The final publication is available at:
http://www.springerlink.com/content/1gl316463585084p
http://dx.doi.org/10.1007/s11042-012-1256-0”

Practical scalable image analysis and indexing using Hadoop

Jonathon S. Hare · Sina Samangooei · Paul H. Lewis

2012-08-06

Abstract The ability to handle very large amounts of image data is important for image analysis, indexing and retrieval applications. Sadly, in the literature, scalability aspects are often ignored or glanced over, especially with respect to the intricacies of actual implementation details.

In this paper we present a case-study showing how a standard bag-of-visual-words image indexing pipeline can be scaled across a distributed cluster of machines. In order to achieve scalability, we investigate the optimal combination of hybridisations of the MapReduce distributed computational framework which allows the components of the analysis and indexing pipeline to be effectively mapped and run on modern server hardware. We then demonstrate the scalability of the approach practically with a set of image analysis and indexing tools built on top of the Apache Hadoop MapReduce framework. The tools used for our experiments are freely available as open-source software, and the paper fully describes the nuances of their implementation.

1 Introduction

In the early days of computer vision and information retrieval the analysis techniques developed were trained and benchmarked using relatively small datasets [45]. These datasets were used primarily as proofs of concept, aiming to validate the ability of increasingly complex algorithms in computationally viable time. Larger datasets were avoided due to storage limitations, cost of computational power and time restrictions.

In recent years the image corpuses available on the internet have grown far beyond these small datasets: In August 2011 flickr hosted their 6 billionth image[1]; as of September 2011 popular image services like photobucket[2] host near 9 billion images; during the New Year celebrations of 2011, facebook reported over 750 million images were uploaded in one evening[3]. If we include all web graphics, news article multimedia and all video sources which can be treated as streams of images, a sense of the scale of the number of images one might wish to search, analyse and process on the internet becomes more apparent.

Developing technologies which can address the extremely large multimedia datasets now available on the web requires experimentation beyond traditional, comparatively tiny, test datasets. With ever cheaper computational power and storage available in the form of scalable cloud solutions and large

Jonathon S. Hare · Sina Samangooei · Paul H. Lewis
School of Electronics and Computer Science
University of Southampton
Southampton
SO17 1BJ
United Kingdom
Tel.: +44 (0)23 8059 {5415,3715}
E-mail: {jsh2,ss,phl}@ecs.soton.ac.uk

[1] `http://www.flickr.com/photos/eon60/6000000000/`
[2] `http://photobucket.com/`
[3] `http://twitter.com/#!/facebook/status/22372857292005376`

scale computational clusters [1], it has become possible to extend the techniques developed in computer vision and multimedia information retrieval to address internet scale datasets. Towards exploiting this potential, the computer vision and machine learning community has developed a range of techniques which take advantage of various forms of computation power and parallelism which addresses data in its modern scale.

The aim of this paper is three-fold; firstly we aim to present a review of different types of computational models and techniques for processing large datasets. Secondly, we demonstrate how the basic MapReduce processing framework can be more effectively used on modern hardware using a hybrid approach that can be implemented within Apache Hadoop. Finally, and most importantly, we present a case study in which we investigate and evaluate the optimal use of the hybrid framework to help achieve large scale image retrieval, facilitating the transition between research using traditional retrieval databases of thousands of images to potentially billions of images given a sufficiently large compute cluster. Implementations of the described techniques are being made publicly available as open-source software tools for the community to experiment with.

To the best of our knowledge this work is the first to investigate and describe how to optimally process and index large image corpora in a scalable distributed manner, and certainly the first to provide a free open-source baseline implementation. In addition, the implementation aspects of a number of the algorithms required for image indexing is also novel. A good example of this is our hybrid MapReduce approximate k-means algorithm, which takes numerous steps to be fast, use memory efficiently, and minimise data transfer costs, whilst at the same time being able to deal with high dimensional data and very large numbers of clusters.

2 Trends in Large Scale Data Analysis

An approach to dealing with large amounts of data in many computational tasks is the subdivision of the task such that parts of the data can be dealt with separately, and therefore potentially in parallel. The most basic form is parallelisation on a single machine, but when dealing with larger amounts of data this will inevitably hit bottlenecks such as system I/O limits and therefore parallelism across multiple machines has inevitably been considered. In their 2006 review Asanovic et al [5] provide a comprehensive discussion on the uses and designs of modern parallel computing hardware, systems and applications. Firstly, they argue that modern computing systems have taken irreversible steps towards parallel architectures. Given that this is the case, they set about identifying a set of equivalence classes for computational problems drawn from applications throughout various scientific fields, each of which can in some way take advantage of these novel parallel architectures. Significantly, they note that various problems in machine learning can take advantage of parallel computing, including processing with large amounts of data. The importance of parallelism in machine learning is further exemplified by many large-dataset and parallelism conferences and workshops which have run world wide to discuss this potential in recent years. These include: parallelism workshops held at the NIPS conference[4] since 2007 and the international workshop on MapReduce[5], running since 2010.

Approaches taken when performing large scale data mining tasks take many forms. Efforts can be roughly split into those designed to take advantage of some specialised parallel architecture on a single machine, e.g.: system CPU threads, GPUs or FPGAs; and those harnessing the potential of distributed systems of single machines, e.g.: traditional Computer Grid systems; message passing and message queue interfaces such as MPI and RabbitMQ; and, increasingly popularly, the data parallelism centric MapReduce framework.

2.1 Single Machine Parallelism

Various forms of single machine parallelism are available often taking the form of some sort of specialised hardware solutions including: multi-CPU machines, FPGAs and GPUs. Over the years many analysis algorithms have been written or modified to take advantage of such systems. Taking multi-CPU architectures as the default option, in this section we explore FPGAs and GPUs as alternative approaches

[4] `http://nips.cc/Conferences/2011/Program/event.php?ID=2518`

[5] `http://www.wikicfp.com/cfp/servlet/event.showcfp?eventid=13548&copyownerid=19220`

to single machine parallelism and show that (currently) they are not suitable for many large scale data analysis problems.

The trend towards inexpensive multi CPU machines has driven an increase in multi-CPU aware analysis techniques. Many programming languages inherently support multiple software threads (Java, .NET, Matlab, etc.) and there also exist several commonly used libraries which support multi-cpu architectures including: OpenMP[6] and SystemC[7]. This level of support and ubiquity means that software CPU threads are the most popular approach to single machine parallelism. There have been many machine learning algorithm implementations which take advantage of multi-cpu systems including: an implementation for parallel Support Vector Machine (SVM) training and classification [16]; transductive reasoning on large scale graphs [46] and learning a regular language from data [2]. Chu et al [9] present a general solution to the parallelisation of machine learning algorithms showing that if algorithms are expressed in summation form they can be made to take advantage of trends in cheap multi-core architectures.

Another single machine solution to parallel algorithm implementation is the use of a machine's Graphical Processing Unit (GPU) to perform computations traditionally performed by the CPU. GPUs contain specialised hardware which perform certain functions much more quickly than the same processes on a standard CPU, and furthermore have the potential to do these calculations in parallel. With the aid of standard frameworks such as OpenCL[8], the GPU's specialised hardware can be used to address general purpose problems beyond graphical rendering Hamada and Iitaka [20]. If carefully translated to work with GPUs, many data analysis tasks can be accomplished in parallel using GPUs. It is therefore not surprising that the use of the GPU has gained much attention over the past few years. Some examples include learning large scale Deep Belief Networks [41] and an interesting application in scalable automatic object detection [10] which allows their algorithm to deal with many more training examples, as compared to just using the CPU, and thus improve its accuracy.

Field-Programmable Gate Arrays (FPGA) allow for another form of single machine parallelism. An FPGA is an integrated circuit which, often aided by some high level language such as Verilog of VHDL, can be electrically programmed to act as almost any kind of digital circuit or system. Where conventional CPUs can only process one instruction per clock cycle, FPGAs can be configured to apply a given function to data several thousand times simultaneously per clock cycle. The successful implementation of data analysis algorithms on FPGAs requires their careful translation to a relatively low level set of instructions, an operation which can map very well for certain problem types, but also prove very difficult for others. These difficulties notwithstanding, there have been several machine learning techniques translated successfully to work in parallel on FPGAs, all reporting significant improvements over standard software implementations on conventional CPU architectures. This includes a neurologically inspired hierarchical bayesian model used for invariant object recognition [42], an implementation of the RankBoost for web search relevance ranking [48] and a low-precision implementation of Support Vector Machines using Sequential Minimal Optimisation [8].

Though both the GPU [6] and FPGA [48] report considerable improvements when compared to software CPU implementations for some applications, these techniques have been used in far fewer applications than threaded software solutions. One reason for this trend is the low level at which programmers must implement algorithms to take advantage of GPUs, FPGAs or other specialised hardware solutions. This programming difficulty is a part of the cost of such hardware solutions. The cost of FPGAs and GPUs is not only increased over comparatively cheap CPUs in terms of hardware, but also in terms of the very specific expertise required to program against such hardware. This in turn means that algorithms are difficult to implement and understand, and therefore difficult to distribute across multiple machines easily. This issue is echoed by Asanovic et al [5] who discuss programability of underlying parallel solutions as a key significance to parallel computing. Furthermore, specifically with regards to GPUs, there are significant considerations which programmers must make when deciding whether any given machine learning problem can be parallelised efficiently using the GPU alone. In their work which examines the true cost of GPU algorithms, Gregg and Hazelwood [19] show that certain classes of algorithm which send large amounts of data to the GPU and extract large amounts of output, take roughly $50\times$ as long to do so as compared to time actually processing the data. This means that reported algorithm speed ups of GPU vs CPU of $100\times$ when dealing with primarily simulated data and minimal GPU data transfer

[6] `http://openmp.org/wp/`

[7] `http://www.systemc.org/home/`

[8] `http://www.khronos.org/opencl/`

are cut down to $2-3\times$ when the algorithm sends and receives large amounts of data to the GPU. With these limited improvements over a single CPU, it becomes beneficial to take advantage of the simpler to program multiple CPU environments over GPUs.

We believe it is the factors of programability and the potential of problem specification incompatibility which results in software CPUs threads being the most common approach in algorithm parallelism. This is also true for attempts to distribute algorithms across multiple machines; i.e. when an individual job is split into tasks to be performed across multiple machines, within each machine the task is commonly further split to work across multiple CPU threads. This being said, there do exist some examples of FPGAs and GPUs being used across multiple machines. For example Shan et al [43] have shown the power of multiple FPGAs harnessed simultaneously using the MapReduce framework. Similarly, Farivar et al [14] demonstrated that the use of multiple GPUs combined using MapReduce in a small 4-node cluster outperforms a CPU cluster of 62 nodes[9], though it should be noted that these results were obtained while testing a simulation algorithm which required minimal data to be sent and received from the GPU to perform the actual calculations, an arguably optimal problem specification for GPUs.

2.2 Cross Machine Parallelism

Parallelism on single machines allows a program to perform the subcomponents of a given task simultaneously and therefore complete the task more quickly, or equivalently allow more data to be dealt with more quickly. Due to various hardware limitations, a single machine cannot be increased in its ability to parallelise indefinitely. This results in the next logical step in parallelism being the combination of the abilities of several interconnected machines in some form of computing cluster. If implemented to correctly use such a cluster, any given algorithm can be scaled to deal with any given size of task (i.e. larger datasets, more simulations etc.) in reasonable periods of time by adding more machines to the cluster. Such systems have been exploited by computational scientists and in resent years various machine learning algorithms have been programmed to scale efficiently across computing clusters. This has been achieved through various means, but two noteworthy schemes are MPI and more recently the popular MapReduce framework.

The MapReduce Framework. MapReduce is a software framework for distributed computation. MapReduce was introduced by Google in 2004 to support distributed processing of massive datasets on commodity server clusters [12]. In a MapReduce system, the data being processed is distributed across the disks of the cluster nodes and the processing task is pushed to the nodes where the data is stored. This is in contrast to more traditional distributed frameworks, where the data is pushed to the computation nodes.

