A Predictive Model for the Parallel Processing of Digital Libraries

A thesis

submitted in fulfilment

of the requirements for the degree

of

Doctor of Philosophy

at the

University of Waikato

by

John Thompson

THE UNIVERSITY OF
WAIKATO
Te Whare Wānanga o Waikato

University of Waikato

2015

Abstract

The computing world is facing the problem of a seemingly exponential increase in the amount of raw digital data, and the speed at which it is being collected, is eclipsing our ability to manage it manually. Combine this with the increasing expectations of a growing number of experienced computer users—including real-time access and a demand for expensive-to-process file types such as multimedia—and it is not hard to understand why managing data of this scale and providing timely access to useful information requires specialized algorithms, techniques, and software.

Digital libraries are being used to help address these challenges. Drawing upon knowledge learned through traditional library science, digital libraries excel in providing structured user access to a wide variety of documents. They increasingly include tools for managing, moderating, and marking up these documents. Furthermore, they often feature phases where documents are independently processed and so can benefit from the application of parallel processing techniques—the focus of this thesis. Whether a digital library collection can benefit from parallel processing depends on considerations such as document type, processing cost per document, number of documents, and file-system input/output.

To aid in deciding when to apply parallel processing techniques to digital libraries, this thesis explores the creation a model for predicting key outcomes of leveraging such techniques. It does so by implementing parallel processing in three distinct open-source digital library tools, undertaking experiments designed to measure key processing features (such as processing time versus number of compute nodes), and applying machine learning techniques to these features in order to derive a predictive model.

The model created predicts parallel processing performance at 96% accuracy (adjusted r^2) for a number of exemplar collection types. The result is a generally applicable tool for estimating the benefits of applying parallel processing to a wide range of digital collections.

Acknowledgements

First and foremost I wish to thank my wife, Dr Lin-Yi Chou, for her support, encouragement, and help. It was her journey to a PhD that inspired me to start my own, and her knowledge on mathematical models proved invaluable as I tackled developing my own. I would also thank my daughter Victoria, whose impending birth provided excellent incentive to hurry-up and finish writing... and whose smiles and giggles since have helped me get through revisions. I also owe much to my family and friends who never grew tired of my lengthy explanations/short lectures of what my research entailed (or at least were very good at pretending not to grow tired).

This thesis would not have happened without the valuable guidance and feedback from my supervisors, Associate Professor David Bainbridge and Professor Ian Witten. My sympathy to the several red pens they have worn out while error-checking my writing.

I would also like to extend my thanks to the Technical Support Group at the Department of Computer and Mathematical Sciences—who kindly set-up the *Medusa* cluster and provided help and expertise during the (many) times I encountered/caused technical issues—and acknowledge the support of the Department and the University of Waikato by way of Scholarships and resources.

Contents

List of Figures

List of Tables

Listings

Chapter 1

Introduction

> We've heard that a million monkeys at a million keyboards could produce the complete works of Shakespeare; now, thanks to the Internet, we know that is not true.
>
> — *Robert Wilensky, "Mail on Sunday", 1997*

At first glance parallel processing seems deceptively simple. Start with a large, unwieldy, task, break it up into a number of smaller subtasks, and act on as many of those pieces at the same time as possible. The goal is to significantly decrease the overall time needed to complete the original task. However, as the opening quote cynically hints, there is more complexity to this than just throwing more 'monkeys' at it.

First, there are issues of how inter-related the smaller parts are. While sometimes the pieces are completely independent and will benefit greatly from processing in parallel, at the other extreme there are tasks that must be performed sequentially and would not benefit at all from being split apart. In practice, there is typically a mix of the two. This balance is captured by Amdahl's Law [8], which is used to find the maximum expected improvement to an overall system when only part of the system can be arbitrarily improved while the remainder is fixed in terms of performance.

A second complexity is the overhead of managing the sub-tasks across multiple processors. This includes algorithms for dividing and scheduling subtasks—with more sophisticated techniques often being computationally expensive in their own right—and also the file transfer costs of getting the data to and from the processors. While techniques such as data locality can offset the latter, it is often the case that parallel processing frameworks simply move the critical path bottleneck from the computer's processors to its file system.

The demand for faster processing is based upon an explosion of raw data over the last few decades. An International Data Corporation report [73] estimated the amount of digital data worldwide at 130 exabytes in 2005, with that number expected to grow significantly to 40 zettabytes by 2020. This latter number equates to more than $5,000$ gigabytes of raw data for each person on Earth. People themselves are not solely responsible for the increase: the greatest growth in production comes from machine generated data. All this leads to the nebulous term *Big Data*—raw data of such size that humans cannot turn it into useful information without machine assistance. At the moment such transformation efforts are in their infancy, with an estimated 3% of the collected raw data actually accessible in some useful way and less than 1% of that being analysed—despite an estimated 33% of this data being valuable.

This has resulted in a surge of activity in computer science fields that address scaling challenges. Some are recent, such as data mining and modelling, while others hark back to the origins of information science, such as organization through taxonomies and metadata. The area of focus for this thesis, digital libraries, provide structured access to a variety of disparate documents and increasingly include tools and procedures for managing, moderating, and marking up these documents.

This thesis explores the benefits of applying parallel processing to digital libraries, in particular those whose scale puts them firmly in the camp of "large, unwieldy, task". Due to the wide range of objects these very large-scale digital libraries contain, the equally wide range of processes that can be applied to said objects, and the software development and hardware costs of parallel processing, it is difficult to determine whether applying parallel techniques will result in a justifiable benefit.

This thesis presents a predictive model, based upon the results of practical serial and parallel digital library implementations, that may be used to determine key features such as potential speed-up benefit. Several pieces of general purpose open-source digital library software have been extended to use parallel processing, using existing parallel frameworks such as Open MPI and Hadoop, and then subjected to a battery of experiments to gather the data that underpins the model. This data is then imported into two data analytic tools, Weka and R, that allow the programmatic creation of several different mathematical models. Using an understanding of the problem, result data, and accuracy metrics from the tools themselves, the best model is selected and presented as the principle finding of this research.

This remainder of this chapter defines the hypothesis underlying this research, explaining its origin, motivation, and value. It also describes the three major components mentioned above, namely *very-large scale digital libraries*, *parallel building*, and *predictive models*. Following that, the scope of the research is delineated, along with the questions used to determine whether the hypothesis has been answered.

1.1 Thesis Statement

The central hypothesis underlying this research is:

> A predictive model can be developed that accurately predicts when it is appropriate to apply parallel building to very large-scale digital library processing.

The hypothesis has three distinct aspects that should be explained at the outset:

- **Very large-scale digital libraries** - those digital libraries that have reached a threshold—in terms of size of current data, rate of acquisition of new data, or demand and numbers of users—such that manual maintenance becomes difficult without automated tools or advanced algorithms for processing.
- **Parallel building** - involves taking the single large batch process that typically builds a digital library, dividing it up into smaller independent parts, and then distributing this work over multiple processors.
- **Predictive modelling** - refers to a mathematical formula or other construct derived from a specific process that, given certain inputs, can predict output values that approximate those the actual process would produce.

The key measure for this model is its "accuracy"—how closely its predictions match those seen in real-world examples—as determined by the calculation of statistical measures, for example correlation coefficient and root mean square error. Similarly, the word "appropriate" is purposely broad since there are interesting features this model can predict other than an indication of the total elapsed processing time; Section 1.3 will expand this.

1.2 Motivation

The hypothesis for this thesis arose as part of a practical project to build a very large-scale newspaper digital library. The commercial digital library company DL Consulting undertook a project that called for the creation of a digital library initially containing some 600,000 digitised newspaper pages. The collection needed to support weekly batch additions, with the final size expected to exceed 2 million pages. This project exposed challenges with collections at this scale—for example, the untenable situation where adding a weekly batch would take longer than a week to process.

In order to address these challenges, DL Consulting experimented with distributed computing, investigating distributed back-ends such as Oracle's DB2 database, and incremental and parallel collection building available via the Apache Lucene technology. Ultimately, incremental solutions proved suitable for the project at hand, but the question remained about the applicability of parallel processing techniques.

Around this time, Dr Hussein Suleman visited the Greenstone Digital Library Laboratory at the University of Waikato. Greenstone is a widely-used, open-source digital library that provides a flexible platform for experimentation, but had an undeserved reputation for being applicable only to small scale collections. In order to explore the processing of very large-scale digital libraries, Dr Suleman drew upon his expertise in distributed indexing architectures [140], and prototyped a version of Greenstone collection building that made use of the parallel processing Open MPI framework. This work allowed Greenstone to leverage the power of multi-core computers.

While experimentation showed that using parallel processing did decrease the time needed to import large-scale text collections [173], the benefit was far less than theoretically possible [77]. Applying the same parallel implementation to a different digital library project [155], whose import materials consisted of a series of multimedia files, resulted in a significant improvement in performance.

These results illustrated the fact that parallel processing is not a magic bullet, as stated in E.M. Rasmussen's influential paper "Introduction: Parallel Processing and Information Retrieval":

> The use of multiple processors does not guarantee an improved processing rate, and several cases have been reported in which an

> efficient serial algorithm for an Information Retrieval application may outperform a parallel one for the same task (Rasmussen & Willett, 1987; Stone, 1987; Salton & Buckley, 1998). The successful implementation of a parallel solution requires an appropriate match of task, algorithm, and architecture. [150]

The experimental findings lead to the question of whether it is possible—given knowledge of the specific hardware configuration, parallel processing framework, and properties of documents to be imported—to create a model that predicts whether parallel building is superior to serial building.

1.3 Contribution

The scenario outlined in Section 1.2 is common in very large-scale digital library construction, and prompts the following questions:

- How long will it take to process the content of the digital library, given a fixed hardware and desired processing configuration?
- How much processing can be completed within a fixed time period and with given hardware?
- What is the optimal hardware configuration given a desired time period and processing configuration?

The novel contribution of this research is a predictive model that answers these questions. Such a model allows a digital librarian—or similarly expert user—to determine whether applying parallel processing to their intended digital library will be beneficial. Use of the model requires some preparatory work to gather input data; most taxing of which is estimating the ratio of processing versus file system cost for the task at hand.

This predictive model can also be applied to determine the number of processing cores for multiple-core machines, or compute nodes for clusters. It can also set bounds on how much processing can be applied within a certain time.

The above three questions explore key features of any digital library building process, each optimising a particular outcome, and thus are important for users looking at ways to establish, configure, and manage very large-scale

digital libraries. The model will be considered successful if it can predict answers to these questions with reasonable accuracy. It should be noted that the first and second questions are closely related to Amdahl's Law [8] and its counterpoint Gustafson's Law [81] respectively, as described in Chapter 3.

1.4 Research Scope

The digital library field is a broad church, spanning both the software and hardware domains, hence it is important to clearly establish the scope of the work reported here.

One issue lies in the definition of a digital library. This research focuses on those digital libraries that include an initial batch process, or *import* phase, where a significant amount of pre-processing is carried out. For instance, multimedia files are converted into a standard, web–streamable, format, while text-based files undergo indexing in order to make them easily searchable. While we concentrate our experiments on this import phase, we acknowledge that there are also interesting questions about applying parallel processing to the user-interaction side of digital libraries. An example of the former is CiteCeer^X [111], which has a large-scale import phase; an example of the latter is the Alexandria Digital Library Testbed [67], which allows interacting with a collection of digitized maps using image processing tools with the workload distributed over several web-servers. We disregard other forms of digital libraries that have no import stage, by requiring documents to already be homogeneous or pre-processed (effectively skipping over the import stage), or that blur the lines between import-time and run-time phases. The reference implementation of the Fedora Project [145] is an example of the former, while the open-source content management system Drupal [37] is an example of the latter.

Another issue is providing a measure of the amount of processing work a computer core needs to do. Such a metric would need to somehow capture the complexity involved in a particular processing configuration offset by the performance of the underlying hardware. This metric, once determined, would also provide a means to calculate the ratio between computer processor utilisation and file system utilisation; something which would later be found to be strongly predictive of the benefits of parallel processing. However, it proves difficult on a modern operating system—running something that combines computation and file access—to accurately predict the complexity cost of a particular process. Accurate measurements are best determined from an actual performance of that process. Thus, when using the model developed

through this research, a user is expected to either select the most similar ratio from a list of exemplar collections, or to perform a serial test build over a small example collection and measure this ratio.

Both hardware specification and network configuration have a significant effect on processing performance. However, this research focuses on the software aspects of parallelism, and only briefly touches on these hardware issues through the issue of file systems and, in particular, distributed file systems (see Section 3.4.3). While we choose to limit hardware concerns there is significant research showing that the choice of hardware and network topography can have a dramatic effect on performance. For example, parallel processing performance can be altered by using weighted/complex segmented topographies [152], optimisation of operating system to specifically support parallel processing [196], or matching network configuration with network topography [113].

1.5 Thesis Structure

This thesis is structured as follows. Chapter 2 discusses very large-scale digital libraries, the issues they raise, and lists a number of real world case studies. This includes: the project that prompted this research (PapersPast [1]); a large-scale project that has drastically benefited from parallel processing (ReplayMe! [155]); and a number of successful, contemporary, libraries that use proprietary software to address the issues of scale.

Chapter 3 introduces parallel processing as a practical solution to the scale issues raised above. It lists some of its advantages and potential challenges, and categorises approaches into a spectrum ranging from highly efficient, tightly coupled "Share Everything" approaches to the hardware agnostic "Message Passing" approaches that trade efficiency for flexibility. The two major frameworks that this research adopts, *Open MPI* [69] and *Hadoop* [198], are explained and contrasted. This chapter concludes by discussing some of the practical challenges encountered when implementing parallel processing, and their solutions.

Chapter 4 then looks at implementations of digital libraries capable of parallel processing. It details the combination of the aforementioned parallel processing frameworks with several general purpose digital library software suites, namely *Greenstone* [203], *DSpace* [165], and *Terrier IR* [143].

Chapter 5 presents a series of experiments run using these three implementations. The results provide compelling evidence for the potential for parallel processing to address some of the issues of scale that are not easily solved during serial digital library importing. The experiments make use of four distinct corpora of document types and explore three distinct levels of metadata processing applied to the documents as they are imported. One experiment in particular—designed to measure the ratio of processing load versus file transfer costs for each of the document type and metadata extraction permutations—is highlighted as being critical to the development of the research model. The chapter concludes with a series of other experimental results that are of general interest in the field of parallel processing, despite not contributing directly to the predictive model.

Chapter 6 explains the process of creating the mathematical model—the Parallel Processing of Digital Libraries (PPoDL) model—from the aforementioned experimental results. The gathered data is processed using the statistical language R [149] and the machine learning workbench Weka [91]. The model is then evaluated in terms of its accuracy, which exhibited performance of approximately 96% accuracy (measured in adjusted r^2) across the test collections.

Chapter 7 revisits the equations that form the PPoDL model and provides several examples of its application, while discussing the model's accuracy under certain collection configurations.

