Royal Society University Research Fellowship.

References

[1] Hadoop: Open source implementation of mapreduce. `http://lucene.apache.org/hadoop/`.

[2] Jsr-166, prelease of java.util.concurrent package. `http://gee.cs.oswego.edu/dl/concurrency-interest/index.html`.

[3] Organizations using hadoop. `http://wiki.apache.org/hadoop/PoweredBy`.

[4] E. Allen, D. Chase, C. Flood, V. Luchangco, J.-W. Maessen, S. Ryu, and G. L. Steele Jr. Project Fortress: A multicore language for multicore processors. *Linux Magazine*, Sep 2007.

[5] M. Armbrust, A. Fox, R. Griffith, A. D. Joseph, R. Katz, A. Konwinski, G. Lee, D. Patterson, A. Rabkin, I. Stoica, and M. Zaharia. A view of cloud computing. *Communications of the ACM*, 53:50–58, April 2010.

[6] M. Bhadauria, V. M. Weaver, and S. A. McKee. Understanding PARSEC performance on contemporary CMPs. In *Proceedings of the 2009 International Symposium on Workload Characterization*, October 2009.

[7] R. Blumofe, C. Joerg, B. Kuszmaul, C. Leiserson, K. Randall, and Y. Zhou. Cilk: An efficient multithreaded runtime system. In *Proceedings of the fifth ACM SIGPLAN symposium on Principles and practice of parallel programming*, pages 207–216, 1995.

[8] L. Breiman. Random forests. *Machine Learning*, 45(1):5–32, 2001.

[9] P. Charles, C. Donawa, K. Ebcioglu, C. Grothoff, A. Kielstra, C. von Praun, V. Saraswat, and V. Sarkar. X10: An object-oriented approach to non-uniform clustered computing. In *Proceedings of OOPSLA 2005*, 2005.

[10] C. Chu, S. Kim, Y. Lin, Y. Yu, G. Bradski, A. Ng, and K. Olukotun. Map-reduce for machine learning on multicore. In *Advances in Neural Information Processing Systems*, page 281, 2007.

[11] S. Daruru, N. M. Marin, M. Walker, and J. Ghosh. Pervasive parallelism in data mining: dataflow solution to co-clustering large and sparse netflix data. In *KDD '09: Proceedings of the 15th ACM SIGKDD international conference on Knowledge discovery and data mining*, pages 1115–1124, 2009.

[12] M. de Kruijf and K. Sankaralingam. MapReduce for the CELL B.E. architecture. *IBM Journal of Research and Development*, 53(5), 2009.

[13] J. Dean and S. Ghemawat. MapReduce: simplified data processing on large clusters. In *Proceedings of the 6th symposium on operating systems design and implementation*, pages 137–150, 2004.

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

[15] D. DeWitt and M. Stonebraker. MapReduce: A major step backwards. *The Database Column*, 2008.

[16] J. Ekanayake, S. Pallickara, and G. Fox. MapReduce for data intensive scientific analyses. In *Fourth IEEE International Conference on eScience*, pages 277–284, 2008.

[17] R. Fitzgerald and D. Tarditi. The case for profile-directed selection of garbage collectors. In *Proceedings of the 2nd International Symposium on Memory Management*, pages 111–120, 2000.

[18] Google. Appengine. `http://code.google.com/appengine/`.

[19] M. Hall, E. Frank, G. Holmes, B. Pfahringer, P. Reutemann, and I. H. Witten. The WEKA data mining software: An update. *SIGKDD Explorations*, 11(1), 2009.

[20] M. Hauswirth, P. Sweeney, A. Diwan, and M. Hind. Vertical profiling: understanding the behavior of object-oriented applications. *ACM SIGPLAN Notices*, 39(10):251–269, 2004.

[21] B. He, W. Fang, Q. Luo, N. Govindaraju, and T. Wang. Mars: a MapReduce framework on graphics processors. In *Proceedings of the 17th international conference on parallel architectures and compilation techniques*, 2008.

[22] Intel. Single-chip cloud computer, 2009. `http://techresearch.intel.com/UserFiles/en-us/File/terascale/SCC-Overview.pdf`.

[23] G. Kovoor, J. Singer, and M. Lujan. Building a Java mapreduce framework for multi-core architectures. In *Proceedings of the Third Workshop on Programmability Issues for Multi-Core Computers*, pages 87–98, 2010.

[24] D. Lea. A java fork/join framework. In *Proceedings of the ACM conference on Java Grande*, 2000.

[25] D. Leijen, W. Schulte, and S. Burckhardt. The design of a task parallel library. In *Proceeding of the 24th ACM SIGPLAN conference on Object oriented programming systems languages and applications*, pages 227–242, 2009.

[26] F. Mao and X. Shen. Cross-input learning and discriminative prediction in evolvable virtual machines. In *Proceedings of the 7th annual IEEE/ACM International Symposium on Code Generation and Optimization*, pages 92–101, 2009.

[27] F. Mao, E. Z. Zhang, and X. Shen. Influence of program inputs on the selection of garbage collectors. In *Proceedings of the 2009 ACM SIGPLAN/SIGOPS international conference on Virtual execution environments*, pages 91–100, 2009.

[28] T. Printezis. Hot-swapping between a mark&sweep and a mark&compact garbage collector in a generational environment. In *Proceedings of the 1st Java Virtual Machine Research and Technology Symposium*, pages 171–184, 2001.

[29] T. Printezis and D. Detlefs. A generational mostly-concurrent garbage collector. In *Proceedings of the 2nd international symposium on Memory management*, pages 143–154, 2000.

[30] J. R. Quinlan. Induction of decision trees. *Machine Learning*, 1(1):81–106, 1986.

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

[32] R. Rifkin and A. Klautau. In defense of one-vs-all classification. *Journal of Machine Learning Research*, 5:101–142, 2004.

[33] X. Shen, F. Mao, K. Tian, and E. Z. Zhang. The study and handling of program inputs in the selection of garbage collectors. *ACM SIGOPS Operating Systems Review*, 43:48–61, July 2009.

[34] J. Singer, G. Brown, I. Watson, and J. Cavazos. Intelligent selection of application-specific garbage collectors. In *Proceedings of the 6th International Symposium on Memory Management*, pages 91–102, Oct 2007.

[35] S. Soman, C. Krintz, and D. F. Bacon. Dynamic selection of application-specific garbage collectors. In *Proceedings of the 4th International Symposium on Memory Management*, pages 49–60, 2004.

[36] P. Viola and M. Jones. Rapid object detection using a boosted cascade of simple features. In *Proc. IEEE Conf. on Computer Vision and Pattern Recognition*, pages 511–518, 2001.

[37] H.-C. Yang, A. Dasdan, R.-L. Hsiao, and D. S. Parker. Map-reduce-merge: simplified relational data processing on large clusters. In *Proceedings of the 2007 ACM SIGMOD international conference on Management of data*, pages 1029–1040, 2007.

[38] R. Yoo, A. Romano, and C. Kozyrakis. Phoenix rebirth: Scalable MapReduce on a NUMA system. In *Proceedings of the International Symposium on Workload Characterization*, 2009.

[39] C. Zhang and M. Hirzel. Online phase-adaptive data layout selection. In *ECOOP 2008 Object-Oriented Programming*, pages 309–334, 2008.

[40] Y. Zhao, J. Shi, K. Zheng, H. Wang, H. Lin, and L. Shao. Allocation wall: a limiting factor of Java applications on emerging multi-core platforms. *ACM SIGPLAN Notices*, 44(10):361–376, 2009.