Logically, the MapReduce computational model consists of two steps, Map and Reduce, which are both defined in terms of $< key,\ value >$ pairs. The dataset being processed is also considered to consist of $< key,\ value >$ pairs. For example, the keys could be filenames, and the values could be the actual contents of the files, or the keys could be the line number of a text file, and the value could be the text on the respective line. In some cases, the key or value isn't important and could be Null.

The Map function takes a $< key,\ value >$ pair as input and emits a list of $< key,\ value >$ pairs:

$$\text{Map}(k_{in}, v_{in}) \rightarrow [< k_{out}^{(1)}, v_{out}^{(1)} >, ..., < k_{out}^{(n)}, v_{out}^{(n)} >]$$

Keys emitted by the Map function do not have to be unique; the same key may be emitted multiple times with the same or different values. The Map function is applied to every item in the input dataset. The Map function only considers the current item being processed and is independent of the other items. This means that it is potentially possible to apply the Map function to all the data items in parallel. The output from all the parallel Maps is sorted and grouped (combined) by key, creating a $< key^{(i)}, [value^{(1)}, ..., v^{(n)}] >$ pair for each unique key.

Each grouped $< key^{(i)}, [value^{(1)}, ..., v^{(n)}] >$ is then processed by a Reduce function. In the original MapReduce paper, the Reduce function returned a list of values:

[9] For more details on the big data GPU state of the art see this Azinta Systems blog post `http://bit.ly/gpuMapReduce`

$$\text{Reduce}(< key^{(i)}, [value^{(1)}, ..., v^{(n)}] >) \rightarrow [value^{(1)}_{out}, ..., v^{(n)}_{out}]$$

Modern interpretations of MapReduce are more flexible and allow the reduce function to emit multiple $< key,\ value >$ pairs (allowing for the same key to be emitted multiple times):

$$\text{Reduce}(< key^{(i)}, [value^{(1)}, ..., v^{(n)}] >) \rightarrow [< key^{(1)}_{out}, value^{(1)}_{out} >, ..., < key^{(1)}_{out}, v^{(n)}_{out} >]$$

For both cases, the number of reduce functions that can be performed in parallel is limited by the number of unique keys output by the map functions.

The MapReduce computational model is coupled with a distributed filesystem and worker processes spread across cluster nodes in order to complete the framework. This arrangement has a number of desirable features:

- **Fault tolerance:** File system blocks are replicated across nodes in the cluster. If the disk of a node fails, then the data is still intact. If a Map or Reduce function suffers a failure it can be re-run on a different machine without having to restart the entire job. The framework is completely resilient to nodes becoming unavailable and re-available, for example due to maintenance.
- **Data locality:** The framework tracks where each block of data is stored and uses this information to intelligently minimise network traffic by performing tasks on nodes where the input data is stored. Modern implementations can be 'rack aware', and will attempt to minimise the traffic between nodes in different racks in preference to minimising between nodes in the same rack.
- **Job scheduling:** Jobs are submitted through a master node which tracks which workers are available and which are busy. The scheduler ensures that the cluster is utilised efficiently, balancing data locality and node utilisation.
- **Scalability:** The framework is massively scalable; processing can potentially be performed on as many machines as there are data records. The practical limitation is the cost of building and running the cluster. The framework allows new nodes to be added to the cluster.

Message Passing Interface versus MapReduce. The Message Passing Interface (MPI) [15] standard is a communication protocol and API specification for parallel computing. It provides the basic tools for parallel processing across machines by facilitating message passing between cluster nodes allowing multiple computer nodes to transmit data and results; coordinate on tasks and interact efficiently. Due to this *all purpose* nature, the technologies implemented on top of MPI are rather varied, ranging from bespoke user applications to the use of MPI as the underlying technology behind many modern computer clusters and supercomputers. Therefore it should be noted that, though we separate our descriptions, it is incorrect to see MPI and MapReduce as two competing technologies addressing the same problem. Rather, MPI is a lower level protocol capable of solving a larger set of problems and MapReduce is a specific methodology designed to deal with a certain subset of parallel problems. Indeed there has even been some recent work on implementing MapReduce over MPI [25]. Several machine learning techniques have been successfully implemented to work across a set of distributed machines using MPI. This includes approaches to training SVMs by chunked training vectors and cascading multiple SVMs [18]; a parallel implementation of the Expectation Maximisation algorithm which was in turn used to implement parallel versions of Gaussian Mixture Models, probabilistic PCA and Sparse Coding [7]; and also, a simulation of 10^9 neurones, exceeding that of a cat cortex [3].

When one examines the current trends in cross machine parallel implementations of data analysis and machine learning algorithms, there are a great many more implementations that use the MapReduce framework as opposed to MPI or in fact any other solution. As we have already discussed, MPI can be used to address the same challenges as MapReduce, and indeed in some situations MPI's lower level nature can allow authors to implement more efficient solutions to given problems. However, as with single machine parallelism, programability and ease of use are overtaking this potential for optimisation. As we discuss below, MapReduce presents the programmer with a restricted, though much simpler programming model. As long as a given problem can be translated into a set of map/reduce steps, various other considerations such as: fault tolerance, data distribution, scheduling and scaling are handled by the MapReduce implementation. With regards to data distribution as an example, in the basic form, data is distributed in MPI through message passing between nodes and can therefore be very inefficient,

especially if data is large and must be transmitted many times throughout the process. The programmer must pay a great deal of attention to achieve efficient data distribution, though if the time and care is taken, the final solution may be more efficient than that achievable using MapReduce. There also exist several well supported implementations of MapReduce which ease its use by novice programmers. These include Apache's Hadoop framework[10] and alternatively the Twister framework[11], both written in Java; the disco framework[12] written in Python and also MongoDB[13], a C++ MapReduce implementation in which map and reduce calls are defined using a javascript like language.

This ease of programability as well as these many implementations usable with many different platforms and codebase languages have resulted in an increasing number of parallel machine learning and computer vision implementations using MapReduce design methodologies [17]. As an overview, consider that of the 12 distributed machine learning algorithms presented at the NIPS workshop for parallel machine learning in the last 4 years, 9 were MapReduce implementations. An example of distributed machine learning over MapReduce is Apache's own Mahout library[14] which provides some implementations of generic machine learning algorithms and techniques over Hadoop. Also, there are many more bespoke applications of MapReduce, some we have already mentioned in the previous sections. An interesting example is the work by Ye et al [49] at Yahoo who have shown an interesting fusion strategy of MPI and Hadoop in creating Gradient Boosted decision trees for large amounts of training data. With regards to computer vision applications, Li et al [29] have shown some interesting results in the application of geolocation applied to a large collection of images using MapReduce. Also, White et al [47] provide some discussions of how certain computer vision and image processing algorithms could potentially be implemented using MapReduce.

3 Practical Hybrid MapReduce with Hadoop

When the MapReduce framework was first described in 2004, it was applied to clusters of commodity dual-processor x86 (2Ghz Xeon with hyperthreading) machines with 4GB of RAM per machine and two 160GB hard drives. Commodity server machines today look a little different. Firstly, processors at an equivalent price-point have gained more cores, and it is now common to have configurations with multiple multi-core processors (for example dual quad core Xeon's with hyperthreading). Secondly, the price of disk based storage has dramatically dropped, and drives of sizes around 2TB are inexpensive. This means that cluster nodes can potentially store much more data. Thirdly, the cost of RAM is still relatively expensive, and so a modern commodity machine at a similar price-point still has around the same ratio of RAM per processor core.

These changes in machine specification can have some implications on how MapReduce is practically used on modern hardware, especially when it comes to memory intensive tasks that might not have originally been possible. In order to utilise modern machines effectively, the MapReduce computational model needs to be enhanced by defining what we term a Hybrid MapReduce model. The following list illustrates some of the different ways in which the MapReduce model can be hybridised for different types of processing requirements:

- **Map without Reduce.** In tasks like feature extraction or image processing an operation is applied to each individual record and the result is stored. The processing does not need to be aware of any of the other input records, or the results of the Map operation.
- **Chained Map and Reduce operations.** Certain kinds of algorithms and processing do not naturally fit into just one single set of Map and Reduce operations. Instead, certain classes of algorithm might be better expressed by chaining sets of Map, Reduce or MapReduce operations.
- **Shared memory across Map operations.** Some kinds of processing require loading a very large supplementary data object into memory which is used as extra information during a Map operation. Loading this large object once for each Map operation is impractical because of the time it would take to load. In addition because a single machine with multiple processors can run multiple Map

[10] `http://hadoop.apache.org/`
[11] `http://www.iterativeMapReduce.org/`
[12] `http://discoproject.org/`
[13] `http://www.mongodb.org/display/DOCS/MapReduce`
[14] `http://mahout.apache.org/`

operations in parallel the total amount of RAM on the machine might not be sufficient for all the parallel Map operations to hold the supplementary at the same time. In order to work around these issues, frameworks like Hadoop allow the number of separate processes to be reduced, and allow for supplementary information to be loaded once by each process before a set of Map operations are carried out by the process.

- **Multithreaded Mappers.** When dealing with large supplementary data as described above, the only option might be to reduce the number of processes loading the data so that memory limits are not exceeded. Unfortunately this would mean that the node was not being used to its full capability as there would be idle CPUs. A solution to this is to allow each process to use multiple threads and process multiple records concurrently within the process.
- **Delayed Emit.** Sometimes operations (Map or Reduce) might need to batch together the output of several calls before returning the output. Often this is done for efficiency as the amount of data being output can be reduced.
- **Out-of-framework data output.** In a pure MapReduce system the only data that can be output comes directly from the outputs of the Map and Reduce operations. In frameworks such as Hadoop, this isn't enforced. The Hadoop framework makes it possible to construct extra data files from inside Mappers and Reducers.

3.1 How Hadoop Works

The Apache Hadoop project provides an open-source implementation of Google's original MapReduce framework. Though implementing many of the key features of MapReduce, Hadoop goes beyond Google's original design providing various hooks at various points throughout the MapReduce framework, and through other framework extensions. These changes allow for all the points listed above, in our description of the Hybrid MapReduce framework, to be implemented effectively.

Hadoop's computational model is extremely flexible. It is easy to create a chain of Mappers and Reducers. In addition, it is possible to have a Mapper with either no reducer (by setting the number of reduce tasks to 0), or a NullReducer. With zero reducer tasks, the output records of the Map operation are written directly to the distributed file system (and are not sorted). With the NullReducer, Map records are sorted by key, but otherwise written to the filesystem unchanged.

In Hadoop, the data to be analysed by a given job is broken down into a set of InputSplits; sets of $< key,\ value >$ pairs comprising a chunk of binary data of a configurable size per job. Practically speaking, an InputSplit can either be backed by a portion of a *splittable* large file, or by a set of smaller files (each representing one $< key,\ value >$ pair). Each split of the data is dealt with by an individual task as started by TaskTrackers on the various cluster machines. Each initialised task is backed by a Context instance which wraps around an InputSplit. As far as is possible, the TaskTracker chosen to process a given InputSplit is close to the machine which physically holds the InputSplit's data. If possible the task is run on the same machine which holds the data to be analysed, if this is not possible a machine is selected which is "close" in terms of rack space or the network hierarchy. It is within these individual tasks that Mapper and Reducer Hadoop classes are instantiated and used to process data. To perform the individual map operations of a task a series of operations take place:

1. A single Mapper instance is instantiated and the setup hook called.
2. Map operations are performed serially through successive calls to the map hook with a $< key,\ value >$ pair from the the InputSplit.
3. The map operations are ended and the cleanup hook is called.