The final chapter provides a brief summary of the research and then revisits any threats to validity or caveats relating to the model and its use. The thesis concludes with a description of several potential research opportunities that this research has uncovered.

Chapter 2

Very Large-scale Digital Libraries

> To ask why we need libraries at all, when there is so much information available elsewhere, is about as sensible as asking if roadmaps are necessary now that there are so very many roads.
>
> — *Jon Bing, "American Libraries Magazine", 2009*

This chapter introduces the challenge of very large-scale data and its features, and advocates the use of digital libraries to help manage this challenge. We start by providing a brief overview of what digital libraries are and how we might classify them, and then describe some case studies that highlight existing very large-scale digital libraries with interesting or exemplar properties. The chapter ends by detailing the three general purpose, open-source, digital library applications used in this research.

2.1 Background

The last few decades have seen an explosion in the amount of digital data being collected. While the digital universe has always comprised of a *variety* of digital objects, recent trends have seen the traditional media of text and images supplanted by audio and video, which offer higher fidelity but at the cost of significantly more data *volume*. As mentioned in the introduction, estimates put the size of the digital universe in 2015 at around 10 zettabytes [73]. Moreover the rate of growth in digital data is now of such *velocity* that it approximates Moore's Law [138]—doubling every two years.

These three dimensions of scale—variety, volume, and velocity—also appear in the report [108], which is credited with providing the basis for what is now generally referred to as *Big Data*. While the term itself may have been hijacked by software advertisers and is now applied to a wide range of incidentally related issues, the idea it originally tried to capture is pertinent to the focus of the work reported here and in computer science in general. There have also been several more "v"-prefixed dimensions suggested in the literature to create a more accurate definition of very large-scale, including:

- **Visualization**. While less of a measure and more of a means of understanding and managing data of this scale, visualization techniques and tools have been suggested as another potential dimension. An practical example is the novel research in visualizing complex algorithms [86] and large-scale data sets [175] which informed NASA when they began experimenting on how to manage the large data sets that they were capturing from their radio telescopes and other sensing equipment [50].
- **Veracity**. IBM proposed a measure of Veracity of data [48]—not so much judging the provenance of the data itself, but instead reflecting that different users of the data have varying thresholds and requirements for trustworthiness.

The problem with data at this scale arises because our ability to process large amounts of data into valuable information has not kept up with such rapid growth. To date, less than half a percent of the digital universe has been tagged or otherwise transformed into something useful [73]. The word "valuable" is subjective but it is used here to describe data that is analytically valuable and can be processed or refined to provide information beyond that explicit in the original data. It has been estimated that 33% of the digital universe might be mined for this valuable additional information [73].

Thus the issue at hand is matching this explosion of data with the development of systems for making the information contained therein accessible. The approach this research focuses on is digital library software as it provides a structured, browse-able, and searchable gateway to a moderated and metadata annotated collection of electronic documents. Digital libraries are now being created that advance into the millions if not tens of millions of documents—for example, the HathiTrust Digital Library [182] has, at the time of writing, full text indices for more than thirteen million volumes and in excess of four billion pages. Such libraries are termed very large-scale, yet another attempt to demarcate the threshold at which manual processing of data becomes impractical

without specific systems to address the scale.

The creation and practical use of very large-scale digital libraries has also raised many new issues, such as those discussed during the Very Large Digital Library (VLDL) international workshop series [123], particularly related to size (in aspects such as disk space, number of documents, and number of unique terms or words), sustainability, interoperability, and user interaction. These are similar to the dimensions discussed earlier in the chapter. Early assumptions that large-scale digital libraries would be the same as large-scale databases, and thus part of a well understood problem, have proven to only be half the story. While the storage of the underlying data can be tackled with database approaches, issues such as indexing the full-text of documents for searching and the growing expectations and differing ways that users interact with systems of this scale require new approaches to resolve.

More will be said about the dimensions of scale later in this chapter, where a number of very large-scale digital libraries will be presented along with a brief case study for each, to help populate the space of scale, variety, and diversity with approaches already in use. Where possible the technologies underlying each library will be named, and any limitations or proposed future developments detailed. The next section will define what a digital library is and how it is different from similar technology such as web search engines.

2.1.1 The Digital Library Environment

This section establishes what a digital libraries is and details a number of ways, critical to this research, in which they may be categorised. The generally agreed definition is that a digital library is a focused collection of digital documents within a tool that allows them to be organized, stored, and retrieved [40]. They are differentiated from web search engines, such as Google, by the fact that they contain focused material, moderated by a specific person or persons, and they tend to cater to a specific user group and their expectations. Another key difference is that digital libraries are about more than just searching. Li and Furht [112], for example, suggest that digital library activities can be grouped into five areas: discover,[1] retrieve, interpret, share, and manage. Note that a digital library system may also be interchangeably referred to as "digital asset management" or "open access repository" systems.

Digital libraries vary greatly in terms of document type and number, user requirements, and interface. One way to differentiate software is in terms of

[1] includes the activity of searching

whether it is designed for a specific purpose or if it is for a more general purpose. There are already examples of bespoke very large-scale digital libraries engineered to manage a certain type of document or solve a specific problem, several of which are discussed in the case studies below. However, these are generally closed-source or proprietary projects, or are so specialized as to not be useful in other situations. This research explores solving the scale issues on general purpose, open-source, digital libraries; those that may be applied to any collection of digital media, and thus cannot be optimised to a specific document type.

A second way to separate digital library types is the method by which new documents are added. One technique is to prepare a number of documents for import into the system all at once. This process is generally known as batch processing and has the benefits of:

- Reasonably efficient use of computer resources without the need of constant human supervision.
- Makes use of indexing and compression techniques in a straightforward manner.
- Generally produces better indexes (in terms of both size and speed of retrieval).

However, there may be a significant number of documents in a batch and, as indexing time is directly proportional to with the number of documents, large batches may take significant time to process.

The main alternative strategy to batch processing is incremental building. In the early days of digital library research, the addition of a single document to an index in an user-interactive fashion was inefficient, precluded the use of complex compression algorithms, and produced non-optimal indexes. However modern indexing techniques have been specifically designed to efficiently support both batch and incremental processing. Consider the saving of incrementally adding a single new document to an existing large scale collection against the cost of processing the entire collection again, only this time with the new document in the mix. The emergence of web-based technologies along with the drastic speed-ups in processing provided by modern technologies [171] meshes well with user-interactive addition. For example, a user can fill in details about a new document in a web-form and then submit for incremental addition. This leads to a scenario where a vast population of users are the initiators of new documents being including in the digital library—a potential

solution to the issue of large scale volume by effectively crowd-sourcing the import workload.

This research will focus on digital libraries that provide batch processing capabilities as these offer the greatest potential of benefiting from parallel processing. This is not to say that the libraries chosen do not also support incremental addition, nor that there are not possibilities for parallel processing in incremental addition. It should also be noted that incremental addition via web-based systems, is, in most cases, already geared towards a form of parallel execution due to the asynchronous nature of the Internet protocols. This is especially likely if the system supports multiple server threads performing incremental additions at once.

Any suggestions that incremental systems appear consistently superior should be balanced with the argument that even incremental addition capable digital libraries may occasionally require a batch processes. For example, if a significant change is made to the data parsed from the documents (for instance an improved automated keyword identification algorithm has been added to the system), a new indexing technique is adopted, or in the case of a full index rebuilt after a catastrophic event such as corrupted disks.

A third aspect used to categorise digital libraries is the choice of underlying data storage. As mentioned in the definition, one of the important tasks of a digital library is the storage of information and while there are methods and technologies to accomplish this, most solutions generally fall on a continuum from database-backed to index-backed.

Database-backed libraries use database technologies to store each document and its associated metadata as a record in a database. Examples of these databases might include MSSQL, MySQL, and Oracle. Historically they have been strong performers when searching the database for records whose specific fields matched certain values (also known as *Fielded Searches*) and fast at retrieving specific original documents and their metadata, but slow and inefficient when attempting to perform full-text search. Contemporary databases incorporate full-text indexing techniques to counter this weakness. This thesis uses DSpace as an example of a database-backed digital library.

At the other end of the spectrum, Index-backed libraries use established algorithms—such as inverted indexes—to allow the quick retrieval of a document based on search terms. Historically they have been strong at full-text and fuzzy searches, but slower (or even completely unable) to retrieve original documents and metadata. Similar to database development, contemporary

indexing tools address this weakness and can now store a complete copy of the original document and its metadata in a manner as efficient as large-scale databases. The need for fast access to the metadata was prompted by advanced searching techniques, such as phrase searching (which requires not only what document a word occurred in, but the ordering of all the words) and faceted searching (which builds upon phrase searching). This thesis uses the Terrier IR Platform as an example of an index-backed digital library.

Whereas modern digital libraries benefit from the advances (and arguably convergence) in the aforementioned underlying technologies, earlier ones could not. They instead made use of both back-end technologies, in a hybrid manner and to varying degrees, in order to leverage the strengths of each. Such a hybrid system, for example, could store the document's file-system location and other metadata in a database, while building several full-text indexes for use in fast searching. This thesis uses Greenstone as an example of a digital library using a hybrid of both database and indexing technologies.

While there are other ways of differentiating and categorising digital libraries, this section has specifically described three—namely: measure of general purposefulness, whether they support batch import, and the underlying storage technology utilised—as they are of significant consequence to this research. This research will elicit how parallel processing can have practical applications over a range of digital library applications, and will show that a general model can be created to predict performance features even taking into account variances caused by differing storage technologies.

2.2 Real World Case Studies

The following section presents a number of case studies of very large-scale digital libraries. The reason for the section is two-fold: firstly, it provides a practical understanding of the features that make up the large-scale realm; and secondly it differentiates the research presented in this thesis from other existing projects and highlights the novelty of its approach. An overview of the publicly known features of these digital libraries is presented in Table 2.1.

2.2.1 HathiTrust

Hathi (pronouced *hah-tee*) is the Hindustani word for Elephant, and in 2008 the newly launched HathiTrust Digital Library is, aptly so, the largest digital

Table 2.1: Overview of case study features

Est.	**Project**		Media		
	Technology		Challenges		
	No. Docs.	Data size	Index size	Institutions	Velocity
1994	**Informedia**		Video		
	Proprietary		Novel video processing and search techniques		
	1,500 h	1 TB	-	-	90 GB/mth
1996	**CADLIS**		Text, Images, and Multimedia		
	Proprietary		Number of users		
	36 Mil.	-	-	1,000	-
1996	**The Internet Archive**		Web pages and Images		
	Proprietary		Scale and date-based versions		
	150 Bil.	10 PB	-	-	100 TB/mth
1997	**CiteSeer^X**		Research papers (full text)		
	Solr		AI and Data mining		
	2 Mil.	-	-	-	-
1998	**Google**		Web pages and Images		
	Proprietary		Every scale challenge imaginable		
	40 Bil.	-	-	-	-
2000	**LOC: Web Archives**		Web pages and Images		
	Lucene		Date-based versions and search		
	-	525 TB	-	-	5 TB/mth
2008	**HathiTrust**		OCR Text, Images and Metadata		
	Solr		Number of documents		
	13 Mil.	569 TB	6.0 TB	90	-
2010	**LOC: Tweet Archive**		Text (140 characters)		
	-		Velocity, Novel search techniques		
	350 Bil.	270 TB	-	-	20 TB/mth
2013	**Elephind**		Newspapers (HTML and Images)		
	Greenstone and Solr		OCR quality, Crowd-sourced error correction		
	141 Mil.	65 TB	325 GB	19	-

preservation project in the western world. Their goal: the digitisation, preservation, and sharing of the record of human knowledge [182]. The HathiTrust is a collaboration by 98 university and research libraries, including those in the Committee on Institutional Cooperation [10], the University of California system, and the University of Virginia. It has also formed partnerships with other large-scale digitisation projects including the Open Content Alliance and Google Book Search [4].

The project is large-scale in both operational and technical terms. The former requires significant inter-institutional cooperation to fund, create, support, and govern an infrastructure of this magnitude. The latter involves issues of variability of content, provision of several Software-as-a-Service [192] layers, and the number and special requirements of users (such as disabilities). Many of the challenges encountered are not new, having been faced by library alliances in the past, but the scale of the project has made them even more challenging. The project is also certified through Trustworthy Repositories Audit and Certification [7] and the Digital Repository Audit Method Based On Risk Assessment [131].

The technical issues faced in launching the project were also unique and certainly deserving of the label very large-scale. The project built upon a repository initially created as part of the Google Print/Google Book Search project and contained some two million books with over one billion scanned images of pages at the time. The HathiTrust Digital Library contains approximately seven million books and serials, resulting in approximated 495 terabytes of materials (mostly images) and a further 400–500 gigabytes of index. Due to the overarching goal of preservation the aforementioned images are of high quality and are in non-lossy formats such as JPEG2000 and TIFF Group4 compression. The remaining materials include a significant amount of metadata associated with the books, in the Metadata Encoding Transmission Standard (METS) [52] and PREMIS [42] standards. Most images have undergone optical character recognition to extract full-text, though the quality of extracted text varies greatly.

It should be noted that this “dirty” text, as HathiTrust terms it, has been shown to complicate index building [15]. Instead of there being a theoretical maximum limit to the number of unique terms encountered [184], low accuracy optical character recognition can produce a near limitless number of unique terms due to combinations and permutations of random junk characters.

The HathiTrust Digital Library originally estimated that a re-import of their seven million volumes would take forty days [34]. HathiTrust addressed

the index scale issue by building their system upon the open-source Java project *Solr* [160]. Solr is an evolution of the popular Lucene Indexer, specifically designed to provide scaling capabilities. Lucene already handled large indexes well, using a technique called geometric merging [110] to allow indexes to grow in size without a linear increase in indexing time, while remaining on-line such that newly added documents are immediately searchable. Solr builds upon this by allowing a large index to be split up into a number of *shards* [164], each of which runs on a different machine (distributed processing). The system then provides clever mechanisms for merging the results from shards while, for example, performing a full-text search. It also helps users of large-scale systems by providing enriched searching techniques such as faceted and fuzzy searching.

The HathiTrust intends to eventually allow full-text index to around ten million books [61], but the current collection being processed (on five computers) originally took around forty days to process. Recent optimisations and care configuration of Solr have shortened this to ten days, but further reductions would require non-trivial changes to the way Solr distributes documents for indexing between shards. HathiTrust programmers have also noted the lack of hierarchical association within Solr indexes, and have instead looked at a two-tier system for implementing the idea of "search all books" and "search within a book". They are also investigating whether new features present in Solr version 4, such as Near Real-time Search (new documents become *online* faster) and Finite State Transducer-based terms indexing (allowing wildcard and fuzzy searching without linear cost to index size), will provide significant advantages at their collection's scale.