Hadoop is designed such that developers have access hooks which are called during these 3 stages of a task; it is by extending such a Mapper that in the most basic configuration a developer writes their map implementations. In particular, the setup hook provides a point at which data can be loaded and be shared across all subsequent map operations within the task; this is explored in more detail in Section 3.2.

Hadoop also provides the ability to control how individual tasks are run on individual machines given various resource limitations of cluster machines using Schedulers. The CapacityScheduler supports scheduling of tasks on a TaskTracker based on the job's memory requirements in terms of RAM and Virtual Memory. Conceptually, each TaskTracker is composed of a number of map and reduce slots each of which allow a certain amount of memory. A given task can request one or more slots and therefore

Table 1 Time taken to complete analysis of 100,000 images with standard single threaded mappers (i.e. number of threads $T = 1$) with M simultaneous mappers per machine. Standard deviation is shown in brackets.

wait (ms)	$2 \times M, 1 \times T$	$4 \times M, 1 \times T$	$8 \times M, 1 \times T$	$16 \times M, 1 \times T$
1	138.67 (2.52)	83.00 (2.00)	73.67 (14.57)	**53.67** (**4.73**)
10	303.33 (1.15)	168.33 (4.51)	105.33 (3.79)	**72.00** (**2.65**)
100	1819.00 (9.17)	955.67 (0.58)	533.33 (4.93)	**286.00** (**1.00**)
1000	16951.67 (5.77)	8804.00 (1.73)	4806.67 (11.55)	**2479.00** (**5.00**)

the number of simultaneous Mapper and Reducer instances on a given machine in the cluster can be controlled by setting their memory limits appropriately.

3.2 Shared Memory and Mapper Threading

As mentioned at the beginning of this section, and as will be shown practically in Section 4.3, there exist situations where it is inefficient or undesirable to allow map tasks to run entirely in isolation. This can be due to the requirement of some resource which is required by each map operation, but is either expensive to load or expensive to hold in memory. Ideally such a resource would be loaded once for a set of map operations and shared. This scheme breaks the structure of the traditional MapReduce framework within which map and reduce tasks are isolated from one another. Nevertheless, such capabilities are made available in the Hadoop MapReduce implementation in the form of Mapper and Reducer instances and the various hooks made available.

The Hadoop architecture can be leveraged to deal with the requirements of a large, expensive to load, shared resource. Firstly, the memory requirements of Mappers and Reducers can be directly controlled as discussed in Section 3.1. Through this mechanism developers can state that all Mappers require more memory, resulting in a given task occupying more TaskTracker slots and therefore fewer Mapper instances running per machine synchronously each with an ability to hold the large resource required for its internal map operations. To take advantage of this extra memory per Mapper instance, within the setup hook of an individual Mapper the expensive resource may be loaded statically and accessed by each map operation. An issue that remains to be dealt with is that the map operations of a single standard Mapper are run serially. This means that with fewer Mappers, fewer map operations occur at one time and therefore the job as a whole can run more slowly. Another feature provided in Hadoop to address this issue is the MultiThreadedMapper. This mapper instantiates individual map operations with a synchronised access to the underlying InputSplit, but once the data is read it allows for asynchronous running of the map operations themselves via software threads.

3.3 MultiThreadedMappers vs Multiple Standard Mappers

By specifying that fewer Mappers can run at a time it is possible to load large resources into memory and by using MultiThreadedMappers it is possible to tune these fewer Mappers to run multiple map operations simultaneously. On a single machine with P (virtual or real) processors one can intuitively expect the ability to efficiently run P synchronous map operations, either by running $M = P$ synchronous mappers each with 1 thread, or running $M \times T = P$ MultiThreadedMappers M each running T threads. Unfortunately, there is an inherent overhead to using MultiThreadedMappers as input and output operations must be synchronised across all threads as all the threads are reading from the same InputSplit, and writing to the same sink. This means that one task with T threads is unlikely to achieve the throughput of T individual tasks. We experimentally explore the actual effects of running different numbers of mappers with different numbers of threads in the remainder of this section in order to determine a set of optimal configurations.

In the following experiments we created an artificial MapReduce job with no reducers and a Mapper whose sole purpose was to *wait* for some period of time for each map operation, in the experiments below to wait for $1, 10, 100$, 1000, 2000 and 5000 milliseconds. We perform this so called *waiting* MapReduce job on 100,000 images simply to guarantee that the total number of Mappers instantiated and run is much larger than the total number of machines and processors in our setup, given that the total number of Mappers initialised is a function of the total size of the data to be analysed. We ran each job for each

Table 2 Time taken to complete analysis of 100,000 images with MultiThreaded mappers with M simultaneous mappers per machine and T threads per mapper. Standard deviation is shown in brackets.

wait (ms)	$1 \times M, 16 \times T$	$2 \times M, 8 \times T$	$4 \times M, 4 \times T$	$8 \times M, 2 \times T$	$16 \times M, 1 \times T$
1	253.67 (3.21)	102.33 (0.58)	66.00 (3.00)	55.67 (2.08)	**51.67** (**4.04**)
10	257.33 (4.04)	117.33 (2.08)	90.00 (6.24)	75.67 (1.53)	**67.00** (**1.73**)
100	527.33 (2.89)	305.00 (4.36)	**281.67** (**3.79**)	287.00 (2.65)	282.33 (2.52)
1000	3360.00 (8.19)	**2196.33** (**6.43**)	2254.33 (6.51)	2422.33 (3.51)	2476.67 (4.73)
2000	4414.00 (2.83)	**4413.67** (**2.05**)	4455.67 (0.94)	4779.33 (9.29)	4917.67 (2.87)
5000	**10793.33** (**1.25**)	10883.00 (8.29)	11058.00 (10.61)	11862.67 (6.85)	12250.00 (20.61)

Table 3 Time taken as per Table 2, but with purposefully sub optimal usage of each machine (i.e. more free resources per machine). Standard deviation is shown in brackets.

wait (ms)	$1 \times M, 8 \times T$	$2 \times M, 4 \times T$	$4 \times M, 2 \times T$	$8 \times M, 1 \times T$
1	179.00 (0.00)	109.33 (1.15)	74.00 (4.58)	**63.00** (**2.65**)
10	204.00 (0.00)	145.67 (3.79)	114.67 (2.08)	**106.33** (**1.53**)
100	580.67 (3.06)	523.67 (5.03)	**507.67** (**5.69**)	532.33 (2.31)
1000	4375.00 (7.21)	**4316.33** (**6.66**)	4441.67 (11.59)	4784.33 (17.04)
2000	**8631.33** (**1.70**)	8720.67 (17.15)	8854.33 (7.13)	9510.33 (2.87)
5000	**21319.00** (**2.45**)	21613.00 (8.83)	22018.67 (5.91)	23693.67 (60.58)

configuration 3 times and measured the times taken. In the results tables the averages of these 3 runs are shown together with the standard deviations in brackets.

Firstly, we performed this job running (2, 4, 8 and 16) standard mappers with 1 thread per mapper running synchronously on each machine. These times are shown in Table 1. In this setup each machine was in essence running the same number of synchronous map operations per machine as there were Mappers per machine. Next, we attempted to maintain a constant number of 16 map operations running at any one point on a given machine, but achieving this with mixtures of more mappers and fewer threads or fewer mappers and more threads. Specifically we ran (1, 2, 4, 8 and 16) MultiThreadMappers each running (16, 8, 4, 2 and 1) threads respectively (Table 2). Finally, as a point of comparison, we purposefully under-utilised each machine, enforcing a maximum of 8 simultaneous map operations and investigating job times for mixtures of more mappers and more threads (Table 3).

The first thing to notice from these experiments is that running fewer Mappers with 1 thread each runs slower regardless of wait time. This is an expected result as in the 8, 4, 2 and 1 Mapper cases the machines in our cluster are being heavily under-utilised in terms of processing capability and I/O. However, as discussed above, sometimes fewer Mapper instances must run simultaneously in order to allow for more memory per Mapper. We can alleviate the issues of under-utilisation by running multiple threads per mapper.

Overall the results of running multiple threads per Mapper show that there is an improvement over running 8 mappers or 16 mappers with 1 thread. The results demonstrate that in general it is not a clear cut case of "run as few" or "run as many" Mappers as possible. Indeed, we show that, with a very small standard deviation between times, the correct choice to make with regard to mappers vs threads is dependant on map operation time. We note that when map operations are very fast (e.g. in the 1 millisecond wait case) a developer should endeavour to initiate as many mappers per machine as memory limitations will permit. On the other hand, when map operation times are longer, the best configuration favours fewer map tasks with more threads per mapper. Though surprising, this is a result echoed in the suboptimal configuration of 8 simultaneous map operations per machine shown in Table 3, fewer mappers and more threads is best when map operations take a long time while many mappers with fewer threads is better when map operations are short.

To put these wait times into context, consider that with the dataset described in Section 6, feature extraction takes 2.5 seconds on average with a standard deviation of 3.6 seconds and feature quantisation takes 0.85 seconds with a standard deviation of 1.4 seconds. The relatively large standard deviations are due to certain extremely large images in the dataset which take 30 to 60 seconds in feature extraction and 10 to 15 seconds in feature quantisation. The wait simulations we present in this section suggest that this disparity in process time necessitates considerably different strategies in terms of Mapper/Thread mix. By taking account of these considerations we can reduce the running time of our implementation of the BoVW indexing pipeline described in the remainder of the paper.

4 Efficient Bag of Visual Words using Hybrid MapReduce

One goal of this paper is to present a methodology for the implementation of a large scale image analysis and indexing pipeline using the hybrid MapReduce framework described in the previous section. In doing so we outline techniques which facilitate content based search across web-scale image corpora.

The Bag of Visual Words (BoVW) pipeline [38, 44] for image retrieval has been shown to be an effective, powerful and scalable method for performing content based image retrieval. In the remainder of this section we outline: what stages are involved in the BoVW pipeline; how each stage has been addressed in the past and the details of implementing a specific incarnation of this pipeline in a MapReduce framework.

Speaking very generally, the BoVW pipeline involves the extraction of sets of *visual words* drawn from a larger visual *vocabulary* representing each image in a corpus. By representing images as sets of words, it is possible to take advantage of highly optimised text-retrieval structures such as the inverted-index to achieve highly efficient retrieval of images. A BoVW indexing pipeline can be broken down into 4 main steps: (1) The features of each image one wishes to retrieve are extracted. Once extracted; (2) a visual feature vocabulary is constructed and used to (3) quantise each image's features as discrete visual words. (4) These visual words are then used to construct an inverted-index. Once constructed, a query image can be used to search this index efficiently. The query image's features must be extracted and quantised using the same vocabulary as that used to construct the index images. These quantised query features can then be used to locate *similar* documents inside the inverted index.

The individual steps of this pipeline have received a great deal of attention. One can extract and describe features from images using a variety of techniques, the approaches available for codebook generation and optimisation have been explored in great depth as has the construction of the inverted index (though this has mainly been addressed for text retrieval). Across all these techniques it has been consistently demonstrated that BoVW-based approaches have comparable or better performance and results when compared to other approaches in image retrieval.

In the following sections we explore the details of the 4 major steps of the BoVW indexing pipeline. We explore the background of each stage followed by details of the stage's hybrid MapReduce implementation. Each stage of the pipeline has presented its own challenges when adapted to work on a MapReduce framework. Various considerations had to be made and solutions engineered to deal with various issues. We believe that the techniques we have developed to implement the specific algorithms selected on a MapReduce framework can be extended to help the implementation of alternative algorithms for each stage of the pipeline. When addressing each stage of the algorithm, we explore the benefits gained when doing parallelising work using MapReduce as compared to the same tasks running on a single machine.