2.2.2 CADLIS

The China Academic Digital Library Information System [197] is a nation-wide academic library with over one thousand member libraries. The contents of the collection include bibliographic records from several academic libraries and publishers (including both Chinese-origin texts and Western journals), full-text theses and dissertations, academic web-pages, and several million multimedia objects preserving the history and knowledge of Chinese Indigenous peoples. The library uses a federated search model, with several different data sources being connected by an open infrastructure framework and communicating with each other using standards such as Open Archives Initiative Protocol for Metadata Harvesting (OAI-PMH) [141] and METS [52]. It also provides a software as a service architecture, allowing smaller libraries to outsource, at no ex-

tra cost, hosted applications such as library inter-loan and e-reserve systems. The project is centrally funded by the Chinese government and steered by the National Administrative Center for China Academic Library Information Systems.

This digital library provides an interesting case study for three main reasons:

- **Project Age**. The system has been in development for over 18 years. It was started in 1996, with the first prototype accessible to select institutions in 1998. Since then several more phases saw the patronage increase to 809 members by the end of 2006. This fact alone makes the system both mature and possibly one of the oldest digital libraries of this scale.
- **Scale of digitized content and users**. The library is large-scale not just in the number of documents—which number six to seven million full-text or similar objects, and thirty million plus bibliographic records—but in terms of the sheer disk size of its contents and number of users. The collection include many terabytes of digital audio and video as part of the Indigenous Peoples' collection, and is accessed by academics and students from a thousand institutions.
- **Architecture**. The current development acknowledges some of the difficulties encountered in physically maintaining a federated library of this scale, in terms of dedicated servers and bandwidth, and is looking towards cloud computing as a potential solution. As well as moving several of the collections into a new cloud-based framework [93], the project is looking to create even more cloud-based applications particularly focused on social networking.

While the project makes use of open standards and open frameworks, all of the software developed to date is proprietary and access to the collections are restricted to member institutions.

2.2.3 The Internet Archive

The Internet Archive is a non-profit founded in 1996 with the aim to provide an historical archive of as much contemporary digital media as possible. In terms of significance some consider the venture as important as the Library of Alexandria—the last comprehensive library of human knowledge [58].

The project is based at Carnegie Mellon University, created from donations from many existing projects, including a mirror of the 1.5 million books from the *Million Books Project*. Perhaps the most famous facet of The Internet Archive is its interactive web archive, *The Wayback Machine*.

Technically the project is a federation of several digital library systems, collectively containing an estimated 10 petabytes of data in 2012. In 2009 the popular WayBack Machine collection alone contained over 150+ billion captured web pages, amounting to some 3 petabytes of data and growing at a rate of 100 terabytes per day [134]. As of 2014 the main collections in the archive contain some 5.6 million books, 1.5 thousand films, 1.8 million audio recordings, and (quite uniquely) some $50,000$ pieces of software—shareware, freeware, demoware, documentation, and abandonware [98]. Further to this there are many more books and other media items in the community managed collections (2.5 million items), other collections (1.6 million items), and external collections (1.9 million items). These numbers make the scale of the endeavour quite clear.

In terms of hardware, the data processing is performed by 800+ low end commodity machines [99]. The Internet Archive has paid special attention to the cost and environmental impact of the data storage requirements of this collection, starting a spin-off company to manufacture custom hardware. A number of these custom-assembled *blade-style* computers is termed a Petabox as each stack was originally intended to provide one petabyte of storage—although the current version of the hardware achieves some 650 terabytes of storage [186]. They do so in an energy efficient way, consuming only 6 kilowatt-hours per stack, and forgoing typical computer cooling components, instead transferring the heat generated to the Internet Archive's building. Alternatively the Petaboxes can be assembled inside a railway or similar shipping crate, complete with generator and heat ducting, to provide a mobile storage site.

In terms of software, the engine underlying the WayBack Machine has been open-sourced and released as the Openwayback package.[2] The layout of The Internet Archive's website, combined with the way large collections of donated content are added ad-hoc, suggest the software will be some form of federated search engine rather than one with a central index and store. There is limited systematic access to the WayBack Machine available in JSON [51], Memento, and Web Archives (WARC) formats. WARC is open-standard format that captures one or more timestamped snapshots of a website within a

[2]https://github.com/iipc/openwayback/wiki

single container [105]. In 2011 The Internet Archive released their large-scale, open-source, web crawler software, named *Heritrix*, for public download [137].

2.2.4 US Library of Congress: Web Archive

The US Library of Congress Web Archive was initiated in 2000 as part of a pilot project to preserve at-risk digital web content [120]. The project, Mapping the Internet Electronic Resource Virtual Archive (MINERVA), contains a number of thematic collections, selectively built and moderated by subject specialists, from web-sites and other materials sourced from the Internet. The theme for each collection is focused on events that affected the United States. For example, the Library of Congress has created collections bringing together archived web-pages covering several US Presidential and Congressional elections, the wars in Iraq, and the events of September 11, 2001. When responding to criticism that it is repeating the work of other agencies, such as the Internet Archive (Section 2.2.3) and the International Internet Preservation Consortium (IIPC), the Library of Congress responds that each agency mentioned has a slightly different focus and collects a different slice of web information available [188]. Moreover, there are many relationships between the various archiving projects—for instance the Library of Congress is one of the founding members of IIPC [189].

A *capture* of an entire web site at a particular date is extracted from information donated by the commercial web-archiver *ALEXA* [107]. Several of these captures are then combined, chronologically, and wrapped in the WARC format to make a single web site *document* in the collection. Each document is further reviewed and enhanced with metadata in MODS format [127]. Apache Lucene is then used to index the metadata of each collection separately. These indexes contain only metadata so there is no full-text search. Primary access to the collections is via a custom web interface to the Lucene index. This interface supports simple searching, as well as the more advanced fielded search, over the metadata and provides a calendar-like browser for selecting which date's capture to view. It is worth noting that full-text search is available via the Library of Congress' separately indexed site-wide search and again in their online catalogue. In 2003 the site-wide search was provided by the proprietary *Inktomi* search engine [79].

Currently the Library of Congress collects around five terabytes per month, and has mirrored a total of 525 terabytes of materials [188], with each thematic collection ranging from half a terabyte to several terabytes in size. The

hardware configuration is surprisingly basic, with each collection served from one or more commodity PCs with multiple hard-drives installed to provide the necessary capacity [158]. Hard-drives are also used to transport the materials from archiving subcontractors to the Library of Congress and as the primary form of backup, raising questions of preservation in terms of equipment failure and data degradation [116].

2.2.5 US Library of Congress: Tweet Archive

The Library of Congress started an even more ambitious undertaking with a project to record and archive all of the messages sent on *Twitter* [151]. Referred to as a "social networking and microblogging service," Twitter has seen enormous growth since its launch in 2006 and is now one of the World's top ten most visited websites. The system allows the broadcast of short messages, called *tweets*, to a number of *followers* with registered users then being able to reply or *retweet* the messages. There is obvious value to researchers and historians alike contained within these messages. The messages themselves often capture a myriad of views about an event or events as they occur. They also provide further explicit metadata, such as the hash-tag mechanism providing good keyword or key-phrase candidates, as well as implicit metadata such as trend information and social networks.

However the project faces a significant scale challenge despite each tweet being only 140 characters long. While each message is relatively small there is an overwhelming number of them and the velocity at which tweets are produced is staggering. The Library of Congress has already been provided some 170 billion tweets amounting to well over one hundred terabytes of raw data [142]. Compounding the challenge, the rate of tweets has grown from 140 million per day to approximately half a billion per day in October 2012. While the Library of Congress is experienced in storing this scale of data using their well-established archival process—involving duplicate copies stored to tape and geographically separated—indexing of data at this velocity is unprecedented.

To address this the Library of Congress is now looking to partner with commercial providers of large-scale technologies, although they also state that "even the private sector has not yet implemented cost-effective commercial solutions because of the complexity and resource requirements of such a task." [142]. Suggested large-scale technologies include using Cloud computing, Hadoop, and NoSQL databases (Pivotal's Greenplum) [36]. The use of Greenplum is relevant to this thesis as it is a distributed database, using a shared-nothing

massively parallel processing architecture and provides column and row centred storage.

The project also raises important questions about how to actually interact with a collection of this size [133]. Even a single search on a subset of the current collection takes over a day to complete and produces millions of results with almost no way to rank or aggregate the results beyond simple counts.

2.2.6 Elephind

The *Elephind* project [25] provides a front-end search portal to the historical newspaper sections of eighteen institutional repositories, including the Chronicling America project, several US National Digital Newspaper Program awardees [87], and the National Libraries of Singapore, Australia, and New Zealand. All told, Elephind indexes 141 million items from 2.7 million newspapers in both full-text and metadata.

Behind the scenes, Elephind is powered by the *Veridian* system by DL Consulting [24]. Verdian, in turn, is built on a licensed version of Greenstone that has been extended so as to provide significantly more scale capability. The major changes to support this scale were the addition of Solr as the indexer and SQLite as the database. With these changes Veridian was able to scale to very large-scale digital library size as evidenced by the creation of several large historic newspaper sites, the largest providing a full-text and metadata index of more than three million newspaper pages.

Elephind makes an interesting case study for three main reasons:

- **The amount of full-text.** The amount of text indexed per document is relatively large compared to digital libraries constructed from traditional digital media such as books, the reason being twofold. Firstly, typically newspaper pages are physically larger than book pages and thus simply contain more text. Secondly, newspapers often rely on layout and placement of articles, sometimes spread across several pages, and it is important to capture and index this information too. This is reflected in the choice of storage format for the source documents, namely by combining the open-formats METS and ALTO [115]. The former provides a structured way of capturing the metadata about a newspaper, while the latter provides a similarly structured way of capturing newspaper physical dimensions and layout information.

- **Use of OCR text.** The full-text of the newspapers is typically extracted using optical character recognition resulting in the dirty text problem mentioned earlier in Section 2.2.1. Rather than the number of new unique words encountered when adding a document to the index plateauing out, as is typical with digital media restricted to a fixed lexicon, errors from optical character recognition continues to add new unique terms with each document [15]. Dealing with these new terms is more expensive in terms of indexing, retrieval, and in storage size of the index, compared with adding a reference to an previously encountered term.

- **Using Cloud technologies to support scale.** Although presented and accessible via a single interface, the collection is drawn from seven separate Solr indexes spread across two different servers. This is accomplished by making use of the new *SolrCloud* functionality available in Solr v4 [103]. SolrCloud provides automated distribution of indexes and inter-index communication including the propagation of searches and collation of results.

As mentioned earlier, the library is a cloud service, served from two Amazon Web Service virtual machines with file space provided by a number of Elastic Block Storage volumes. The collection is currently distributed across seven Solr cores of a maximum size of 50 gigabytes each. Whenever this maximum is exceeded a new core is established as experimentation has shown that larger cores begin to suffer performance issues.

While the collection has the potential for continued expansion and scale support by simply provisioning further virtual machines, there is still one bottleneck carried over from the Veridian system. Veridian—like Greenstone before it—uses a combination of index and database for its back-end storage and the current SQLite database does not easily support distribution. Thus the database is currently centralized on one machine and is nearing the limits of file size for the operating system. DL Consulting is investigating the use of the Amazon Relational Database web service as one solution as it compliments the use of other Amazon web services and should provide appropriate scalability.

2.2.7 Informedia

The *Informedia* Digital Library is an experimental library focusing on video media. The collection is hosted at Carnegie Mellon University and was initially created from an archive of documentary and educational programming

videos [193]. This archive includes contributions from WQED/Pittsburgh, Fairfax County (Virginia) Public Schools, and the Open University (United Kingdom).

This collection is of large scale both in terms of content size and in processing requirements when indexing. The collection features over a thousand hours of video and audio in a compressed MPEG format that adds up to approximately one terabyte of data [114]. Typically indexing and metadata extraction adds a further ninty gigabytes of data per month—so even the project's rate of growth could be considered in the very large-scale range—and it is these processes that are of direct interest to this research. The Informedia project utilises modern artificial intelligence techniques to enhance their collection. These techniques include: instance text extraction allowing full-text search of video media (uses Sphinx-II transcription software [94]), natural language processing for subject extraction, video segmenting and structuring, face detection, and cross-language information retrieval [47].

Text-base collections provide accurate and precise information retrieval by way of full-text search. In comparison video collections typically only provide browsing access sometimes augmented with some keyword/tag filtering, a much impoverished way to locate information. Browsing a collection of thousands of hours of videos is, in particular, impractical. Thus, though expensive to run in terms of computing power, these aforementioned video processing techniques are arguably essential in order to make a collection of this scale useful.

2.2.8 CiteSeer^X

CiteSeer^X is a digital library of scientific and academic papers focused on Computer Science and Information Technology [74]. The project was started in 1997 funded by grants from US National Science Foundation, NASA, and Microsoft Research. In 2008 the original site was replaced by CiteSeer^X, which was modelled on the original site but made use of a new search engine, *SeerSuite*, that contained modern search algorithms and optimised implementations [111]. CiteSeer was considered the first academic paper search engine and paved the way for later projects such as Google Scholar and Microsoft's Academic Search.

The project is relatively small compared to other case studies mentioned here, with around two million documents available for full-text search. In

regards to this research, CiteSeer^X remains interesting in that the document processing includes a number of data mining or artificial intelligence techniques and so incurs a greater processing overhead than other text-based digital libraries. Techniques include metadata extraction and automatic document linking by citations and references.

In terms of technology, public access to the data is via Amazon's cloud-based S^3 service [181] suggesting that the data is most likely stored in the cloud. The open-source SeerSuite software makes use of Solr and other Apache projects to provide a highly scalable digital library solution. The software is also modular allowing it to be used in testing new information retrieval techniques and algorithms [75]. The SeerSuite software is an open-source project and is available for public download.

2.2.9 Google Search Engine

While the label of Digital Library is debatable, there is no question of Google Web Search being the largest-scale of any of the case studies as they index an estimated forty billion unique web-pages [104]. Since Stanford University unveiled the prototype engine in 1998 [31], Google has risen to dominate the web search market to the point where the act of searching the web is known as *Googling*.

Google has pioneered or significantly advanced many of the solutions being applied today to manage the issues of scale. Google's *BigTable* [44] software moved away from tabular views of data, instead focusing on column and row based views to provide a database that allows massively parallel access to information. In an attempt to solve the problem of processing large data sets, researchers at Google developed a framework for parallel processing called *MapReduce* [54] with Amazon's *Hadoop* being the best known implementation of this framework.

When it comes to distributed computing, Google once again leads the field, with the search engine giant powered by approximately one million individual computers and with a framework in place to scale up to ten million machines [135]. This is at a scale well beyond the means of most libraries and institutions. Furthermore Google applies several optimisations and weight-based decisions when indexing to avoid having to index every word—the exact details of which remain a closely guarded secret.

Where Google's approach differs from that chosen in traditional libraries

(digital and otherwise) is that its self-professed goal[3] is to provide a search engine to all of the world's digital information without moderation or dissection. In contrast, the definition of Digital Library requires there to be moderation, often carried out by specific digital librarians, to ensure the collection is of a certain scope and level of veracity. Google understands this distinction and provides more constrained collections, such as Google Scholar [11, 76].

2.3 General Purpose Open-source Digital Libraries

The case studies shown above are all good examples of systems that address, in various ways, the issues common to very large-scale digital libraries. They include examples of scale solved by distribution (by way of federated networks) and/or by parallel processing. They are all bespoke solutions, however, designed to handle a specific document type or focus on a specific issue.

This thesis is interested in the scalability of more general digital libraries and so instead focuses on finding a solution applicable to general purpose open-source digital library software. There are many such digital library systems available, for example: the open-source projects CDS Invenio [38] (also known as CDS Ware), Evergreen ILS [30], EPrints [185], Fedora [168], and the popular commercial system CONTENTdm [9]. However, this research will limit itself to three chosen to be a representative cross-section of the digital library landscape as detailed in Section 2.1.1. This section introduces the three systems, namely: DSpace, Greenstone, and the Terrier IR Platform.

Table 2.2: Comparison of the three chosen general purpose digital library applications

Software	Data Store	Batch Importing	Parallel Processing
DSpace	Database	*bin/dspace import*	None
Greenstone	Database and Index	*import.pl*	None
Terrier IR	Index	*desktop_terrier.sh*	Hadoop

[3] `http://www.google.co.nz/about/company/`

2.3.1 DSpace

DSpace was released in November 2002 as a joint effort between developers from MIT and HP Labs [165]. It has since grown to be the most popular open-source institutional repository systems in the world, with over 1600 registered institutions using the software [187]. The system, implemented in Java, has an emphasis on capturing organizational structure, both in terms of the process of submitting and authorizing materials, and in directly mirroring the organization's structure between and within the collections by using hierarchical structure. This structured approach translates well to common library practices and thus DSpace has seen significant uptake. In brief, the system is flexible enough to capture many document types and provides a mechanism, *Media Filters*, allowing for extra handling for specific formats. These features allow complex and interesting collections to be created. For example, the eJamaica.org digital library [147] uses DSpace with the addition of video streaming and spatial mapping and browsing tools to improve accessibility.

The front-end of DSpace is provided by one or more web interfaces, with popular options being the built-in JavaServer Pages and XML interfaces. The latter is called *Manakin* and is the default interface. DSpace is customisable and providing new interfaces requires only beginner programming knowledge.

Behind the scenes each record in a DSpace collection is represented by one or more bitstreams, which are stored directly on the file system. These bitstreams are then tied together and positioned within the collection hierarchy by several pieces of metadata. This metadata is collected during the document submission process; in Manakin this involves a series of web-based forms each requiring several metadata fields be filled in. The submission process also includes a licensing page and is subject to moderation. The strict nature of this process once again fits in well with library practices. The metadata is then stored in either a PostGres [170] or Oracle [118] database, as configured by the installer.

DSpace provides a mechanism for processing or extracting further information from bitstreams by way of filters [57]. Each filter handles a specific document type, as defined by file extension, with the aim of processing this file in some way to extract metadata or new child bitstreams to be attached to the record. An implementation of this filter process that performs key frame extraction on video bitstreams is given in Section 3.4.4.

There is some research into DSpace's scaling performance. One experiment suggested a million document collection was possible but required careful con-

figuration to avoid exhausting available memory and other issues [136]. It also noted occasional unexplained slowdowns when indexing.

2.3.2 Greenstone

Greenstone [203] is a well established, open-source, digital library creation tool available for over a decade. The software was released in 2000 by the University of Waikato and makes use of novel indexing techniques [205]. One of the funders of the project was UNESCO as part of their goal of improving learning materials in developing countries. Due to that goal Greenstone was designed to be portable, capable of running on a wide range operating systems and computer specifications including quite old computers running operating systems such as Windows 3.1. The system was also easily customisable via plain text files, encouraging even novice users to contribute back to the project. For example, a common donation to the project is text string translations and thus the interface now includes 45 languages.

Greenstone is a popular digital library tool due to its ease of building, customising, and serving collections on desktop PC or laptop resources, and has become especially popular for courses training people in the use and creation of digital libraries. It has also been deployed at scale [15]. For instance, the Elephind and Veridian projects (Section 2.2.6) are built upon Greenstone and demonstrate the software's flexibility, in terms of underlying indexer and database, that can be combined with emerging technologies, such as SolrCloud, to create collections of millions or tens of millions of documents.

ReplayMe! is a customized version of Greenstone mentioned due to it being a practical example of a very large-scale digital library used later in the thesis [155]. The goal of ReplayMe! is to create a digital library from all the free-to-air television available in New Zealand. Automatically annotated with metadata extracted from a electronic program guide, the collection provides an archive of video allowing a user to revisit past favourites or an episode missed, much like commercial or pay-to-view systems like TiVo [179], but without the need to explicitly set a recording schedule. The system then provides a novel digital library browsing interface compatible with programmable remote control, as well as full-text search over the metadata, to allow useful access to the stored content. Another requirement of the project was that the system be a self-contained, self-sufficient digital library—for example, it does not require an Internet connection—and run on a commodity desktop computer.

The challenges of scale present in ReplayMe as a trade-off between veloc-

ity and volume. Accounting for the multiple channels of free-to-air television, there is approximately sixteen hours of raw video captured for every hour that passes. The file size of the video recorded is such that the prototype system can only store a two week window of programs within its six terabyte disk space. While converting these raw files to a compressed, web-streamable, format would both save disk space and allow users to access the content remotely, the current capture and indexing processes are purposely lightweight. There are two main reasons for this lightweight approach. The first is the constraint that the system run on typical personal computer, which (at the time of the project) limited it to single processor architectures. The second is that experimentation showed that current video conversion techniques when run on a commodity personal computer take roughly real-time to perform. It might have proven feasible to encode one television channel but not several at once.

The research in this thesis explores whether parallel processing could be leveraged to handle real-time video processing of this scale. Moreover, a video collection can benefit from extra video processing methods to extract and present valuable information, similar to those investigated by the Informedia project in Section 2.2.7. A future version of ReplayMe! might offload processing work such as video conversion and key frame extraction to a shared, parallel-processing enabled cluster.

2.3.3 Terrier IR Platform

Terrier—a syllabic abbreviation of TERabyte RetrIEveR—is an open-source digital library specifically built for large-scale, high performance, information retrieval experiments [143]. It was developed in 2000 by the University of Glasgow in Scotland as a test-bed platform, and supports a wide range of information retrieval techniques from the traditional TF-IDF method [97] through to modern versions of probabilistic techniques such as Okapi BM25 [153] and Divergence From Random [6] implementations. Similar to Greenstone, Terrier supports indexing of various document types via plugins added to the indexing process. This plugin layer is the second of four steps in the Terrier indexing process, with the complete process being:

- **Collection**. The documents are read in from several different sources such as file systems, web-harvesters, or other servers.
- **Plugin**. The documents are then parsed using a number of different user-defined plugins with the typical output being a stream of tuples

made of a single term, its position in the document, and any field in which the term occurs.

- **Pipeline**. Parsed data can then be (optionally) transformed by a number of filters, for instance, stemming and stop-word removal.

- **Index**. Finally, the indexed data is written to four different data files, namely: the lexicon, an inverted index, a document index, and a direct index.

The use of these four index structures, along with an additional weighting layer that can be applied on a per term or per document basis, provides the flexibility required to test the wide range of information retrieval techniques mentioned.

As well as supporting this range of techniques, Terrier was the first open-source digital library platform to support experimentation in parallel processing including Hadoop integration, thus providing several MapReduce-based indexing approaches [128]. The map phase fits well with the first three stages of Terrier indexing (collection, plugin and pipeline), while the reduce phase generates the data structures in a serial fashion as found in the index stage. Limited distributed computer support is also available, as multiple reduce phases may be configured, each partitioning off some part of the index to produce multiple separate indexes.

Terrier is purposely modular, with many of the stages in the indexing process, the weighting layer, and the information retrieval layer all being configurable and with a selection of existing plugins to choose from. Beyond that, the platform benefits from being open-source and written in Java and thus provides a programming interface to allow the development of further collection sources, parsing plugins, pipeline filters, weighting models, and information retrieval implementations.

Terrier was chosen for this research as, due to its focus on information retrieval techniques applied to the data structures of the index, it is purposely lightweight on the storage and retrieval of the original document materials. When displaying document content the information is reconstructed from the data found in the index, as compared with retrieving it from some accompanying database of information. In this regard Terrier can be seen as being at the index-only end of the digital library landscape introduced earlier in Section 2.1.

2.4 Discussion

This chapter has described the digital library environment, with specific emphasis on issues of scale and how they relate to the technologies underlying digital library implementations. It then provided case studies on a number of very large-scale digital libraries, exploring the different ways in which scale is encountered. While scale due to number of documents is obvious, there are less obvious factors to also consider, such as velocity of new data or application of data mining techniques to content beyond the realm of manual management. In particular, this latter feature—describing an increasing complexity in importing and processing the documents within the collection—will shortly prove to be a key element to the model created through this research.

Following this, the chapter introduced the three open-source, general purpose digital library software solutions selected for exploration in this research. The three were chosen so as to represent the range of back-end data stores, from predominately database-backed through to predominately index-backed. Despite having distinct storage approaches, the three digital libraries exhibit several common features:

- They all support the batch import of documents; a fact that will prove important during the implementation detailed in Chapter 4.
- They all allow the files to be imported to be controlled by a file list of some form; allowing a batch import to be split into smaller parts to be distributed across parallel computers.
- They all provide a mechanism for specifying how a particular file format should be processed for inclusion into the collection; allowing this research to explore media types that incur more expensive processing costs than plain text.

By leveraging these three facts it is possible to rearrange the import process flows of each of these libraries so as to integrate a parallel processing framework in a broadly similar fashion. Once implemented, this research can then explore the interaction between certain scale factors and parallel processing, in order to gather information for the predictive model.

Of the possible scale factors identified in the case studies, this research will focus on two in particular, namely: number of documents, and complexity of processing. The former is the most common challenge encountered in very

large-scale collections, is an obvious factor when it comes to modelling digital library processing costs, and is directly answerable by parallel processing. For the latter, parallel processing again promises to mitigate the increasing amount of work done when importing and processing documents. As digital libraries mature from pure text into collections of complex multimedia such as audio and video files, the digital library software must also evolve by way of advanced processing techniques, to capture more metadata, and novel searching techniques, to provide meaningful access to these materials. Such support comes at the expense of increasing processing complexity and thus longer import and response times. Fortunately, these costs are often naturally partitionable and thus can be readily divided across multiple computers should the digital library support parallel processing.

Extending the three digital libraries to support parallel processing essentially involves two steps: restructuring the import process so as to suit the flow of a parallel processing framework, and then altering or replacing any underlying technologies within the digital library that do not support parallel reads and writes. The next chapter will introduce the two frameworks this research will utilise—as part of a wider discussion on parallel processing—thus laying the groundwork for Chapter 4, which will bring all these ideas together, providing implementation details for the three digital libraries when combined with the two parallel processing frameworks.

Chapter 3

Parallel Processing

> Ten people can pick cotton ten times as fast as one person because the work is almost perfectly partitionable, requiring little communication or coordination. But nine women can't have a baby any faster than one woman can because the work is not partitionable.
>
> — *McConnell, S. "Brook's Law Repealed", 1999*

Parallel processing has been mooted as a possible solution to the scale issues mentioned in Chapter 2. The basic idea behind parallel processing is to break a lengthy or computationally difficult problem down into smaller parts that can be processed simultaneously so as to arrive at the conclusion quicker. This paradigm can be applied at many of the layers that make up modern computing, from the underlying hardware level, through bit, instruction, and data levels, all the way up to software and higher-level abstractions. For example, concurrency, as an instruction-level technique, involves the division of several processing tasks into time slices, which are then interspersed on a single processor allowing it to multi-task or to emulate several processes running at once. At a hardware level, parallel processing has typically involved increasing the number of transistors in the central processing unit in order to allow it to perform more calculations at once and with larger numbers. But as we reach the physical limits of how fast we can push instructions through a single core, manufacturers are turning to creating multi-core processors as a way of increasing processing power. Moore's Law, which predicts the doubling of transistors in a computer every two years [138], is now met by packing more and more processing cores into a single computer. At present four cores processors are the norm (quad cores), with multiples thereof just on the horizon.

Another hardware approach to parallel processing is distributed computing.

Rather than running on a single computer, distributed computing achieves the benefits of parallel processing by spreading its workload over a number of computers. This requires either the underlying operating system to be aware of multiple computers or for the software itself to support distribution in some form.

Some examples of operating systems providing distributed support include:

- Beowulf cluster configured operating systems [19] such as the open-source Rocks project [144]
- cloud computing platforms such as Amazon's Web Services [95] or Microsoft's Azure [199]

Examples of software supporting distributed computing include:

- the shard technology of Solr [160],
- large-scale databases with built-in support for distribution such as IBM's DB2 [126] and MySQL [101] (via the Cluster project [154])

Again, a good parallel processing framework can abstract away the actual topography of underlying hardware making the developed parallel processing code similar regardless of whether it occurs on a single processor machine, a multi-core machine, distributed computing over a cluster of several machines, or hundreds of machines in the cloud. Indeed the terms parallel processing and distributed processing are fairly interchangeable these days with the boundaries between subtle and often blurred.

However, the parallel processing potential of hardware needs to be matched with software specifically designed to leverage this potential. This mirrors the growth in digital library size requiring specific software to manage it. Well-designed parallel processing software provides a means to increase computing efficiency and decrease processing time by splitting a processing task between multiple cores—cores that might otherwise sit idle. Developing or re-factoring software to take advantage of parallel processing is in general a difficult manual task, although modern programming languages and frameworks have significantly decreased this cost. Some modern compilers even offer mechanisms for automatically running applicable code segments in parallel [45], for example some looping constructs where no iteration is dependant on another.

A third reason to make use of mature, third-party, parallel processing frameworks is to mitigate a serious challenge in parallel programming: that of synchronisation. Synchronisation is used to ensure that processing actions occur in the order intended regardless of if they are executed on separate processors or even machines. For example, in a database system being accessed by multiple parallel threads, there needs to be a mechanism to ensure that record additions are carried out before any actions that edit or delete those same records. In other words, synchronisation is used to maintain some critical sequential ordering in a systems that are asynchronous. Failure to address this issue when programming leaves the potential for difficult to locate and resolve problems, such as a deadlock between two processes competing for the same resource but now stopped indefinitely with each waiting for the other to proceed. Such is the difficulty in locating them that such bugs are sometimes referred to as "Heisenbugs"; attempts to observe them using debugging tools often change the program timing and thus obscure the problem.

Closely related to synchronisation is the consideration of the ratio of processing work that can be done in parallel—and thus benefits from parallel processing—versus that which must occur sequentially. Significant improvements can be achieved in those cases where there is little or no dependency between processing tasks. By contrast, some processing tasks may be highly dependent on each other to the point of there being only a single, linear path through the tasks and hence precluding any benefit from parallel processing; the term for this situation being "inherently sequential". Typically software contains instances of both cases. The balance between the two cases is captured in Amdahl's Law, used to calculate the improvement to an overall system when only part of the system is improved [8].