4.1 Feature Extraction

The first stage in the indexing pipeline is the extraction of meaningful features from images in a corpus. This stage has an immediate analogy with the text retrieval pre-processing stage known as "tokenization" with the extracted image features being analogues to words (or sub-parts of words such as letters or phonemes). The extraction of local features from images is a popular technique used in this first stage and has received a great deal of attention. Many kinds of local image features have been extracted for description with varying levels of success [35, 34]. A popular [38] approach to local feature extraction and description is the Scale Invariant Feature Transform (SIFT) [30]. This feature holds a 128 dimensional quantised edge histogram relative to a dominant orientation of a given region, providing for some invariance to orientation. For purposes of uniqueness and stability, these regions are selected at salient points within images. Modern salient point detectors can be scale- [30, 28] or affine- invariant [35]. Some of the most popular detectors currently cited in the literature include: the difference-of-Gaussian (DoG) detector [30], the Maximally Stable Extremal Region (MSER) detector [32], and the Harris- and Hessian-affine detectors [35].

This step of the pipeline is an example of an algorithm which is well suited to MapReduce. Breaking down the task of feature extraction from images in a corpus into multiple atomic tasks is most easily achieved by separating the task at the image level. In this scheme there is no shared state between the individual tasks of the process, i.e. the extraction of features from one image has no bearing on the extraction from another and can therefore be formed easily in parallel. Therefore, the input to an

individual map task is an individual image and the output is the local features extracted from that image. In the notation adopted in Section 2.2, this leads to the following Map and Reduce functions:

$$\text{Map}(fl^{(n)}, I^{(n)}) \rightarrow [< fl^{(n)}, \mathbf{ft}^{1\ldots F(I^n)} >]$$
$$\text{Reducer}_{\text{null}}(fl^{(n)}, \mathbf{ft}^{1\ldots F(I^n)}) \rightarrow [< fl^{(n)}, \mathbf{ft}^{1\ldots F(I^n)} >]$$

The keys provided to the map tasks are unique image identifiers, specifically the image's filename or unique identifier, $fl^{(n)}$, while the values are the image files themselves $[I^{(0)} \ldots I^{(N)}]$. Each map task is handed one $< fl^{(n)}, I^{(n)} >$ pair, where $I^{(n)}$ is the n^{th} image of N images in the corpus. Defining $F()$ as a function which returns the number of features in a given image $I^{(n)}$, all the $F(I^n)$ local features $\mathbf{ft}^{1\ldots F(I^n)}$ of the image $I^{(n)}$ are emitted with the filename $fl^{(n)}$ as the key by a single map task.

Diverging from the traditional MapReduce paradigm, we note that the output of the individual mappers are already in their final form. That is to say, if a reducer were used, it would be the *null reducer* which outputs the same $< key, value >$ pair as it was given as input, performing no further aggregation or processing. Therefore, in our implementation we explicitly forgo the reduce step for the feature extraction portion of the pipeline entirely, instead using the output of the map tasks as inputs for the next pipeline stage. By doing so we avoid the wasteful null reducer step, amounting to a sorting and copying of map outputs to a different location with identical content in the Hadoop distributed file system.

4.2 Vocabulary construction via K-means Clustering

An inverted-index, the efficient retrieval structure we are ultimately creating, works most efficiently when documents are represented by discrete words which are a subset of a much larger vocabulary, i.e. documents are represented by a sparse vector of discrete terms. In their current form, local feature descriptions are both dense and continuous as well as relatively expensive to compare. Therefore the next stage of the indexing pipeline is the process by which the complex local features extracted from an image can be meaningfully represented as discrete terms. A popular approach is vector quantisation wherein the set of local features extracted from an image are quantised against a vocabulary of discrete visual terms. The use of vector quantisation itself has an immediate analogy with a pre-processing stage known as "stemming" in the indexing of textual documents.

The process of constructing a visual vocabulary has enjoyed much attention over the last decade. Sivic and Zisserman [44] selected a set of video frames from which to train their vector-quantiser, and used the K-means clustering algorithm [31] to find clusters of local descriptors within the training set of frames. The centroids of these clusters then became the *visual words* representing the entire possible vocabulary. The biggest problem of the K-means based approach is that it is computationally very expensive to create large vocabularies in high (i.e. 128) dimensional spaces. More recently, Nistér and Stewénius [37] proposed the use of hierarchical K-means to enable them to build visual vocabularies with over 1 million SIFT-based terms. This approach was shown to be faster than a standard K-means. However, there was a significant drop in precision when hierarchical K-means vocabularies were used as compared to standard K-means vocabularies of equivalent sizes. A more recent alternative suggested by Philbin et al [40] uses an efficient KD-tree implementation to approximately allocate samples to centroids. Using this approximate approach it is possible to speed up both the training and quantisation phases of the classic K-means algorithm allowing for much larger vocabularies to be constructed while maintaining the power of standard K-means vocabularies.

In our MapReduce implementation of this pipeline stage, an individual mapper is handed the SIFT features belonging to a single image. The goal of this stage is the output of K centroids which can be used as the representative features of a vocabulary of K terms. We present a MapReduce implementation of the approximate K-means algorithm. There are two main stages to this algorithm:

1. **Selecting starting centroids**. There are various approaches to efficient initial cluster centroid selection [4], however, in our case we consider selecting initial centroids by sampling randomly from features in the image corpus.
2. **Iterative centroid recalculation**. Sample points must be assigned to the clusters and the centroids of the clusters are then adjusted. This is repeated for a given number of iterations, resulting ultimately in a better set of cluster centroids.

Random Centre Selection. The goal of this portion of the algorithm is the random selection of K centroids from the features contained in a corpus of images. This stage must guarantee that no particular image I^n or type of feature within a given image $\mathbf{ft}^{1...F(I^n)}$ is preferred over any other. An approach for efficiently selecting random features from a set of features is to know the total number of features in the set, select K random integers between 0 and the total number of features (i.e. $\sum_{n=1}^{N} F(I^n)$ for N images) and selectively extract the features as the initial centroids. Assuming a fair random number generator, this approach guarantees effective extraction with no preference to a given image or type of feature. However, in a MapReduce framework this approach is inefficient due to the lack of prior knowledge of the total number of features. Not only must an initial MapReduce task be used to count the features of the images, but it is not known at the map task level where in a sequence a given $< key,\ value >$ pair being analysed is and therefore whether the features inside the image being analysed are to be selected.

Instead we present a technique for selecting a subset of elements from a larger set randomly using MapReduce. Our approach to this problem uses the map task to randomise the order of the features for a given image $I^{(n)}$ and emit each of the $F(I^n)$ features $\mathbf{ft}^{1...F(I^n)}$ with a random key $rnd^{(r)}$ such that there exist R random keys in total. With each feature assigned a random number, the reduce phase can select the first K features seen across the R random keys and therefore efficiently select K random centroids.

Though tractable, this solution results in every feature from every image being emitted, dealt with and potentially skipped in the reduce step. Assuming many more total local features across all images ($\sum_n^N F(I^n)$) than required centroids K, this results in many more features emitted than are required. In turn this results in higher network load, hard disk activity and therefore a drop in overall performance.

By utilising the design of the Hadoop architecture we engineer a solution to this issue using the "Delayed Emit" hybridisation mentioned in Section 3. In Hadoop each map task is run in isolation within the context of an overall Mapper instance. A Mapper is instantiated once per InputSplit on each machine, at which point the Hadoop programmer is given access to the setup hook, called once per Mapper instance. The Mapper's map function is now called for every $< key,\ value >$ within the InputSplit and these calls constitute the actual map tasks. Finally, once a Mapper instance exhausts the underlying InputSplit, a cleanup function is called. In our approach, instead of emitting key value pairs in the map tasks, we use the map tasks to populate a priority queue with the features $ft^{(n)}$ of each image I^n. These features are prioritised by a random number $rnd^{(r)}$ assigned to them at the map stage. Once each map task of a given mapper is called, we use the cleanup function to actually emit features with random numbers, choosing to only emit the top K features randomly ordered by their assigned $rnd^{(r)}$ key. This scheme results in a fair random selection of features whilst not emitting every feature of every image.

At the reduce stage we again take advantage of Hadoop's architecture to efficiently select the final K random centroids. As with the map tasks, many reduce tasks are run in the context of a single Reducer instance, each with their own setup, reduce and cleanup functions. We purposefully allow only one Reducer instance to run. We keep a count of the number of features emitted, E, across all reduce tasks run in the single reducer instance, with the initial value of $E := 0$. Defining $F()$ as a function that counts the number of features assigned to a random number r, then each reduce task is given as its key a random number, $rnd^{(r)}$, and a set of $F(r)$ features, $\mathbf{ft}^{1...F(r)}$, as its value. The features in the value correspond to those that were emitted with that particular random number in the various map tasks. In an individual reduce task, each feature is emitted with E as its key becoming the k^{th} centroid at which point $E := E + 1$. Once $E = K$, no further features are emitted by the reduce function.

Formally, our Map and Reduce tasks can be expressed as follows, where N is the number of images processed by a single mapper task, [Map(•)], and R is the total number of random keys emitted by the mappers and processed by the reducer, [Reduce(•)]:

$$[\text{Map}(fl^{(n)}, \mathbf{ft}^{1...F(I^n)})]_{n\in 0...N} \rightarrow [< rnd^{(1)}, ft^{(1)} >, \ldots, < rnd^{(L)}, ft^{(L)} >] \mid L = \min(K, \sum_{n=0}^{N} F(I^n))$$

$$\left[\text{Reduce}(rnd^{(r)}, \mathbf{ft}^{1...F(r)})\right]_{r\in 0...R} \rightarrow [< 1, ft^{(1)} >, \ldots, < K, ft^{(K)} >]$$

Our methodology results in each feature being loaded once within the various map tasks while only a maximum of $K \times M$ features (where M is the number of mappers launched by Hadoop which is a function of the number of mapping slots, the size of file splits and the amount of data to be processed in the given job) are emitted in total at the map stage and from these features only K centroids are selected at the reduce stage. This MapReduce approach for feature selection, though emitting more features than are strictly necessary at the map stage, scales well. The actual reading of the features of individual images is performed in parallel across multiple machines and features themselves are emitted sparingly. We can now select initial centroids from much larger sets of features than was possible with the single machine implementations. This in turn means that the random initial centroid selection is more likely to accurately reflect the actual feature distribution of the document corpus.

It should be noted that the principles behind this random centroid selection scheme can be adapted to select a subset of the features of the images in a given corpus. This allows the iterative K-means cluster assignment step (discussed below) to be applied to a smaller random subsets of features, maintaining the process' accuracy whilst reducing the total run time.

Iterative Cluster Assignment. In this portion of the algorithm the K centroids of the K-means algorithm are estimated. We describe the process using the approximate K-means algorithm, which uses an ensemble of KD-trees to hold the current centroids for efficiency, but the standard K-means is a trivial modification of this.

Starting with the random centroids selected from the previous step and using an iterative process, sample features are assigned to their closest centroids at which point the centroids are re-evaluated as an average of samples assigned to them. In MapReduce we formulate this algorithm as a set of successive MapReduce jobs. In each job, each map task is handed a single image's $F(I^n)$ features $ft^{1\ldots F(I^n)}$. Whilst holding a copy of the current K centroids $[< 1, ft^1 >, ..., < K, ft^K >]$ in memory in approximate KD-trees, the map task emits the index of the closest centroid (using Euclidean distance) for each feature using the centroid index k as the key and the feature $ft^{(m)}$ as the value. The reduce task then iterates through each of the $F(k)$ features $ft^{1\ldots F(k)}$ assigned to the centroid k and emits a new centroid for each k which is the average feature ft^{avg} of all the features assigned to k. This process is efficient as both the map and reduce portions can be run on multiple machines and consolidated. After each job is run, a new set of the centroids is available and used for the next iteration of the process. This is repeated for a pre-set number of iterations (experimentally, we have found 30 iterations to work well). A single iteration can be formally expressed as:

$$Map(fl^{(n)}, \mathbf{ft}^{1\ldots F(I^n)}) \rightarrow [< k^{(1)}, ft^{(1)} >, \ldots, < k^{(F(I^n))}, ft^{F(I^n)} >],$$
$$Reduce(k, \mathbf{ft}^{1\ldots F(k)}) \rightarrow [< k, ft^{avg} >]$$

A minor nuance of this algorithm is that in a given iteration, a given centroid index k might be assigned no features at all. Therefore, using a process similar to that described in Section 4.2, L features $\mathbf{ft}^{1,\ldots,L}$ are selected at the end of each iteration where L is the number of centroid indexes not assigned any features. These randomly selected features are used in the place of the missing centroids in proceeding iterations.

Another interesting nuance of our implementation of the approximate K-means algorithm is that each map task must have available a large and identical copy of the KD-tree representing the current centroids. This is the data structure which is used to quickly identify closest centroids and is vital to the efficiency of this algorithm. Not only is the loading of the current centroids and construction of the KD-Tree structure time consuming, but the resulting data structure can be very memory intensive (for example, $8 * 128 * 1,000,000 = 1GB$ for 1 million centroids stored as 64-bit doubles without including the tree structure). The time taken to construct the KD-tree means it would be preferable to load the KD-tree once for all map tasks which run on the same machine, or at least those map tasks running in the same Mapper instance. This configuration can be achieved by loading the KD-tree on the initialisation of the Mapper instance (i.e. when the setup function is called), making the centroids available to a group of map tasks to be run under that mapper.

The memory intensive nature of the centroids is a different problem that culminates in a limiting of the number of Mapper instances which can simultaneously run on a single machine. Furthermore in a basic Hadoop configuration each running Mapper instance can run only one map task at a time. This

means that the limitation of Mapper instances could result in an under utilisation of processing and I/O capability available on a given Hadoop machine. As previously discussed, a solution to this problem is to use software threads instead of separate processes to utilise processing capability and I/O. In Hadoop this can be achieved using a special kind of Mapper called a MultithreadedMapper. Unlike standard Mapper instances which are separate operating system processes (separate JVMs in the case of Hadoop which runs on Java), the threads in a MultithreadedMapper have access to shared memory which can hold large shared data structures (e.g. the centroids KD-tree data structure) while maintaining the ability to run an expensive algorithm synchronously (e.g. centroid assignment). This means fewer copies of the expensive centroids object can be loaded in memory while still taking full advantage of available processing and I/O capability on each machine.

Another notable implementation of the K-Means algorithm as an iterative MapReduce process is the one offered by Mahout[15]. Their implementation is more flexible than ours, offering a large set of configurable options including: different distance metrics; initialisation strategies (including canopy clustering) and various convergence criteria which we hope to implement in future releases. Key innovations offered by our implementation over the Mahout implementation are the MapReduce random centroid selection and the allowance for approximate clustering. In terms of cluster assignment over the various iterations of the K-means algorithm, the Mahout implementation only supports cluster assignment through checking of every cluster separately. This works well enough when k is small, however it is well known [37, 40] that in BoVW, large vocabularies (i.e. one million of visual terms or more) achieve higher accuracy. Such high numbers of clusters would not be handled efficiently by the brute force approaches. This was a key motivation of our use of approximate centroid assignment. This inability to deal with large numbers of centroids is further reflected in Mahout's strategy for cluster loading. Where we use the MultithreadedMapper to allow for a single cluster instance to be loaded for many map tasks, Mahout requires that all prior clusters are loaded once per map task. This approach is acceptable when cluster numbers are low, but has a significant cost (both in loading times and system memory) when cluster numbers are high. With regards to the random initialisation strategy, Mahout selects features as a pre-processing step on a single machine. All the feature vectors in the training corpus are iterated through and random numbers drawn. A given vector is selected as a centroid if this random number is below a certain threshold. The problems with this approach are twofold: Firstly if the feature vectors are not randomised before this process, then a poorly chosen threshold will not allow for a complete pass over the data. This means initial centroids will be more likely selected from feature vectors earlier in the dataset rather than those towards the end of the dataset. In the case of images, our features are organised in distinct chunks (sets of features per image) meaning initial centroids could potentially be selected from only the first few images. In certain datasets this might even mean features are only selected from images of a single scene. If a better threshold is selected or the features are randomised as another preprocessing step, the single machine non-MapReduce approach taken means that the entire corpus of features must be potentially transmitted to a single machine so as to allow for their selection as initial centroids. For very large feature sets this is clearly inappropriate; an issue which motivated our random selection MapReduce implementation.

4.3 Feature Quantisation

At this stage we have constructed a vocabulary of feature centroids which gives us the ability to address the next stage of the pipeline, namely the transformation of image features from a continuous space of high dimensional features, ft, to a set of discrete, *visual terms*, fq.

Both practically and conceptually this stage is extremely similar to a single iteration of the AK-means algorithm described in Section 4.2. The input to this stage is the features $\mathbf{ft}^{1\ldots F(I^n)}$ extracted from each of the N images $I^{(n)}$ and a vocabulary of terms represented by the K centroids $[<1, ft^1>, ..., <K, ft^K>]$ of the previous stage. Using the same MultiThreadedMapper setup discussed in the previous section, the centroids are made available in memory for each map task as an ensemble of KD-trees which are used to efficiently identify the closest centroid $fq^{(m)}$ for each feature $ft^{(m)}$. It is at this stage where this algorithm differs from that of Section 4.2. For each of the $F(I^{(n)})$ features a closest centroid $ft^{(k)}$ is

[15] `https://cwiki.apache.org/MAHOUT/k-means-clustering.html`

identified such that for each image $I^{(n)}$ there exists a set of quantised centroid assignments $\mathbf{fq}^{1\ldots F(I^n)}$ which are the terms which define the image $I^{(n)}$ and are the output of each map task:

$$Map(fl^{(n)}, \mathbf{ft}^{1\ldots F(I^n)}) \rightarrow [< fl^{(n)}, \mathbf{fq}^{1\ldots F(I^n)} >]$$

As with the initial feature extraction in Section 4.1, the data emitted by the map task is the final format of this stage which means that the reduce step is the null reducer and is therefore skipped for efficiency reasons.

4.4 Index Construction

In order to perform efficient retrieval, visual terms must be indexed to allow for fast comparison of a query set of terms against the visual terms from the target image corpus. Fundamentally, visual terms are like terms in a text document and it is common to follow ideas developed for large-scale text indexing when indexing visual terms. In particular, a typical index consists of the three following components:

1. **The Document Index** The document index is a record of the images in a collection. Typically, each record in the index stores the numeric document identifier, the number of visual terms in the document and some additional metadata (such as the location or filename of the image). The document index is typically loaded into the memory of the machine that hosts the search engine. It is common to keep the records ordered by their identifier, and numbered sequentially so that records can be looked up by direct addressing.
2. **The Inverted Index** The inverted index is like an ideal book index. For every visual term in the corpus it stores the references to the images that contain the visual term and the number of occurrences in the respective images (note that non-occurrences are not stored). Pairs of (documentId, frequency) are called postings. The postings for all images containing a particular term is called a postings list. The inverted index is formed by appending the postings lists for all terms in the same order as the terms appear in the lexicon (see below). The inverted index is typically very large and is stored as a file that resides on a disk[16]. Postings lists are read directly from the inverted file on disk as required by a given query. Some inverted indexes are known as *augmented* indexes. Augmented indexes store additional information with each posting. A common use of an augmented index is to store the position of visual terms in the image; this allows search results to be improved by applying geometric consistency constraints at query time. A number of such geometric constraints have been suggested in the literature [44, 27, 39], and each of these may require slightly different information to be stored in the index.
3. **The Lexicon** The lexicon is an index of the visual terms in the corpus. Each record of the lexicon contains the visual term, the frequency of the visual term in the entire corpus, and an offset that determines where to start reading the postings list for the respective visual term from in the inverted index (sometimes the end offset is used instead; sometimes both start and end are included). The lexicon, like the document index is also loaded into main memory for efficient access. The records are indexed by their terms through a hash-table or b-tree structure to ensure that records for a given term can be looked up very quickly. The lexicon might also contain a numeric identifier for each visual term.

Because the inverted index must be streamed from disk at query time (as it is likely to be too big to fit in RAM), it is usually heavily compressed to minimise the amount of data that needs to be transferred. Even if the index can fit in RAM, compression is still used to minimise its size. A number of techniques are used to compress the index, including integer coding schemes such as unary, gamma and variable-integer coding (which reduce the number of bits required) and encoding of differences rather than absolute values (reducing the magnitude of the values being stored).

For small corpora, the inverted index can be constructed directly in RAM, however for even modest corpora sizes this becomes impractical. The classical technique for constructing an inverted index uses

[16] Google has recently moved to a distributed in-memory inverted index for its web search engine, however this requires massive amounts of hardware [see 11, and the accompanying presentation at `http://videolectures.net/wsdm09_dean_cblirs/`].

two passes through the data. In the first-pass, a structure called a direct index is created. The direct index stores lists of (termId, termFrequency) for each document. In the second pass, for each and every term the direct index is scanned for occurrences of the term and the postings list is constructed and written to disk. If the document corpus is large, then the direct index (and inverted index) will not fit in memory, and will have to be written to disk. Unfortunately this means that the second pass will be very I/O intensive and thus rather slow.

An alternative to two-pass indexing called the single-pass technique [24] solves some of the problems of two-pass inverted index construction. The single-pass technique works by processing documents and directly building the postings lists in memory. Once the available memory is exhausted, the postings lists are flushed to disk as a 'run'. The 'run' is essentially a sub-index over a portion of the documents in the corpus. After the postings lists have been flushed and their memory has been released the indexer continues working through the corpus until the memory is exhausted again at which point another run is created. The process continues until all the documents have been processed. Finally, all the runs are merged on disk into the final inverted index. Single-pass indexing is considerably quicker than the two-pass technique [24].

The original MapReduce paper claimed that MapReduce could trivially be used for the construction of simple inverted indexes, however it failed to give any concrete details as to how this might be achieved; in particular, the paper did not discuss how the term-frequencies within each document would be stored. The recent paper by McCreadie et al [33] evaluates and compares four different strategies for MapReduce indexing:

1. **Per-token indexing.** The map function outputs $< term,\ docId >$ pairs for every term and document in the corpus. If a term occurs tf times in a document, then $tf < term,\ docId >$ pairs will be emitted. The reducer aggregates the pairs for each term and document identifier in order to calculate frequencies and thus generate the postings list.
2. **Per-term indexing.** The per-token approach emits a lot of data from the mapper. This can be trivially reduced by calculating the per-document term frequencies and emitting $< term,\ (docId,\ tf) >$ pairs instead. The reducer then only needs to sort the list of (docId, tf) by document to form the postings list for each term. It should be noted that the equivalent of per-term indexing can be achieved using the per-token approach with a *combiner* function that performs a localised merge on the output of each map task.
3. **Per-document indexing.** Instead of making multiple emits per document, another approach is to construct a list of $[(term,\ tf)]$ tuples and emit them all at once for each document in the form of a $< docId,\ [(term,\ tf)] >$ pair. The reduce phase then builds the inverted index from this data. This approach is analogous to the standard two-pass indexing technique; the first pass is the map-function, which results in a direct index, and then the reduce function performs the second pass, inverting the direct index.
4. **Per-posting list indexing.** The final strategy incorporates the single-pass indexing technique. Each mapper processes a set of *documents* (i.e. a file containing the extracted visual terms for a given image) and builds a set of in-memory postings lists. Once memory is exhausted, the postings lists for each indexed term is emitted by the mapper as $< term,\ [(docId,\ tf,\ metadata)] >$ pairs (the metadata is the data for augmentation of the index). The reducer then performs the merging-task in which the postings lists for each flush are combined.

The evaluation by McCreadie et al [33] demonstrates that the fourth, per-posting list, option is the most optimal strategy, minimising both the amount of data emitted by the map function and the overall indexing time. In the brief description of the per-posting list strategy above, a number of important details were omitted for clarity. Full details are given in the following section.