This law, however, assumes the amount of work will remain fixed. In practice, as programmers encounter faster and more efficient computers, they tend to process larger and more complex challenges in the same amount of time. This second trade-off was described by Gustafson's Law [81]. The perfect example of this is computer start up times. Despite significant advances in processing and memory speeds, starting a computer still takes a reasonable amount of time—reasonable in that this amount of time is acceptable to most users. A more complex operating system and many more drivers are loaded in the same amount time that used to only allow for a basic operating system and a few drivers. Two of the three questions this research seeks to address with the developed model are mirrored by the aforementioned laws, leaving only the question concerned with trying to predict the optimal hardware given fixed time and processing cost.

Turning now to specific examples of parallel processing being applied to digital libraries, Stanfill [167] proposed a number of ways to leverage the power of parallel processing that is applicable to the information retrieval component of a digital library system on indices in the order of terabytes. Examples such as the China Academic Digital Library Information System [197] and HathiTrust Digital Library [207] show how existing projects can leverage distributed processing.

The remainder of this chapter will present features of parallel processing important to the scope of this research. It will introduce the two specific parallel processing frameworks investigated by this research, while also explaining how the implementation will use a Single Program, Multiple Data-stream approach (SPMD). It will briefly discuss earlier work in parallel processing within digital libraries, and in particular grid-based approaches. This chapter will then discuss some of the challenges encountered common to all three parallel processing implementations outlined in Chapter 4.

3.1 Shared and Share Nothing Architectures

Parallel processing architectures are typically implemented across a spectrum, classified upon which memory and data resources are being shared between process threads and to what extent the sharing occurs. At one extreme, there are *shared* approaches where processing threads share memory and data access, while at the other extreme, *share nothing* approaches where each processing thread maintains their own copy of memory and data.

Shared approaches incur little communication overhead, as data can be stored and accessed via the shared resources, but requires careful thought to synchronisation of resource access and is highly dependent on the hardware configuration. This configuration is most suited for use on a single, multi-core computer. Early examples of parallel processing on shared resource hardware include early super-computers. Modern computers now leverage this same architecture on a lesser scale. Traditionally, the term parallel processing was more readily associated with shared approaches although that distinction is far less obvious given modern architectures.

Share nothing approaches pass data back and forth between processing threads as if they are individual devices in a network, even when they exist on the same machine. The goal of this approach to avoid a single bottleneck or point of contention within the distributed process. Share nothing

approaches were traditionally associated with the term distributed processing. While there are many examples of share nothing architectures, such as the venerable Teradata Database Computer [178], arguably the best known is the Message Passing Interface (MPI) in its various incarnations. The outcome from a 1992 workshop on standardising communications in parallel environments [194], and drawing upon lessons learned in supercomputing and the earlier Parallel Virtual Machine architecture [20], MPI is arguably the predominant standard protocol for share nothing parallel architectures. There are several revisions of the standard available, with Version 1 being the best established and popular, Version 2 providing advanced features such as parallel input/output and remote memory manipulation (reminiscent of shared approaches), and the recently released Version 3 focusing on high fault tolerance.

The benefits of MPI include:

- **Language independence**. Each node in an MPI cluster need only adhere to the protocol as defined in the standard and may be implemented in any programming language.
- **Simplified logic around data**. There is no shared data and each thread maintains its own copy of the data.
- **Understood models of communication**. Framework is directly comparable with well-understood computer science fields such as networking.
- **Hardware flexibility**. Supports many hardware configurations and network topographies.
- **Communication flexibility**. Communication available in both point-to-point and broadcast messaging approaches.
- **Hardware independence**. Allows implementation to be run on a single-core computer, a multi-core computer, multiple computers (grid or cluster), or even a cloud-based network without significant changes.

The main drawback of share nothing approaches is the decrease in efficiency due to the significant cost of communication overhead and issues around fault tolerance. Note that this is mostly a drawback of particular implementations rather than something intrinsic to share nothing architectures.

Another approach to consider is MapReduce, due to its popularity and use in this research. This provides a hybrid model that sits somewhere in the

middle of the spectrum, in that each processing thread has its own memory space but effort is made to share data—by way of distributed file systems—in order to leverage the benefits of *data locality*. This approach also makes use of message passing and task schedulers to provide high fault tolerance. It can be hard to distinguish this approaches from recent MPI implementations and the lines are being further blurred by projects to implement MPI on top of MapReduce [23] and vice-versa [146].

3.2 Frameworks

In order to add parallel processing functionality to the three chosen digital libraries, this research incorporates existing parallel processing frameworks into the batch import processes of the aforementioned libraries. The scheduling and monitoring of parallel processes is a complex and an easily fallible problem, as discussed above, and so the risk of issues was minimized by selecting mature and popular frameworks. Two quite different parallel processing frameworks are considered and integrated, namely *Open MPI* and *Hadoop*.

Open MPI is an open source implementation of the MPI framework explained above, encapsulating all of the features from Versions 1 and 2 of the standard and some of the features of Version 3 [69]. It is a mature piece of software—having been under development and refinement since 2004—and brings together knowledge gained in several earlier, successful, message passing implementations: PACX-MPI [70], LAM/MPI [33], LA-MPI [13], FT-MPI [60], and Sun HPC ClusterTools 6 [180]. Open MPI is designed for high performance, being used in several of the fastest computers in the world's Top500 SuperComputers [166, 139].

Open MPI is well suited for a SPMD model of parallel processing [12]. This approach divides the data across multiple compute-nodes each of which applies the same program/processing; all compute nodes are assumed to be homogeneous in terms of capability and program availability. In our implementation a single *master* thread splits the input to the process into tasks, each of which can then be processed independently and in parallel by one or more *worker* threads. The workers may start and end at any time, they are not synchronised nor do they communicate with each other. Instead, the master thread provides a central communication point and is solely responsible for scheduling: assigning tasks to waiting workers, gathering task results from completed workers, and exiting the program once all tasks have been processed. Open

MPI denotes each processing thread with a rank, where rank zero is the master thread, and information is passed back-and-forth between threads using derived data-types so as not to have issues with minor differences in standard types.

The second parallel processing framework selected was Apache's Hadoop project [198]. This is an open-source, Java implementation of the MapReduce architecture originally developed by Google [54] and loosely based upon the *map* and *reduce* functions found in functional programming. Hadoop's implementation of the MapReduce process is similar to the SPMD model, but has three phases and features additional rules that allow for more complex process flows. The phases are purely logical in that there is no synchronisation; a phase begins as soon as its first input is available and runs at the same time as other phases. The phases are:

- **Map**. This involves transforming a number of key/value pairs into intermediate key/value pairs. Practically this involves a master thread reading in the input file, splitting it into smaller lists based upon a combination of InputFormat and InputSplit configuration, and then delegating multiple worker nodes to apply the user defined Mapper. One change from the SPMD model is that workers may, in-turn, delegate work resulting in a nested structure.

- **Shuffle**. Wherein the key/value pairs output by each worker node are simultaneously sorted and then dispatched to the appropriate Reducer. At present the shuffle phase is arguably an inseparable part of the work done by a reduce node, with a one-to-one relation, but research suggests that it could be separated to improve performance [80].

- **Reduce**. Gathers a number of key/value pairs with the same key and combines them in some fashion to produce the desired output from the process. Once again deviating from the traditional SPMD model, there may be multiple reducers running at once, each acting upon a distinct key as defined by a Partitioner and writing to a separate output file.

Hadoop, as well as having task splitting, scheduling, and result aggregation functions similar to Open MPI, provides a number of valuable features for parallel processing including: fault detection and tolerance, built in timing reports, and integration of a distributed file system.

3.3 Federated Search Digital Libraries

As mentioned in Chapter 2 there are already several large-scale digital library projects attempting to leverage parallel processing ideas in order to address the issue of scale. This research acknowledges there are other ways multiple computers can be applied to tackle the issues of scale, such as the EuropeanaLocal federated search library [130].

Due to their prevalence and popularity, of particular note are grid-based approaches to building digital libraries. A grid is a means of coordinating a network of separate computing resources, allowing both information and processes to be distributed amongst them. This approach lends itself well to digital libraries, permitting the typical creators of digital library projects—traditional libraries and institutions—to pool their resources and minimise repetition of tasks. This is especially useful where these tasks might be expensive or difficult, such as the digitisation of historic documents. The coordination in grids relies upon well-defined inter-node communication using well-established open protocols to ensure the consistency and mergeability of results sourced from different nodes [102]. In information retrieval sciences, a typically used protocol is the venerable Z39.50 [121]. Access to a grid-based system is typically through some gateway interface that allows the scheduling of tasks and the combination of results. In the digital library realm, such interfaces are often referred to as federated search digital libraries. Also common to grids is the idea of 'virtualisation' of services to make them somewhat agnostic of location, hardware, and user/organisation.

Examples of grid-based digital libraries include:

- The Cheshire3 digital library [157] uses an object-oriented approach to create a grid-based digital library. Objects not only represent data and storage–such as a document, search result set, or user information—but also processes such as parsing, transforming, and extracting. There are also two special objects: one allows transformation between communication/transport protocols, while the second encapsulates a workflow. A workflow is some user-defined sequence of other processes to be performed on the data and is expressed in XML. The system uses a master compute node to distribute the actual processing work to an appropriate 'slave' node. Two implementations were developed: one using MPI as in this research, and the other using an approach called Parallel Virtual Machines (PVM) [20]. The applicability of the MPI approach was demonstrated by building an example collection created from 15 million

abstracts from the UK MedLine collection and utilising the 256 computer TeraGrid.

- DILIGENT [43] is a digital library built upon the European Grid Infrastructure (EGEE/EGI) [71] and adhering to the DELOS reference model [41]. The goal was to leverage existing materials while distributing the workload of creating new collections to libraries and institutions throughout Europe and then utilise a federated search through a common gateway to combine the results. A test-bed framework was developed and several example collections creating, for example the ImpECt Earth Sciences scenario [39].

- The research on a Hybrid Distributed Grid [140] proposes system combining a small number of dedicated commodity computers (traditional grid) with a dynamic number of computers 'scavenged' together from otherwise under-utilised machines in their downtime. An example of the latter might be office computers left on overnight. Several drawbacks of scavenger grids—the heterogeneous nature of the machines and the fact they could leave the network before their scheduled work is processed—are addressed by leveraging third-party software with high reliability: Lucene as the indexer [82], Condor Job Scheduler for task scheduling [117], and Storage Resource Broker as the distributed file system [18]. An experiment, conducted with thirteen dedicated machines and a further sixty-six machines allocated dynamically, showed that under heavy processing workloads the hybrid scavenger grid was more cost-effective (per processing core) than the equivalent high-end multi-core computer.

- Greenstone3 [14]—the next evolution of the Greenstone library explored in this research—provides support for grid-based digital libraries through a SOAP-based message passing interface [27]. Much like the Cheshire3 software, a request to Greenstone3 can contain a XML 'recipe' of documents/data and the processes to be run on them. For each step in the recipe the system determines which of its registered modules can process that step, potentially running on other nodes in the grid.

In relation to the the work presented in this research, grids are a closer match to the Multiple-Program, Multiple Data-stream (MPMD) approach to parallel processing [66], where each compute node may offer different processing programs and capabilities requiring some higher level manager to allocate the workload to the best node to run a process on. In contrast—and as dis-

cussed in Section 3.2—this research's implementations follow the SPMD approach, where nodes are expected to be homogeneous in terms of capabilities.

3.4 Challenges

This section will discuss the challenges faced, common to all digital library and framework implementations, in order to undertake the research reported on in this thesis. It will discuss:

- tools specifically created during this research in order to accurately monitor the parallel import process,
- the aspects that influenced the choice of underlying database (for those digital libraries requiring such),
- file system considerations,
- cluster configurations,
- an exemplar heavy processing task by way of video processing, and
- the difficulties in the creation of large-scale corpora.

3.4.1 Measuring Time

One issue with parallel processing that may not be immediately apparent is the difficulty in actually measuring the time taken for processing tasks to complete. When measuring time on a computer there are at least two different measurements that we might consider recording: elapsed time, and process time.

Elapsed time is the measurement that parallel processing aims to decrease. This 'real time' could easily be measured from a wall clock or stopwatch, and thus is straightforward to measure. However, when measured on a computer it is susceptible to interference from other tasks running on the computer resources utilised. This problem is readily apparent on modern computers capable of multitasking several processes at once. Controlling elapsed time is further exacerbated when attempting to spread the workload over multiple computers. The subtle interactions between processes and machines become

```
************************************************
Import complete
************************************************
* 3 documents were considered for processing
* 3 were processed and included in the collection
* Waiting for TDBServer to exit... Done!
real 203.41
user 199.18
sys 0.48
```

Listing 3.1: Serial processing time

```
************************************************
Parallel import complete
************************************************
* 3 documents were considered for processing
* 3 were processed and included in the collection
* Ended: 1396310026.753448
* Review gsimport-w*.log files for any warnings
* Waiting for TDBServer to exit... Done!
real 186.41
user 328.61
sys 162.80
```

Listing 3.2: Parallel processing time

too chaotic to exactly measure, and random events—such as virus scans—can have a significant effect on elapsed time, potentially making it harder to extract stable results from experimental runs.

In contrast, process time is an approximation of the time spent by processors on the task. This is the cumulative time spent by each processing core executing the instructions that make up the processing task. It is an approximation calculated from the number of time slices allocated to a task multiplied by the clock speed of the underlying hardware, rather than an exact measurement. Because of this, and the fact that process time is accumulated separately for each processing task, it is immune to external influence from other processing tasks on the same computer. If another busy program causes the processing task to wait until resources are free, this time is not included in the process time. Thus measuring process time provides far more stable results during testing on modern computers.

Consider the illustrations Figure 3.1 and Figure 3.2, each showing the results of using the Unix program `time` to measure the time taken for a process to complete. Figure 3.1 shows the typical timing result displayed for a serial Greenstone import process. The import utilised a single processing thread

and thus the total elapsed time measurement, referred to as "real" time in the aforementioned Figures, can be assumed to be the sum of:

- 199 seconds of "user" time, synonymous with process time, which is time spent by the processor executing the Greenstone import code's instructions,
- $<$ 1 seconds of "sys" time spent by the processor executing operating system level instructions such as file transfer commands, and
- approximately 3 seconds time spent where the processor was not, itself, actively executing any instructions associated with the import process. This could be due to waiting for resources or by the influence of other running processes.

This straightforward summation becomes inadequate when trying to describe the results shown in Figure 3.2. These results show the same collection being import using the parallel processing-enabled Greenstone implemented during this research and utilising four processing threads. The total elapsed time has decreased—as expected—but now the process time supposedly took longer than the elapsed time. This result may initially seem strange, but remember that the process time is the cumulative time spent executing instructions on a processor and the task is now executing on four processors—each of which will have a share of the import task to process plus some extra processing cost incurred by initialisation and communication. There is also the subtle effect of Amdahl's Law to consider, that some parts of the process may not be run in parallel, and thus processing threads become blocked waiting on the serial parts to complete. As a case in point, the parallel processing example shown uses the Open MPI framework in which the master thread waits for the worker threads to finish using a system level synchronisation function, as evidenced by the drastic increase in recorded system time.

This research initially used the Linux `time` tool to measure timings. This program can be prepended to a system command in order to capture these details. As experimentation progressed issues arose when attempting to measure time in child processes started by top-level applications. For example, the time taken for calls to `HandBrakeCLI` for video processing were not reported in the results shown by the time tool. A second measurement solution, using regular polling of the Linux `ps` tool, provided a more accurate measure of time spent across both parent and child processes. This tool could also be configured to list all the running processes matching some wildcard pattern. However,

```
================= Running test: demo#1 ================

* Note: to gracefully exit create a file: exit.now
* Started: Wed Apr 9 11:45:03 2014
* Deleting existing archives directory... Done
* Synching file system... Done
* Dropping memory disk cache... Done
* Waiting for drop_caches to complete... 3 2 1 Done
* Running baseline collection import... Done
  - Duration:    4.272298 seconds
  - System Calls Breakdown:
    - I/O Duration: 2.645311 seconds
    - I/O Percent: 61.9%
    - Other Duratn: 0.091382 seconds
    - Other Percnt: 2.1%
  - See 'debug-baseline.tsv' for raw data
* Deleting existing archives directory... Done
* Synching file system... Done
* Dropping memory disk cache... Done
* Waiting for drop_caches to complete... 3 2 1 Done
* Running import and tracking I/O metrics... Done
  - Import?      Completed
  - Found:       11 documents
  - Processed:   11 documents
  - Duration:    10.218800 seconds
  - System Calls Breakdown:
    - I/O Duration: 4.176242 seconds
    - I/O Percent: 40.9%
    - Other Duratn: 0.140113 seconds
    - Other Percnt: 1.4%
  - See 'import.log' for Greenstone import details
  - See 'strace.out' for STrace details
  - See 'debug-import.tsv' for raw data

================== Import Results =====================

 Import Duration: 5.946502 seconds
 System Calls Breakdown:
 - I/O Duration: 1.530931 seconds
 - I/O Percent: 25.7%
 - Other Duratn: 0.048731 seconds
 - Other Percnt: 0.8%
```

Listing 3.3: An example of the strace timing tool's output

its timing granularity was limited to whatever was used as the polling time and was not suitable for measuring processes spread over multiple computers. Finally, as it became apparent that the ratio of processor to file transfer time would be the major determining factor in the model, a timing solution using low level system trace logs was developed. An existing application that analysed strace output was located [200], then extended to support parallel Greenstone. An example of the output from this tool is shown in Figure 3.3. The Unix program `strace` produces an exhaustive listing of system calls, with each entry providing enough information to determine what processing thread it belonged to and whether it was a file transfer instruction. Care was taken while developing this tool to avoid influencing timing results via actions such as I/O or system-level process management. An example of the former is delaying the writing of logs until such an action would not influence timing results, while an example of the latter is ensuring correct accumulation of total time spent by instructions that started execution on one processing core, were suspended, and then were later resumed on a different core.

In terms of running a battery of import tests, the early results, detailed in Chapter 5, were initially disappointing. Processing time was only being halved despite utilising eight or more cores. Several profiling tools were then used to analyse the import process to determine what factors were limiting the performance. An example tool used was the Perl-specific New York Times Profiler [32]. This quickly revealed a number of bottlenecks, many of which were subsequently mitigated by careful selection of configuration options. Even with these fixes in place, performance was still not as good as expected. Real world data suggests that the optimal number of parallel threads for a compute-bound load, in order to maximize processor utilisation, occurs at P or $P + 1$ (where P is number of computer processors) [77]. Under the assumption that importing might be exhibiting file transfer bounds rather than processing ones, further tests were designed to determine if I/O was the remaining limiting factor in parallel importing.

While modern operating systems offer significant benefits to typical users, such as journalling, this made attempting to measure precise processing times far more challenging during the execution of this research. During this early testing of the parallel processing code it was thus prudent to have as much control over processing and file transfer events as possible. To this end several different techniques were used in order to systematically control parts of the influencing factors from the operating system environment, including:

- The use of random access memory drives in order to avoid the costs of

file transfer to and from the disk;

- Changes to operating system level features such as flushing disk cache and scheduling explicit calls to disk synchronization in order to reset the file cache between tests; and
- Various versions of processing plugins that ranged from doing nothing at all—in order to measure the base cost of just initializing the Digital Library being tested—through those that performed no file transfer by storing data in memory, to those that performed only file transfer.

Without these many precautions, optimisations performed by the underlying file and operating system were found to have a dramatic and unpredictable affects on program performance, sometimes reducing processing times by 50% or more and at other times producing dramatic spikes in processing load.

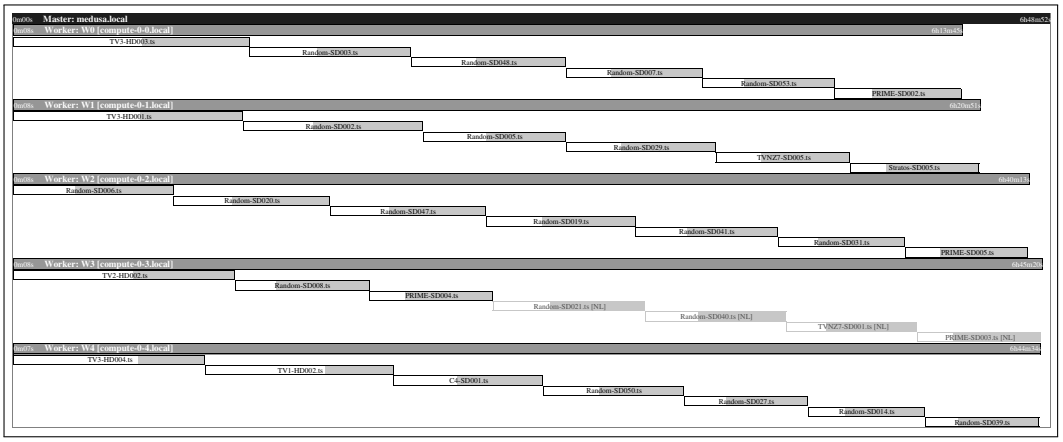


Figure 3.1: Preview of the Gantt-like timing visualisation of a parallel batch import of video documents

A final challenge is making sense of the various timings provided by the tools explained above. A parallel processing architecture invites subtle and complex interactions between running threads that can be difficult to unravel when considering a spreadsheet of numbers. In order to help visualize the results we drew upon the field of project management where complex projects must schedule and measure the timing of many tasks. These tasks may be dependent on the timing of pre- or co-requisite tasks while independent of the timing of others. A popular tool to visualise this interaction is the Gantt Chart [72]. Figure 3.1 presents a indicative preview of the Gantt-like chart—complete versions of selected charts can be found in Appendix B. The chart effectively communicates important information such as:

- The total elapsed time is indicated by the summary or *rolled up* task bar.

- Each worker thread—be it a processor core in a multi-core computer or compute node in a cluster—is represented with a summary task bar indicating: a unique name, start time relative to master thread, and total elapsed time for that worker.
- Each processing task is labelled with the document/file being processed.
- Vertical cascading shows the ordering of tasks over time.
- Shading of the task indicates the time spent processing in grey, while a white background indicates file input/output.
- Specific to processing on a cluster, a solid task border indicates data locality—that the process was performed on the same compute node as the data resided—while a dotted line and the suffix "[NL]" indicates data was not local and had to be transferred to the compute node.

For example, it is interesting to note that in the given Gantt chart all the ten minute video files segments take approximately the same amount of time to process (shaded region) regardless of their video quality and resulting difference in file size. The file size is evidenced by the unshaded region in the task bar and is significantly longer for high definition video segments. This counter intuitive results was investigated and shown to be caused by the way video files were processed into web-streamable format.

3.4.2 Databases

Another challenge encountered, one that is specific to Greenstone, was the limitations on parallel processing presented by the choice of underlying database. The default database in Greenstone is GNU's flat-file database system GDBM. This venerable open source version of the Unix DBM application is both mature, having been released in 2001, and also very fast, due to a simple key to value approach to managing data. The speed arises from the efficient dynamic hashing algorithm [161] applied to the key. However, typical implementations of this algorithm—as found in DBM and GDBM—preclude concurrent or multiple writers updating the database and thus are problematic for parallel processing. It is important to understand that—given that each connection to a GDBM database must be established as either a reader or a writer—a writer connection takes an exclusive lock over the database preventing any further readers or writers from being created.

Looking at the way GDBM is currently used in Greenstone, the import process relies heavily upon two databases; `archivesinf-doc` that records the associated files and metadata for each document in the collection, and `archiveinf-src` that remembers where each document and associated file was found on the file system allowing for later incremental builds. The default program flow already contained the potential for either of these databases to be accessed by multiple writers, albeit in a non-concurrent fashion. For instance, a high-level plugin uses a database to keep track of the directories and files to process, while lower level plugins can add associated file details to that same database. Thus database connections are short-lived by design; they are opened, acted upon, and then closed in as short a time as possible. Unfortunately there is an overhead cost each time a database connection is opened, something that happens several times per document, and thus the implemented database access is comparatively expensive compared to an approach that used a one-off (atomic) database connection at start-up. This cost may be exacerbated by parallel processing due to the increasing likelihood of multiple threads having to wait in order to secure a write lock over key database files.

Greenstone interacts with the GDBM database via a suite of tools collectively called GDBMEdit, with the list of tools being:

- **db2txt** - extracts the entirety of a GDBM database and represents it as a sequence of text. This text follows a simple predefined format allowing it to be easily parsed with Greenstone's Perl scripts.
- **gdbmdel** - allows a user to remove a single GDBM key-value pair from the database by providing the specific key string.
- **gdbmget** - allows a user to retrieve the value associated with the specific key string provided.
- **gdbmkeys** - returns a list of all the key strings in the database.
- **gdbmset** - allows the addition of a key-value pair to the database. If the specified key already exists in the database, then the associated value is updated instead.
- **txt2db** - takes a block of text as input, in the same simple pre-defined format used in *db2txt*, and transforms it into a GDBM database, appending to or overwriting any existing database in the same location as specified by configuration parameters.

The initial solution explored was to implement a version of GDBMEdit executables that supported simple synchronisation by way of file locking. Renamed GDBML, these versions of the tools would correctly wait for one writer to finish and release the lock before a second connection could be established. This addressed the first problem, in that multiple threads could attempt to write at once (albeit each in turn). However this implementation was slow, reducing the access to the database to a blocking serial process and thus effectively cancelling out any improvement in speed resulting from parallel processing. Due to the overhead of file locking this database often proved to be slower than simple serial processing as evidenced in Figure 5.12 in Chapter 5. Moreover this solution does nothing to address the issue of the overhead of short-lived database connections.

The second solution explored was to locate a DBM-compatible database that allowed for multiple writers. After considering several possibilities, such as Kyoto Cabinet [90], GDBM for Cilk/Cheerio [55], and the commercial Berkeley DB with Concurrent Data Store (CDS) [206], we selected TrivialDB [176] to integrate. Developed as the database back-end for the Samba project, TrivialDB was selected as it was somewhat mature, provided good parallel processing support, and had performed well in DBM survey studies [89]. TrivialDB should not be confused with the better known, and similarly named, Transactional DBM (TDBM) project [29].

Unlike many other database implementations, TrivialDB allows multiple readers and writers connections to be open simultaneously. Moreover, it allowed a connection to be opened once within Greenstone's program flow and then persisted through the life of the processing. This addresses both of the issues identifed above, and so was added to Greenstone as an integral part of the move to parallel processing. Implementation details were published at the 2012 Workshop on Very Large-Scale Digital Libraries [174] but, in short, TrivialDB was a drop-in replacement for GDBM requiring only minor code edits to support the change. TrivialDB versions of the database executables called by Greenstone were implemented, and the option of configuring Greenstone to use TrivialDB added to Greenstone. This change in database yielded immediate results, as shown in Figure 5.12. By removing the linear overhead of opening and closing database connections, Greenstone experienced significant reductions in processing time. Unsurprising when you consider the case of a million document import process dropping from some 4 million short-lived database connections to a total of 4 persistent connections. Indeed, early experiments involving the processing of text in serial yielded processing times nearly as fast as parallel processing, predominately due to the change in database.

While TrivialDB's initial performance was impressive, moving to a cluster of machines revealed an issue. The Samba software—and by extension TrivialDB—achieves the illusion of parallel writers by using file locks to synchronise access to ranges of bytes within the database [5]. This semaphore-based approach works well on a single multi-core machines, or where all remote machines support exclusive or opportunistic locking, but does not work on clustered machines as they typically use some form of shared file system—such as NFS—and the properties of file locking on shared files are not well defined. While there is a version of TrivialDB specifically designed for use on clusters, CTDB [177], it is reliant on use of a clustered file systems, such as Luster [208], RedHat's GFS [28], and IBM's GPFS [16]. These file systems are not typically available on default cluster software.

Having identified a third requirement in that any database implementation should be stable when run on the default shared network file system, we investigated databases that had some form of centralized control point—for synchronisation via file locking—but without triggering a bottleneck where access must still be serial. This prompted us to develop client-server versions of both GDBM and TrivialDB. This communication model allows multiple clients, each writing to the database, to be synchronised by a single server. In the case of TrivialDB, this server can in turn use multiple threads to write to the underlying database. Meanwhile the GDBM version of the server made use of file locking, similar to the GDBML version of the database, in order to ensure synchronisation and retained the benefit of a single, persistent, database connection.

Finally, inspired by Hadoop's Reduce phase, we implemented a cluster version of TrivialDB that makes use of multiple database files, one per worker thread, that are merged into a single database upon normal program termination. This model exhibits performance similar to the original TrivialDB as the extra processing cost of the merging phase is minimal and is further offset by the performance gains due to there being no bottleneck to multiple databases being written at once. Meanwhile, the extra file transfer cost of writing multiple files is mitigated by the NFS layer and the optimisation therein.

The TrivialDB for clusters implementation later proved to be most suitable for all of the experiments carried out. Evidence of this, and of the relative performance of the other database implementations mentioned above, is presented in Section 5.4.1.

3.4.3 File System

While this research intended to model the parallel import process of digital libraries, it is impossible to do so without paying some consideration to the underlying file-system. Experimentation over the course of this project showed the drastic effect of differing file systems on overall performance. This section will discuss some of the major systems used and the challenges they entailed. As an overview, this section will talk about: scoping-out hardware concerns, network file systems, distributed file systems, Hadoop Distributed File System, Thrift, NFS-Proxy and MapR.