*Per-posting list indexing with hybrid MapReduce.*The per-posting list indexing approach actually diverges from a pure MapReduce implementation. In particular in the map function implementation, a number of pieces of extra information need to be recorded and passed to the reducer outside of the normal emit method. Specifically, the following items are written to disk (actually to the distributed file system) by each mapper:

– Housekeeping information, including details of the InputSplit number, the number of flushes (and how many documents were indexed per flush) and the total number of documents indexed by the mapper.

– A document index and lexicon for the documents and terms encountered during the life of the mapper.

As mentioned previously, during a flush, the postings lists are emitted by the mapper function as $< term,\ [(docId,\ tf,\ metadata)] >$ pairs. Both the key (term) and value (postings list) are actually augmented with additional metadata when they are emitted, and the postings list data is highly compressed. The metadata consists of information about the mapper, the flush number and the InputSplit number.

By sorting the data emitted by the map functions it is possible to create two different forms of index. If the map output is sorted by split number, it is possible to build a "sharded" index, which is basically a set of smaller indexes of non-overlapping documents (useful for distributed querying scenarios). Each reducer function will receive a set of postings corresponding to non-overlapping documents. The number of shards is controlled by the number of reducers.

Alternatively, the map outputs can be sorted by term, and each reducer will receive the postings corresponding to a unique set of terms. The reducers then build inverted files for the given terms. A final post-reduce process links the inverted files and lexicons into a single index (note that the inverted files don't need to be actually merged into a single file if the lexicon contains information on which inverted file a term resides in, in addition to the offset of the postings list).

5 The OpenIMAJ and ImageTerrier Hybrid MapReduce implementation of an image analysis and indexing pipeline

As part of the OpenIMAJ[17] [23] and ImageTerrier[18] [23, 22] open-source software projects, we have created a set of tools that implement the methodology described in Section 4. In addition we have also implemented some supporting tools for handling Hadoop SequenceFiles. SequenceFiles are a form of archive file containing many smaller files. SequenceFiles are important in the Hadoop, because it is inefficient to store many small files on the Hadoop distributed file system. In terms of InputSplits, SequenceFiles are splittable, and will only split between records, which makes them ideal for storing multiple images or sets of features.

The complete set of Hadoop-based tools and their functionality are described in the following list:

– **SequenceFileTool**. Allows the easy creation, examination and extraction of Hadoop SequenceFiles. The tool may be used to construct a SequenceFile containing a large number of images as a precursor step to extracting image features, or to extract the features obtained from a MapReduce based feature extraction task.
– **HadoopLocalFeaturesTool**. Map-only distributed local feature extraction from large volumes of images as described in Section 4.1. Currently implements difference-of-Gaussian SIFT [30], Min-Max difference-of-Gaussian SIFT [21] and ASIFT [36].
– **HadoopFastKMeans**. Iterative MapReduce implementation of the exact and approximate K-Means algorithms as described in Section 4.2. The tool has the (optional) ability to perform an initial subsampling of data, and there is also the ability to control how many threads are used in order to optimise throughput performance.
– **HadoopClusterQuantiserTool**. Multithreaded MapReduce implementation of vector quantisation as described in Section 4.3. Given an existing quantiser definition (created by the HadoopFastKMeans tool), compute nodes read the quantiser into RAM and quantise data points in parallel.
– **HadoopImageTerrier**. Implementation of the hybrid MapReduce single-pass indexing scheme described in Section 4.4. Our implementation differs from the Terrier[19] implementation used by McCreadie et al [33] in that it allows many different forms of augmented index (with geometric information) to be created. The Terrier Hadoop implementation only allows for augmenting indexes with block offsets calculated from the offsets of tokens in a text document.
– **HadoopGlobalFeaturesTool**. Map-only distributed global feature extraction from large volumes of images (including many standard features such as colour histograms, etc).
– **HadoopImageDownload**. Given a file containing a large number of URLs, download the images in parallel. Useful for downloading all the images from ImageNet [13], for example.

[17] `http://www.openimaj.org`
[18] `www.imageterrier.org`
[19] `http://www.terrier.org`

Table 4 Comparison of time taken to complete various pipeline stages with 10^6 inputs on a single machine vs. a 3 machine hadoop cluster.

	Single Machine	3 Machine Cluster	Relative Speedup
Feature Extraction	68992	20962	3.29
K-means Clustering	41565	11697	3.6
Feature Quantisation	98589	29049	3.4
Index Construction	34669	4648	7.5

– **SequenceFileIndexer**. Construct indexes of key/offset for the entries within a SequenceFile.

6 Experiments using OpenIMAJ and ImageTerrier

In order to demonstrate our indexing pipeline, we have performed a range of experiments. Firstly, we have investigated the scalability of the pipeline with varying image dataset sizes and compute configurations. Secondly, we present the results of an actual retrieval experiment using the pipeline.

6.1 Scalability

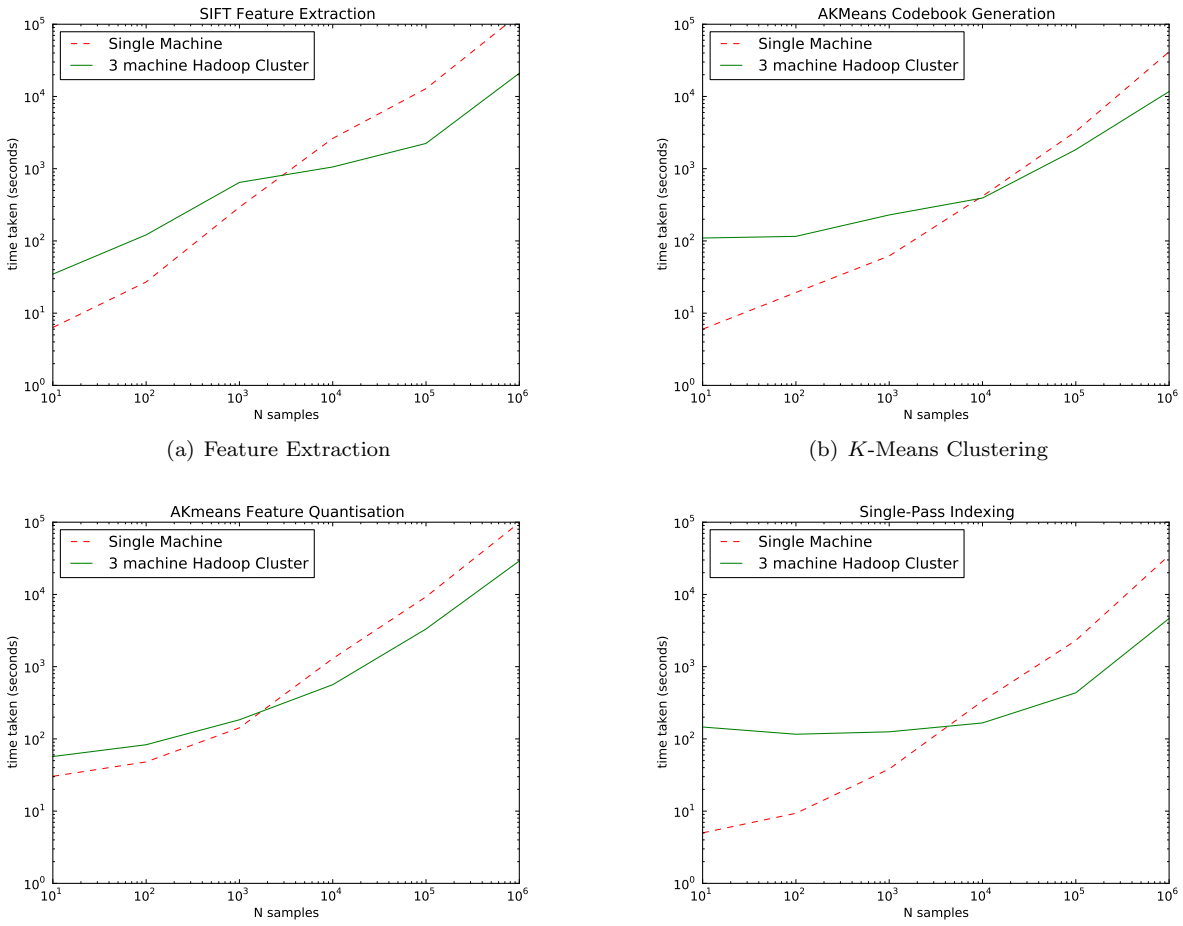
In order to assess the scalability of the approaches described in this paper we have performed a number of experiments with varying image dataset sizes, and performed using two compute configurations: a single machine with 16 processors, and a small cluster of 3 machines with a total of 48 processors (16 each). The machines are all the same specification, and have 12GB of RAM and 8TB of disk storage (over four disks). The cluster is networked through a single Gigabit switch with the machines connected by dual bonded Gigabit ethernet. Other than the Linux operating system, the machines were not running any additional tasks during the experiments.

For the test dataset, we randomly sampled 6 datasets of sizes increasing in powers of 10 from 10 to 1,000,000 images from the ImageNet[20] [13] collection. Using each dataset, we timed how long each of the four pipeline stages (feature extraction, clustering, quantisation and indexing) would take running on a single machine (using optimised multithreaded implementations with 16 threads) compared to running on the mini-cluster with 48 threads. Hadoop jobs were optimised as described in Section 3.3 to balance memory usage against number of processes and threads. Graphs showing the recorded timing information are shown in Figure 1. Note that in order to use the number of images as the dependent variable for all tests, the clustering and quantisation jobs are setup differently; specifically, for the clustering experiment, we set the number of clusters to 5000 (which in reality is far too small). The quantisation jobs were run using a fixed vocabulary of 1,000,000 visual terms learned from a different dataset.

The graphs in Figure 1 show that for all of the four stages, the Hadoop variant is much faster with large amounts of data. With smaller amounts of data (below around 5000 images), the hadoop variant is slower. The reasons for this are two-fold; firstly, there is an inherent overhead in the setup of a Hadoop job — the program needs to be transferred to each of the nodes, and Java virtual machines need to be started for each task. Secondly, with smaller amounts of data, the cluster will be under-utilised as there are likely to be less InputSplits than available processor slots. There are a few things that could be done to remedy this situation (i.e. smaller distributed filesystem block size, resulting in more InputSplits), but there might well be unintended consequences from these. Realistically, it is just a feature of the framework that it doesn't work well with really small amounts of data (relatively speaking).

Table 4 shows by how much the Hadoop-based tools are an improvement over the single-machine multithreaded variants. In all cases, the Hadoop variant improves over a single machine, but exceeds the three-times performance boost we might expect from using three times the amount of hardware. We believe that the extra boost is testament to the way that the MapReduce framework is able to spread I/O across the cluster. In particular, for the single machine tests, all the data has to come off a single disk over a single bus. When using Hadoop, the I/O on a single machine is spread across all disks and multiple buses. For a task like indexing which is almost completely I/O-bound we see really big improvements with the distributed setup.

[20] `http://www.image-net.org`



(a) Feature Extraction

(b) K-Means Clustering

(c) Feature Quantisation

(d) Index Construction

Fig. 1 Time taken for each pipeline stage as a function of number of images. Experiments were performed on a single machine with 16 processors and a 3-machine Hadoop cluster with 48 processors.

We have presented results comparing single machine parallelism to multiple machine parallelism using a MapReduce framework and show a predictable significant improvement. This improvement is to be expected as more machines result in more processing capability, however we would like to draw specific focus to the significance of *how* these multiple machines were utilised. For example, a much simpler scheme is one in which processing is not performed on local data and instead accessed through some shared network file store. Each machine would be told which portion of this shared data store it is responsible for and would proceed to process that portion of the data. In addition, the specific processing each machine should perform would have to be pushed to the machine in some fashion (transfer of binaries etc.). Our implementation of such a scheme uses a standard Unix Network File System (NFS) to give machines access to both data and processing binaries. Each machine then initiates a test, equivalent to our single machine parallelism test, utilising threads to perform the processing on parts of the data. The code for our technique is available on Github[21].