The scope of this research attempts to delineate between improvements in processing speed due to digital library, database, or parallel processing framework choice from those garnered by specific operating system, hardware, or topography improvements. The exemplar measurements given in Appendix A were gathered from modest hardware configurations of typical commodity computers and network devices. Should the intended hardware be significantly different, especially in terms of file input and output either from local disk space or across a network in the case of cluster computing, more accurate measurements should be garnered by processing a smaller sample collection and measuring key metrics as explained in Section 5.3.

Further to the above, and as mentioned in Section 1.4, there is a wealth of current research showing the affects of file system choice and configuration on the potential benefits of parallel processing. We have already mentioned research showing the benefits of carefully matching network configuration with network topography [113], or the more advanced step of segmenting the topography using optimised weighting to improve performance [152]. Moreover, the choice of software can alter performance as evidenced by research on configuring operating systems to topography for specific parallel processing tasks [196].

None of the test-bed hardware used during the experimentation documented in this thesis have been specifically optimised in terms of operating system, file system, hardware, nor topography. The single multi-core computer made use of a standard install of Ubuntu 12.04 LTS with the only significant change being the avoidance of the Logical Volume Manager feature. LVM allows for more flexible drive partitioning and support for dynamic addition of hard drives and allocation of drive space, but introduces another layer of complexity to an already complex file system, comes with its own time overhead, and even has the potential to damage the stability of normally safe Linux file systems.

The Rocks-based Beowulf cluster used in experiments is assembled from fifteen commodity PCs connected by standard CAT5 cabling to a 24 port 100Mbs switch in a standard star topography. The majority of tests on this cluster involved files shared by a default network file system mount, with the only optimisation being a significant increase in the number of file server daemons to sixty-four instead of the default eight. This configuration was chosen to reflect the sort of hardware available to small to medium sized institutions.

Network File System (NFS) is the standard file system for managing files that reside on a remote file space. It works by having a server computer share a region of file space via a number of communication daemon processes. Other machines on the same network run client software to connect to the server, are assigned their own daemon, and then communicate file actions to the server. The client software allows this remote file space to behave similarly to any local file space: all of the basic file actions are supported, such as copy, move, and delete. By default, a Rocks cluster uses a standard NFS as the means of sharing space across compute nodes in the cluster.

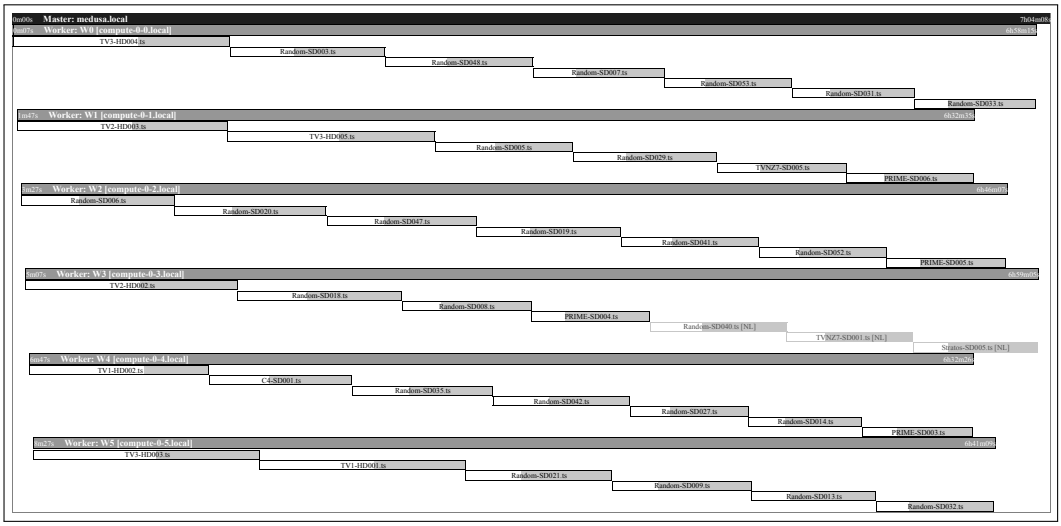


Figure 3.2: Preview of staggering the start of workers to avoid NFS contention in a parallel batch import of video documents

The default networked file system may not be the best choice of file system for optimal parallel processing. A standard NFS set up effectively creates a file transfer bottleneck on the single machine responsible for serving out files to the various compute nodes. This bottleneck becomes significant when processing files which require high file transfer but low processing requirements. It can even sometimes be problematic for high processor load tasks. Consider the import previewed in Figure 3.1 and provided in full in Figure B.1. Within the first few moments all fifteen processing nodes attempt to access video files. This causes an increase in file transfer time for these initial videos due to

contention. This becomes more apparent when we re-run the same experiment, but with a slight delay between the starting of each worker, as previewed in Figure 3.2 and displayed in full in Figure B.4. The file transfer time for high definition video segments is now more similar. Reviewing the statistics shown on the full chart, the average percentage of time spent on file transfer per file drops from 74% in the first experiment to 67% in the second. The total processing time also decreased, but this is not conclusively due to the staggered start as the order in which files are processed is somewhat random, and so the balance of work between threads and thread starvation need to be taken into account.

While the latest version NFS has better support for parallel processing [88], a distributed file system might be a better choice of file system on a cluster. Such a file system balances the files to be shared over all the nodes in the cluster and similarly distributes most of the file management work. There are several distributed file system implementations available for Rocks-based clusters, for example the popular Lustre file system [159] and its several derivatives. However most of these file systems, including Lustre, require significant changes to the operating system or hardware configuration to install. Instead, this research made use of the Hadoop Distributed File System (HDFS) [26]–included as part of Hadoop—to measure the potential for distributed file systems to improve performance. Critically, the initial set up and initialisation of this file system does not require any changes to the underlying Rocks operating system nor to the hardware topography.

Using HDFS as the file system allows the Hadoop framework to better exploit *data locality*, scheduling a processing task to be processed on the same compute node that a copy of the file to be processed resides. This has benefit of moving the process to the data, drastically reducing the file transfer costs. The possibility for data locality is directly related to the configured *replication factor*—the number of nodes that a copy of a particular file exists upon. A low replication factor, such as a single copy, maximizes the amount of space in the distributed file system but reduces the chance of data locality during a CPU intensive parallel process as a compute node may already be in use when another tasks whose file resides on that node comes along. Increasing the replication factor allows for more chance of data locality as the scheduler tries to match process to task, but at the trade-off of consuming more file space on the file system.

During the course of this research several experiments were run to determine effects of replication factor on data locality and hence parallel processing

performance. The results presented later in this research (Section 5.4.3) show that the greatest gain when parallel processing digital libraries occurs at a replication factor of 3, with diminishing returns thereafter. Serendipitously, HDFS's default replication factor is set to 3, although that is likely due to concerns of data-safety (evidenced by the algorithm for placing the file replicas [26]) or inspired by previous research on RAID5 systems [46] that showed 3 replicas provided a good trade-off between redundancy and performance.

From within Hadoop access to files on the HDFS is transparent—they are referenced just like any other file except using the prefix "hdfs:". Outside of Hadoop, access to this distributed file system is either through libraries integrated into the project or by a Java-based command-line tool. We chose to evaluate the latter so as to measure realistic support for general purpose digital library software rather than customised/bespoke solutions. The tool offers full support of HDFS features but with the added start-up and shut-down costs of the Java-based tool were sometimes magnitudes above the cost of the file interaction itself.

One improvement is to move instead to a client-server model where the connection to the HDFS is created once and then persists throughout the import process. There is an existing Hadoop project that supports such interaction, the Apache Thrift framework [163]. Thrift allows a user to define communication protocols or application programming interface and then, when run, instantiates a server adhering to that definition. The goal of the Thrift project is to improve portability and interoperability, balanced against the extra cost of transforming data to be platform agnostic. Access to HDFS via Thrift was also experimented on during this thesis's research.

The optimal solution would be to provide transparent, NFS-like access to the HDFS system. A commercial distributed file system solution offered by MapR Technologies provides direct file access to HDFS files [183] but this solution required the cluster to be a specific hardware configuration well beyond the means of individuals and small sized institutions. There are some open-source efforts being made to connect these two file systems, most notably the HDFS-NFS-Proxy component from the Cloudera project. Unfortunately this component is still under active development and proved to have several issues that ultimately meant that experimentation was fairly limited. The issues encountered were:

- Difficulty in matching user credentials between HDFS, NFS, and the Linux system, the only solution for which was hard-coding machine name, user, and group ids in the digital library software.

- Random file truncations when files are over a certain size.

- The proxy software would erroneously consume all of the processing and memory resources on the node hosting the network file system server resulting in the cluster becoming non-responsive until restarted.

An alternate method of ensuring that data is available to all compute nodes is simply to provide each node with a copy on its local file space. This is typically impractical for very large-scale digital library collections due to their size requirements. In contrast, there may be ways to reduce the number of replicas by trying to improve data locality. Were a digital library import process able to perfectly match the processing to the node that the file resided on, perhaps leveraging metadata associated with the file, then only a single copy on the cluster would be required. Unfortunately, this is a non-trivial problem considering fault detection and tolerance, and as evidenced by the complexity of Hadoop's mechanisms for data locality. Moreover, any such mechanism still eventually requires local file access in order to distribute the collections contents out to the compute nodes so there may not necessarily be any savings in processing time once this is factored in. Indeed, experiments showed little difference between importing from local file space and importing from the networked file system other than a initial high contention moment explored in the earlier staggered starting experiment shown in Figure 3.2.

3.4.4 Video Processing

One of the more CPU intensive document types to import into a digital library are video formats. As mentioned in the case studies (Section 2.2.7) much research has gone into how to effectively present multimedia files since they do not suit the traditional full-text search paradigm as well as text does. These novel presentation and interaction approaches come at the cost of extra processing demands: either when building the collection or at runtime.

Support for video collections is now available in Greenstone via the video-and-audio extension. This makes use of several third-party tools, such as MediaInfo, Handbrake, FFMPEG, and Hive2čiteHeesch:2003:Video, to allow for more feature-rich processing of video files. In particular, the Hive2 library provides data mining techniques for extracting key-frame images that better partition the media—as compared with the more common but arbitrary technique of extracting key-frames at a certain fixed period.

In order to fairly test video processing across the three digital libraries investigated, the same video processing steps were re-implemented for both DSpace 4.2 and Terrier IR 4.3. Aside from some minor alterations relating to where files are located, and how metadata is added to the internal document representation, the processing work was kept the same. Since the converting and keyframe extraction steps quickly overshadow any other processing cost, the expectation (confirmed through experiments detailed in Chapter 5) was that video processing performance will be essentially the same between the three libraries.

The use of Handbrake and other third-party libraries raised another challenge when running parallel processing timing experiments since many modern image handling libraries attempt parallel processing by default, utilising as many cores as available. To control this factor, parallel processing within that tool was disabled during its configuration. Where such a configuration setting was not possible, system level tools were used to restrict the processor affinity to a single core.

3.4.5 Very Large-scale Corpus

In order to test the performance of the digital libraries mentioned above it was necessary to gather a number of test corpora of significant size as to provide meaningful stresses, comparable to those found in real-world very large-scale digital libraries. There are several well-known corpora available on-line, such as those available from TREC [83], although these tend to focus on text and on specific aspects/problems of information retrieval. This research makes use of several test collections in order to experiment with certain media types or expose certain combinations of material and process that were expected to raise challenges. An overview of these test corpora is given in Table 3.1.

It should be noted that it was tempting, at several points during the development of the import materials, to create multiple copies of the same documents in order to reach the desired numbers. This technique might have be sufficient as this research became focused on the ingest phase of collection building. However, this technique would not provide an adequate stress test for the indexing phase as the repeated content would exhibit a fixed number of terms and would not increase the size of the index as expected when processing real-world documents. At some stage the lexicon would stop growing and entries in the inverted index would simply be repeats albeit with different document identifiers. In comparison, research has shown that indexes built

Table 3.1: Custom corpora used during experiments

Corpus	MediaType	#Files	Avg. Size (KB)	Total Size (GB)
Lorem Ipsum	OCR Text	1,000,000	3	2.86
	200 to 8,500 characters, average word length 16 characters, 1.2 new unique terms per document			
Wang	Images	1,000	29	0.03
	predominately JPGs, with some PNGs and GIFs, of widely varying sizes			
Jamendo	WAV	96	43,690	3.99
	WAV files ranging in duration from $1m44s$ to $19m43s$			
ReplayMe!	TS Video	96	351,820	32.21
	mix of high (1920x1080) and standard (720x576) definition video files encoded in AVC format at 25 fps and broken into 10 minute segments			

from machine generated text, common for very large-scale digital libraries, continues to encounter new unique terms for each document [15].

In order to create a scalable, and copyright free, OCR-like text collection, this research repeated an idea similar to Lorem Ipsum text. A n-gram model [162] was created from a large sample of "*de Finibus Bonorum et Malorum*"—the source document for Lorem Ipsum. Briefly, an n-gram model uses the previous sequence of n items and probabilities of sequences starting with those items to predict the next item. The granularity of item can be set dependent on application; in this research we used characters, but models can also be built using words, syllables etc. The n-gram model was then further tuned, by varying n, extra weighting factors, and configurable levels of noise characters, to produce material that had text metrics similar to the Elephind newspaper materials introduced in Section 2.2.6. Specifically the n-gram model was able to introduce enough noise character to continue the linear growth of new words to index in each document. This model could then be used to generate whatever number of plain text documents was required, ranging from a few hundred to several million. This collection is referred to during experiments as the "Lorem Ipsum" collection.

The second corpus formed for this research was a 1,000 image subset extracted from a larger corpus [195] used during development of the content-based image retrieval software *SIMPLIcity*. This material is a subset of the even larger COREL corpus (some 200,000 images). These image corpora provide a wide range of image sizes and types—predominately JPEG format but

including a number of GIF and PNG images—similar to what might be found in a typical image-based digital library. This research's corpus is referred to as "Wang".

The third corpus was sourced from the royalty free, music download site, Jamendo. It was created by downloading the top 100 free, MP3 format, music tracks, which were then converted into WAV format. The resulting audio tracks vary in size, quality, and duration, but are representative of a typical audio digital library. Ultimately 4 of the files were discarded due to them being malformed or damaged in some way, leaving 96 audio files. This corpus is subsequently referred to as the "Jamendo" collection.

The fourth, and final, corpus was gathered from the import files for the ReplayMe! system [155]. This quickly supplied a significant amount of material as for each hour of television air-time it records approximately 16 hours of video. In New Zealand, digital television is broadcast in H.264 (or MPEG-4 Part 10, Advanced Video Coding) format, which the ReplayMe! system writes to disk as MPEG-TS files. This video is (arbitrarily) split into 10 minute segments to increase the opportunity for parallel processing. This test corpus contains a single hour of television air-time, resulting in 96 video segments, and is referred to as the "ReplayMe!" collection.