For the test configuration involving feature extraction from 1 million images, we presented single machine parallelism completion times nearing 138000 seconds and MapReduce completion times near 20600 seconds. Using our non-MapReduce, multi-machine approach we achieve completion times near 49000 seconds. These results are to be expected, they show that network transfer notwithstanding, using multiple machines in some fashion is always better than using a single machine. However, we also see the dramatic benefits of data-local processing as provided by the Hadoop MapReduce framework. Furthermore, many conveniences are provided by the MapReduce framework which our custom approach

[21] `https://github.com/sinjax/ssh-imagenet-timeingTest`

had to provide manually (or indeed not provide at all). These conveniences include the automatic splitting of data across jobs, the automatic delivery of processing to be performed to each node, and the relative conceptual ease of the MapReduce paradigm.

6.2 Retrieval

We demonstrate the retrieval capabilities of ImageTerrier and the indexing pipeline by performing a set of experiments with indexes created from a large image corpus using the indexing pipeline described in this paper. We explore the time taken to search these indexes as well as evaluating the performance of the indexes against a standard benchmark. Using the indexing pipeline we can build disk-based indexes that achieve state of the art retrieval results on datasets of over 1 million images with small retrieval times; in particular, we achieve query times of under 0.9 seconds per query with indexes of 1 million images, which compares very favourably to previously published times of more than 15 seconds on similarly sized corpora using disk-based indexes [40].

It is important to note that the retrieval experiments presented here are performed on a single machine using a single thread; we do not aim to demonstrate distributed retrieval performance in these experiments. Our distributed indexing pipeline was configured to produce a single non-sharded index for these experiments. In addition, the experiments were performed with the index stored on a standard 7200 RPM SATA hard-drive. A modern solid-state drive would undoubtedly reduce query times.

Experimental setup. The experiments performed to investigate the performance of the ImageTerrier indexes take the form of a traditional image retrieval or object recognition experiments. The UKBench dataset[22] [37] and evaluation protocol is used as the basis for the experiments presented here; the UKBench dataset consists of 10200 images of 2550 specific objects under varying orientation and illumination conditions. There are 4 images of each object in the dataset. The UKBench retrieval protocol is to take each image in turn as a query and calculate the four best matches (one, usually the first, of which should be the query image itself). A score is assigned based on how many of the top-four images are of the same object as the query (essentially this is equivalent to the precision at four documents retrieved). The score is averaged over all 10200 queries, and has a maximum value of 4.

The indexing pipeline is optimised for dealing with many more images than are present in the UKBench dataset. Therefore, to properly test the technologies we include a set of distractor images into the standard UKBench corpus. For this purpose we use the ImageNet [13] dataset used in the scalability experiments. For the retrieval experiments, we include 0 to 1 million distractor images in increments of powers of 10 to show the effect of large image sets on retrieval performance. In the experiments we perform the complete UKBench evaluation protocol 3 times for each combination of valid and distractor images. The indexes generated are the basic type (i.e. without any extra geometric information). We present retrieval results using 3 score weighting schemes (L1, L1/IDF and TF/IDF) [see 22, for more details]. Starting with 0 distractors and adding more in powers of 10 up to 1 million distractors results in a total of 63 tests (7 distractor combinations performed 3 times, each with 3 scoring strategies). For these experiments we elected to use a general vocabulary of 1 millions terms trained against the MIRFlickr25000 dataset [26], unrelated to both UKBench and ImageNet; this gives us a lower retrieval performance, but is more indicative of a retrieval scenario where the entire corpus is not known prior to indexing. Even though the dataset is different, the approximate k-means clustering was still performed in the distributed manner described in the earlier sections of this paper, using our `HadoopFastKMeans` tool. Results and discussions are presented below.

Performance. In Figure 2 we show the average time taken for queries per experiment. The first thing that should be noted is that, once the index is loaded and initialised, a query is likely to take under 0.9 second with indexes of up to 1 million documents. This is comparable to the query times reported by Nistér and Stewénius, but it should be noted that our inverted index is completely disk-based, whereas Nistér and Stewénius's was held in RAM. For comparison, Philbin et al [40] report query times of between 15s and 35s for a disk-based inverted index of 1.1 million images and 0.1 seconds of an in-memory index of 100,000 images. This result is very important as it shows that using our distributed indexing pipeline,

[22] `http://www.vis.uky.edu/~stewe/ukbench/`

not only can you build indexes quickly and efficiently, but that the resultant indexes can also be searched quickly.

In more detail, the results show that with up to 1000 distractors the query time is constant. Beyond 1000 distractors there is a near-linear increase in the time per query as a function of the number of documents (note the x-axis is logarithmic). This is to be expected as even in an inverted index, the postings list of a given term is likely to increase linearly with number of documents and in turn so will the time taken to score documents. However, the key improvement promised by the inverted index approach comes not in a reduction of the complexity of the algorithm, but rather the gradient of the linear complexity. The inverted index strategy promises a line of much lower gradient than a brute force strategy. We also expect a further drop in this gradient if the searching strategy is extended to work in a distributed manner or the index is placed on a faster hard-drive.

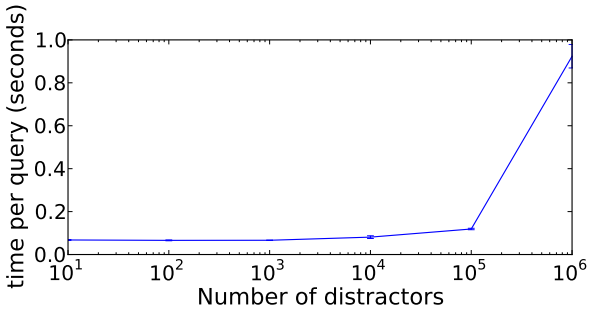


Fig. 2 Time taken for an average query against number of documents in the index.

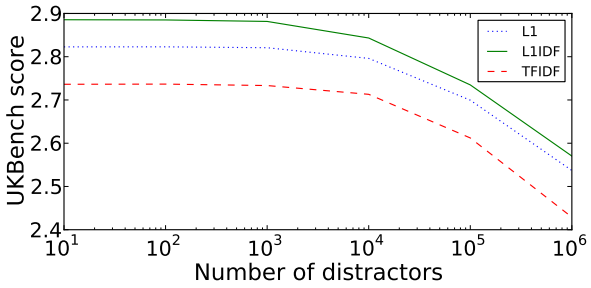


Fig. 3 Average UKBench score for 10,200 UKBench queries given number of distractors. Scores shown for 3 scoring schemes.

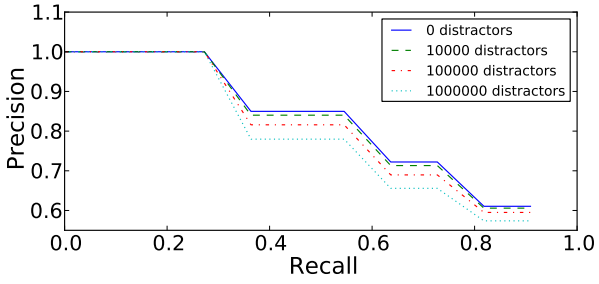
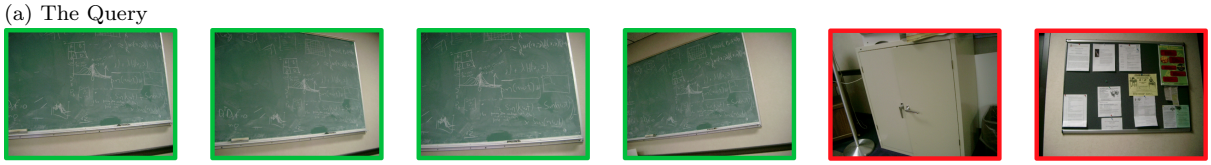


Fig. 4 Average interpolated precision/recall curves for 10,200 UKBench queries against total number of documents. Scores shown for L1IDF scoring scheme.

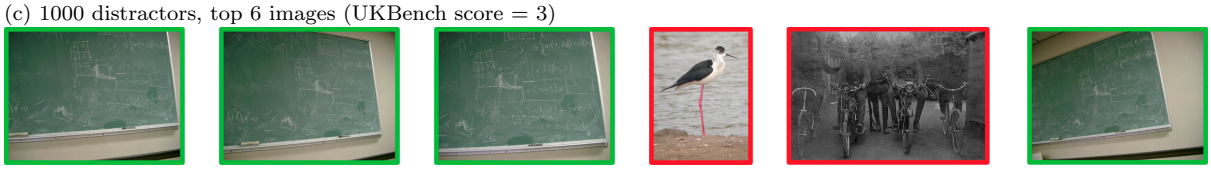
In Figure 3 we show the UKBench evaluation performance as distractors are added. Figure 4 shows the average interpolated precision recall curve across all queries. We notice that the scores fall as distractors

(a) The Query



(b) 0 distractors, top 6 images (UKBench score = 4)

(c) 1000 distractors, top 6 images (UKBench score = 3)



(d) 100000 distractors, top 6 images (UKBench score = 3)

Fig. 5 Example query and top-ranked results with increasing distractors.

are added. This is to be expected; within UKBench, images which only tentatively achieved high scores with images that did not match very strongly can be easily confused when more images are added. An example of such a situation is shown in Figure 5 which shows the same query with 0 distractors, 1,000 distractors and 100,000 distractors. It should also be noted that the best score we demonstrate without distractors is 2.9 which is slightly below the state of the art. However, as previously mentioned, these experiments used a general vocabulary trained against a different dataset unrelated to both UKBench and ImageNet. Using UKBench itself to create a vocabulary from SIFT features we achieve scores near 3.2 with a 1,000,000 term vocabulary (compared to 3.16 reported by [37] using a non-hierarchical vocabulary from MSER-SIFT features). We have achieved our best results with a score of over 3.65 by using ASIFT features [36] quantised with a 8 million term vocabulary (learnt from the ASIFT features themselves).

7 Conclusions

This paper has described in detail how a typical image analysis and indexing pipeline can be efficiently scaled across a distributed cluster of machines. In order to achieve this we have described a set of hybridisations of the original MapReduce framework that allow the processing tasks to be performed effectively on modern server hardware, taking into accounts limits such as the available amount of RAM. The theoretical discussions of the hybridisations of the MapReduce framework have been backed up with actual implementations and experiments to demonstrate their worth, and in particular, we have released the entire suite of tools implementing the bag-of-visual-words indexing pipeline on top of Apache Hadoop as a set of freely available open-source tools. We believe that this is the first work to fully describe a complete scalable image indexing pipeline and provide a baseline implementation.

Using our indexing pipeline tools we have performed a number of experiments to assess the actual performance increases we can achieve by using a distributed system. The experimental results show that the speedup achieved using a cluster of three machines over a single machine outweighs the three-times speedup that might be expected. The extra speedup is due to the way the Hadoop framework is able to distribute I/O across multiple disks and data buses even on a single machine. The resultant disk-based index created by our pipeline has been shown to be as effective as existing state-of-the-art techniques, whilst at the same time being many times more efficient and scalable.

Our approach is based around both commodity hardware and software, utilising both commonly available hardware architectures and common programming paradigms. Therefore, our approach scales

cheaply in terms of hardware. In addition, our approach can be easily understood and extended by researchers and other programmers. Nevertheless, the GP-GPU hardware parallelism solutions we discuss in 2.1 have been supported by Amazon's Elastic MapReduce since late 2010 (see `http://bit.ly/MSDdCE`). Furthermore, GPUs themselves have been dropping in price, increasing in computational power and GP-GPU computational frameworks have been increasing in functionality over the last half decade. With these factors in mind we predict that hybrid solutions that combine MapReduce paradigms with more specialised hardware such as GPUs will become more common over the next decade. As a practical example, it is easy to see how a process like the difference-of-Gaussian interest-point detection used in the SIFT algorithm, can be implemented on a GPU, and rolled into our MapReduce framework, potentially offering the best of both worlds. We are considering actively looking at such a hybrid approach for future versions of our tools.

8 Acknowledgements

The development of the tools and techniques described in this paper was funded by the European Union Seventh Framework Programme (FP7/2007-2013) under grant agreements n° 270239 (ARCOMEM), 231126 (LivingKnowledge) and 287863 (TrendMiner) together with the LiveMemories project, graciously funded by the Autonomous Province of Trento (Italy).

References

1. (2008) Amazon Elastic Compute Cloud (Amazon EC2). Amazon Inc., http://aws.amazon.com/ec2/
2. Akram HI, Batard A, de la Higuera C, Eckert C (2010) Psma: A parallel algorithm for learning regular languages. In: NIPS Workshop on Learning on Cores, Clusters and Clouds, Vancouver, BC Canada, URL `http://lccc.eecs.berkeley.edu/Papers/PSMA_LCCC_2010_NIPS.pdf`
3. Ananthanarayanan R, Esser SK, Simon HD, Modha DS (2009) The cat is out of the bag: cortical simulations with 109 neurons, 1013 synapses. In: Proceedings of the Conference on High Performance Computing Networking, Storage and Analysis, SC '09, pp 63:1–63:12
4. Arthur D, Vassilvitskii S (2006) k-means++: The advantages of careful seeding. Technical Report 2006-13, Stanford InfoLab, URL `http://ilpubs.stanford.edu:8090/778/`
5. Asanovic K, Bodik R, Catanzaro BC, Gebis JJ, Husbands P, Keutzer K, Patterson DA, Plishker WL, Shalf J, Williams SW, Yelick KA (2006) The landscape of parallel computing research: A view from berkeley. Tech. Rep. UCB/EECS-2006-183, EECS Department, University of California, Berkeley
6. Bhuiyan M, Pallipuram V, Smith M (2010) Acceleration of spiking neural networks in emerging multi-core and gpu architectures. In: Parallel Distributed Processing, Workshops and Phd Forum (IPDPSW), 2010 IEEE International Symposium on, pp 1 –8, DOI 10.1109/IPDPSW.2010.5470899
7. Bornschein J, Dai Z, Lücke J (2009) Approximate em learning on large computer clusters. In: NIPS Workshop: Learning on Cores, Clusters and Clouds.
8. Cadambi S, Durdanovic I, Jakkula V, Sankaradass M, Cosatto E, Chakradhar S, Graf HP (2009) A massively parallel fpga-based coprocessor for support vector machines. In: Proc. IEEE FCCM, FCCM '09, pp 115–122
9. Chu CT, Kim SK, Lin YA, Yu Y, Bradski GR, Ng AY, Olukotun K (2006) Map-Reduce for Machine Learning on Multicore. In: Schölkopf B, Platt JC, Hoffman T (eds) NIPS, MIT Press, pp 281–288, URL `http://www.cs.stanford.edu/people/ang//papers/nips06-mapreducemulticore.pdf`
10. Coates A, Baumstarck P, Le Q, Ng A (2009) Scalable learning for object detection with gpu hardware. In: Intelligent Robots and Systems, 2009. IROS 2009. IEEE/RSJ International Conference on, pp 4287 –4293, DOI 10.1109/IROS.2009.5354084
11. Dean J (2009) Challenges in building large-scale information retrieval systems: invited talk. In: Proceedings of the Second ACM International Conference on Web Search and Data Mining, ACM, New York, NY, USA, WSDM '09, pp 1–1, DOI 10.1145/1498759.1498761, URL `http://doi.acm.org/10.1145/1498759.1498761`
12. Dean J, Ghemawat S (2004) Mapreduce: simplified data processing on large clusters. In: Proceedings of the 6th conference on Symposium on Opearting Systems Design & Implementation - Volume 6, USENIX Association, Berkeley, CA, USA, OSDI'04, pp 10–10, URL `http://dl.acm.org/citation.cfm?id=1251254.1251264`

13. Deng J, Dong W, Socher R, Li LJ, Li K, Fei-Fei L (2009) ImageNet: A Large-Scale Hierarchical Image Database. In: CVPR09
14. Farivar R, Verma A, Chan E, Campbell RH (2009) Mithra: Multiple data independent tasks on a heterogeneous resource architecture. In: CLUSTER, IEEE, pp 1–10
15. Forum MP (1994) Mpi: A message-passing interface standard. Tech. rep., Knoxville, TN, USA
16. Franc V, Sonnenburg S (2008) Optimized cutting plane algorithm for support vector machines. In: ICML '08: Proceedings of the 25th international conference on Machine learning, ACM, New York, NY, USA, pp 320–327, DOI 10.1145/1390156.1390197, URL http://dx.doi.org/10.1145/1390156.1390197
17. Ghoting A, Krishnamurthy R, Pednault E, Reinwald B, Sindhwani V, Tatikonda S, Tian Y, Vaithyanathan S (2011) Systemml: Declarative machine learning on mapreduce. In: Data Engineering (ICDE), 2011 IEEE 27th International Conference on, pp 231 –242, DOI 10.1109/ICDE.2011.5767930
18. Graf HP, Cosatto E, Bottou L, Durdanovic I, Vapnik V (2004) Parallel support vector machines: The cascade svm. In: NIPS
19. Gregg C, Hazelwood K (2011) Where is the data? why you cannot debate gpu vs. cpu performance without the answer. In: International Symposium on Performance Analysis of Systems and Software, Austin, TX, ISPASS
20. Hamada T, Iitaka T (2007) The chamomile scheme: An optimized algorithm for n-body simulations on programmable graphics processing units. URL http://www.citebase.org/abstract?id=oai:arXiv.org:astro-ph/0703100
21. Hare J, Samangooei S, Lewis P (2011) Efficient clustering and quantisation of sift features: Exploiting characteristics of the sift descriptor and interest region detectors under image inversion. In: The ACM International Conference on Multimedia Retrieval (ICMR 2011), ACM Press, URL http://eprints.ecs.soton.ac.uk/22237/
22. Hare J, Samangooei S, Dupplaw D, Lewis P (2012) Imageterrier: An extensible platform for scalable high-performance image retrieval. In: The ACM International Conference on Multimedia Retrieval (ICMR 2012)
23. Hare JS, Samangooei S, Dupplaw DP (2011) OpenIMAJ and ImageTerrier: Java libraries and tools for scalable multimedia analysis and indexing of images. In: Proceedings of ACM Multimedia 2011, ACM, MM '11, pp 691–694, DOI http://doi.acm.org/10.1145/2072298.2072421, URL http://doi.acm.org/10.1145/2072298.2072421
24. Heinz S, Zobel J (2003) Efficient single-pass index construction for text databases. J Am Soc Inf Sci Technol 54:713–729
25. Hoefler T, Lumsdaine A, Dongarra J (2009) Towards Efficient MapReduce Using MPI. In: Recent Advances in Parallel Virtual Machine and Message Passing Interface, 16th European PVM/MPI Users' Group Meeting, Springer
26. Huiskes MJ, Lew MS (2008) The MIR Flickr Retrieval Evaluation. In: MIR '08: Proceedings of the 2008 ACM International Conference on Multimedia Information Retrieval, ACM, New York, NY, USA
27. Jegou H, Douze M, Schmid C (2008) Hamming embedding and weak geometric consistency for large scale image search. In: Proceedings of ECCV 2008, Springer-Verlag, Berlin, Heidelberg, ECCV '08, pp 304–317
28. Kadir T, Brady M (2001) Saliency, scale and image description. Int J Comput Vis 45(2):83–105
29. Li Y, Crandall D, Huttenlocher D (2009) Landmark classification in large-scale image collections. In: Computer Vision, 2009 IEEE 12th International Conference on, pp 1957 –1964, DOI 10.1109/ICCV.2009.5459432
30. Lowe D (2004) Distinctive image features from scale-invariant keypoints. IJCV 60(2):91–110
31. Macqueen JB (1967) Some methods of classification and analysis of multivariate observations. In: Proceedings of the Fifth Berkeley Symposium on Mathematical Statistics and Probability, pp 281–297
32. Matas J, Chum O, Urban M, Pajdla T (2002) Robust wide baseline stereo from maximally stable extremal regions. In: Rosin PL, Marshall AD (eds) BMVC, British Machine Vision Association
33. McCreadie R, Macdonald C, Ounis I (2011) Mapreduce indexing strategies: Studying scalability and efficiency. Information Processing and Management DOI 10.1016/j.ipm.2010.12.003
34. Mikolajczyk K, Schmid C (2003) A performance evaluation of local descriptors. In: CVPR, vol 2, pp 257–263

35. Mikolajczyk K, Tuytelaars T, Schmid C, Zisserman A, Matas J, Schaffalitzky F, Kadir T, Gool LV (2005) A comparison of affine region detectors. IJCV 65(1/2):43–72
36. Morel JM, Yu G (2009) ASIFT: A New Framework for Fully Affine Invariant Image Comparison. SIAM J Img Sci
37. Nistér D, Stewénius H (2006) Scalable recognition with a vocabulary tree. In: CVPR, pp 2161–2168
38. O'Hara S, Draper BA (2011) Introduction to the bag of features paradigm for image classification and retrieval. CoRR
39. Philbin J (2010) Scalable object retrieval in very large image collections. PhD thesis, University of Oxford
40. Philbin J, Chum O, Isard M, Sivic J, Zisserman A (2007) Object retrieval with large vocabularies and fast spatial matching. In: CVPR
41. Raina R, Madhavan A, Ng AY (2009) Largescale deep unsupervised learning using graphics processors. In: International Conf. on Machine Learning
42. Rice KL, Taha TM, Vutsinas CN (2009) Scaling analysis of a neocortex inspired cognitive model on the cray xd1. J Supercomput 47:21–43
43. Shan Y, Wang B, Yan J, Wang Y, Xu N, Yang H (2010) Fpmr: Mapreduce framework on fpga. In: Proceedings of the 18th annual ACM/SIGDA international symposium on Field programmable gate arrays, ACM, New York, NY, USA, FPGA '10, pp 93–102, DOI http://doi.acm.org/10.1145/1723112.1723129, URL `http://doi.acm.org/10.1145/1723112.1723129`
44. Sivic J, Zisserman A (2003) Video google: A text retrieval approach to object matching in videos. In: ICCV, pp 1470–1477
45. Smeulders AWM, Worring M, Santini S, Gupta A, Jain R (2000) Content-based image retrieval at the end of the early years. PAMI 22(12):1349–1380, DOI http://dx.doi.org/10.1109/34.895972
46. Subramanya A, Bilme J (2009) Large-scale graph-based transductive inference. In: NIPS workshop on Learning on Cores, Clusters and CloudsLarge-Scale Machine Learning: Parallelism and Massive Datasets
47. White B, Yeh T, Lin J, Davis L (2010) Web-scale computer vision using mapreduce for multimedia data mining. In: Proceedings of the Tenth International Workshop on Multimedia Data Mining, ACM, New York, NY, USA, MDMKDD '10, pp 9:1–9:10, DOI http://doi.acm.org/10.1145/1814245.1814254, URL `http://doi.acm.org/10.1145/1814245.1814254`
48. Xu NY, Cai XF, Gao R, Zhang L, Hsu FH (2009) Fpga acceleration of rankboost in web search engines. ACM Trans Reconfigurable Technol Syst 1:19:1–19:19
49. Ye J, Chow JH, Chen J, Zheng Z (2009) Stochastic gradient boosted distributed decision trees. In: CIKM '09: Proceeding of the 18th ACM conference on Information and knowledge management, ACM, pp 2061–2064