Chapter 4

Implementations

> He knew what he had to do. It was, of course, an impossible task. But he was used to them. Dragging a rat all the way from the wood to the hole had been an impossible task. But it wasn't impossible to drag it a little way, so you did that, and then you had a rest, and then you dragged it a little way again . . . The way to deal with an impossible task was to chop it down into a number of merely very difficult tasks and break each one of *them* into a group of horribly hard tasks, and each one of *them* into tricky jobs, and each one of them . . .
>
> — *Terry Pratchet, "Truckers", 2004*

This chapter discusses the implementation details of applying parallel processing to the importing phase of the three selected digital libraries: Greenstone, DSpace, and Terrier IR. The rational behind this selection of digital libraries was given in Section 2.3. For each of the digital libraries we describe the default sequence by which documents are imported, followed by details on how the Open MPI parallel processing framework was integrated into this sequence. This is repeated for the Hadoop framework integration, and then the beneficial outcomes of parallel processing are discussed. Furthermore, while the more general challenges encountered have been discussed in Section 3.4, this chapter provides additional details of challenges specific to a particular digital library system where pertinent.

4.1 Greenstone

In this section we describe the process by which Greenstone builds digital library collections, pointing out areas which may benefit from parallel processing approaches. We then present two implementations of parallel Greenstone: one using the Open MPI framework and the other Hadoop.

4.1.1 Overview

A Greenstone digital library logically involves two distinct phases: *build-time* and *run-time*. The build-time phase is the process by which a gathering of disparate document, image and other digital media files, is processed into a usable digital library collection. Build-time, itself, can be further broken into two sub-phases: *importing*, where the documents are transformed into a standardized format, and *indexing*, where information is extracted, processed, indexed, and stored in the built collection. Subsequently, the run-time phase generates pages of content from the previously built collection. It does so based upon a series of arguments included as part of the URL request and delivers the content to the user, typically as web pages viewed on a web browser. This phase is generally run as part of a web-server.

Looking at build-time in more detail, the act of importing starts by searching through a configured input file directory, checking for each file found whether there is an import *plugin* available to parse and extract metadata from this document format. Each file may be processed by multiple plugins, with each plugin generating some number of derived intermediate files in the output archives directory. These intermediate files are generally centred around a representation of the input document in the form of well-structured and machine readable XML. The transforms applied by the plugin vary greatly in complexity. A simple plugin might wrap the entire content of a plain text file in the appropriate mark-up. A more complex plugin might involve deconstructing a MARC file containing multiple records into many intermediate documents, one per record, and then using each record's unique identifier field to bind that metadata to the full text and/or other media located as separate files. This plugin exhibits complexity by way of multiple interactions and relations between input, intermediate, and final files.

A second form of complexity the plugins can support is centred on content analysis. For instance, a plugin could utilise machine learning or artificial intelligence techniques to analyse the content of a video file determining the

Table 4.1: Example collection building times in Greenstone (hh:mm:ss).

No. of Documents	50K	100K	500K	1M
Importing time	3:22:34	6:52:18	35:10:59	74:35:17
Indexing time	15:29	32:42	3:51:56	7:02:58

most appropriate key-frame information to be then stored as metadata values within the intermediate document representation.

Anecdotally, Greenstone's build-time typically grows in a linear fashion to the number of documents imported, with importing taking significantly more time than indexing. Reproduced from earlier research [15], Table 4.1 contrasts the importing and indexing times of a Greenstone collection containing over one million documents. These documents were of the same general form, namely plain text files extracted by optical character recognition (OCR) from historic newspapers, but varied greatly in character count. In this configuration importing takes significantly more time than the building of indexes and processing time is proportionally linear to the number of documents being imported.

Figure 4.1 presents a flowchart of Greenstone's build-time processes demarking the importing and indexing stages. Notice the example of the bottom two documents imported being logically joined together as part of the importing process. This figure helps to illustrate where the potential lies in the system for supporting parallelism. During importing, each document is handled independently thus lending itself well to parallelism. Compare this with indexing, where the main opportunity for parallelism is the multiple passes that the indexer makes, only some of which are independent of one another.

To expand on this latter point, certain configurations of indexer in Greenstone offer more parallel processing opportunity than others. All indexers in the system have three broad common requirements: compressing and indexing the text of the imported documents, indexing the metadata of the imported documents, and building an information database relating metadata to documents. For the indexer used in Figure 4.1, *managing gigabytes* (MG) [205], text compression and indexing (via an inverted index) must be completed sequentially as later index building depends upon them. After this point there are some number of separate, independent, indexes built for each metadata field configured to be indexed. Finally, a database is populated, containing information gathered during the indexing steps. As such, MG offers some po-

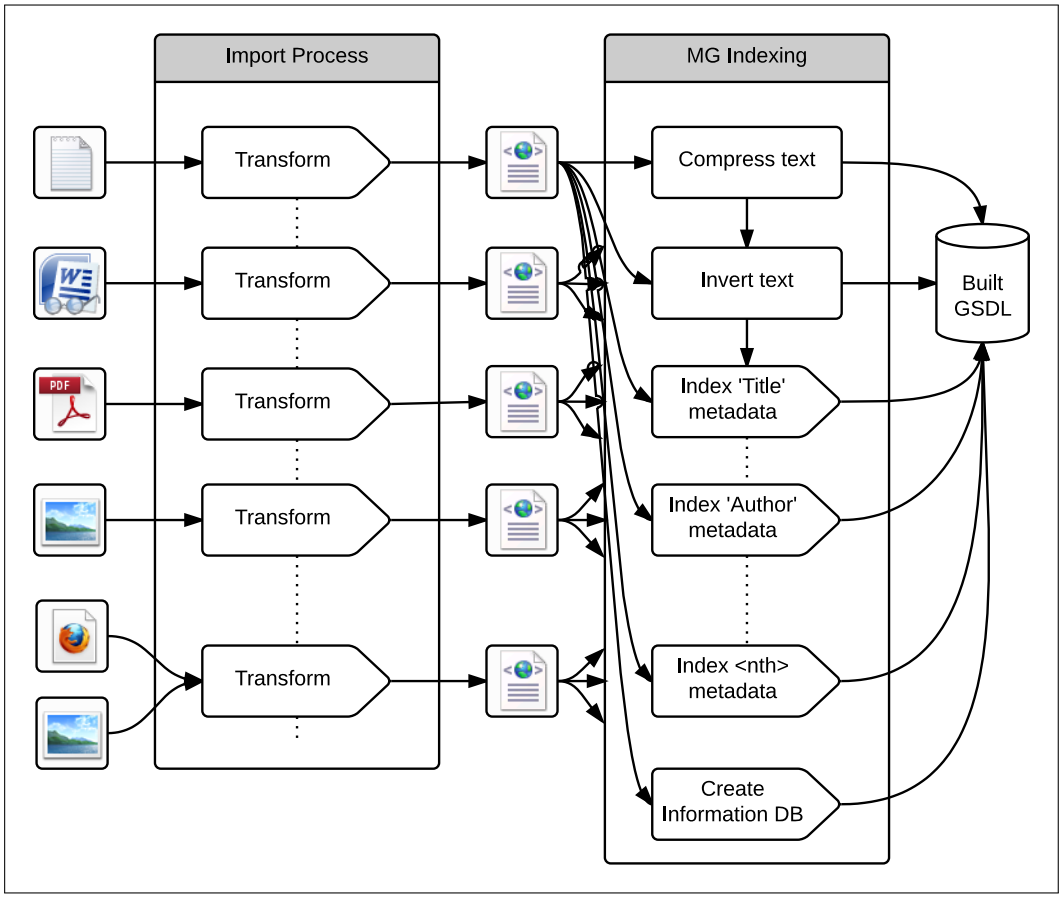


Figure 4.1: Parallel processing opportunities in Greenstone's importing and MG indexing processes

tential for parallelism in the indexing stage. If we naively assume all steps take roughly the same time to occur, t, then for a index of n metadata fields, parallel processing can theoretically reduce processing time from that given in Equation 4.1 to an to given in Equation 4.2. This assumes no limit to the number of physical processing cores available; as many as $n+1$ would be required for optimal performance [77].

$$MGSerialIndexingTime = (2+n) \times t \tag{4.1}$$

$$MGParallelIndexingTime = (2 \times t) + (\frac{1}{n} * t) \tag{4.2}$$

While this may initially seem promising, MG is a very basic indexer, lacking more advanced search functionality, and is inefficient in the way it builds indexes one at a time. Contrast this behaviour with that of the more modern MGPP [201] and Lucene [82] indexers for Greenstone, as shown in Figure 4.2. The former builds all of the document and metadata information into an index

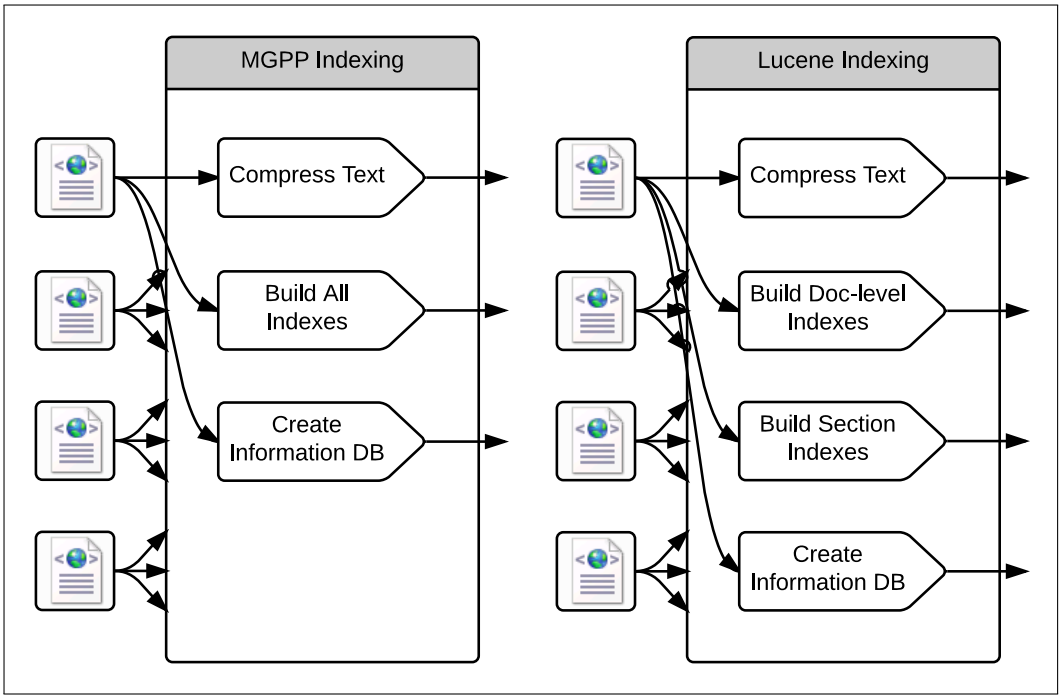


Figure 4.2: Parallel processing opportunities in Greenstone's available indexers

in a single step, while the latter builds at most two indexes, one at document granularity and another, optionally, at section granularity. Neither of these indexers need run text compression and indexing sequentially. Applying the same assumption—that all steps take time t—then similar optimal indexing times for Lucene and MGPP are displayed in Equations 4.3 through 4.6.

$$MGPPSerialIndexingTime = 3 \times t \tag{4.3}$$

$$MGPPParallelIndexingTime = \frac{1}{3} \times t \tag{4.4}$$

$$LuceneSerialIndexingTime = 4 \times t \tag{4.5}$$

$$LuceneParallelIndexingTime = \frac{1}{4} \times t \tag{4.6}$$

The limited potential for parallel processing severely limits the total number of workers required; Lucene would make use of five processing cores maximum, and MGPP only four. However, the limited potential for parallel processing on the MGPP and Lucene indexers are not intrinsic to the indexers themselves but are a consequence of Greenstone's implementation of them.

Reimplementing them with more opportunity for parallel processing would, in turn, expose more benefits.

4.1.2 Scope of Work

While some development was undertaken to support indexing, the work reported here and later experiments focus on the importing stage of the build-time processes of Greenstone. This focus was chosen as importing offers:

- clear opportunities for parallelism as illustrated in Figure 4.1,
- greater possibility of significant savings in processing time as importing is generally slower than indexing, particularly for processor intensive documents like multimedia and images, as illustrated in Table 4.1,
- room for experimentation such as altering the number in each group of documents being ingested by each process.

Further, the accompanying issue of large scale full-text indexing is already being explored (for instance [17]) and so concentrating on importing is more novel. Indeed the existing literature suggests it is not guaranteed that Greenstone's indexing algorithms and architecture would actually benefit from using multiple processors as some indexing processes are more efficient in serial [150]. Also left for future research is the opportunity to apply parallel processing to the run-time processes of Greenstone.

It is also worth noting that, at the time of this research, MG, MGPP, and Lucene were the only indexers publicly available as part of Greenstone. In the interim, Solr has been experimentally added as a fourth indexer option to Greenstone. Solr, as mentioned in Section 2.2.1, is the successor to Lucene and is capable of all the functionality of its predecessor while adding additional search functionality and significant optimisations geared towards web access. While not leveraged in the current implementation, one advantage of Solr is that document importing is provided as a web-based action, potentially allowing massively parallel import of documents. The impact of this will be addressed during the concluding chapter of this thesis.

In a challenge particular to Greenstone, there is at least one place where the import stage becomes difficult to process in parallel, specifically in the case of inherited metadata. Greenstone allows imported documents to be annotated

with accompanying metadata by including one or more *metadata.xml* files. Each file contains a file pattern, expressed as a regular expression, indicating the applicable documents and the metadata to associate. This feature is not provided by DSpace or Terrier. The complication arises for parallel processing in that metadata can be defined higher up in the import directory and the file pattern designed to match to child directories and files. During this research this case is avoided by stipulating that inherited metadata is not supported during parallel processing. When parallel processing is applied to a collection marked as containing inherited metadata in its configuration file, a preparation script is used to temporarily propagate the metadata down to the document it is assigned to.

4.1.3 Open MPI and the SPMD technique

In the following section we will show how Greenstone was extended to add parallel processing support utilising the Open MPI framework. But, before going on to give GSDL specific details, we provide a digital library agnostic overview of the framework's integration, as this is of benefit to the various DL architectures considered. Open MPI [123] was introduced in Section 3.2. This open source project has been developed with high performance in mind and supports a plethora of operating systems, as well as cluster and multi-core configurations.

The general approach to integrating Open MPI into a digital library's import process can be visualised using the flowchart shown in Figure 4.3. Technically, this approach uses a single program, multiple data (SPMD) technique [53] for parallel processing. The process begins with a top-level script that generates a file listing of the files to be imported. Open MPI is then invoked and this single large file listing passed in. The master thread then splits the input listing into smaller parts, based upon some configurable size typically referred to as the batch size. The master then initialises a variable number of worker threads and communicates the file path to one of these smaller file lists through to each worker thread. The worker then calls the appropriate functions within the applicable digital library collection in order to batch import their listing of files. The result returned to the master thread can be as simple as a one for success or a zero for failure.

The implementation developed during this research depends solely upon Open MPI's low-level fault tolerance: any failure within the Greenstone software causes the entire process to halt with an appropriate error message. Note

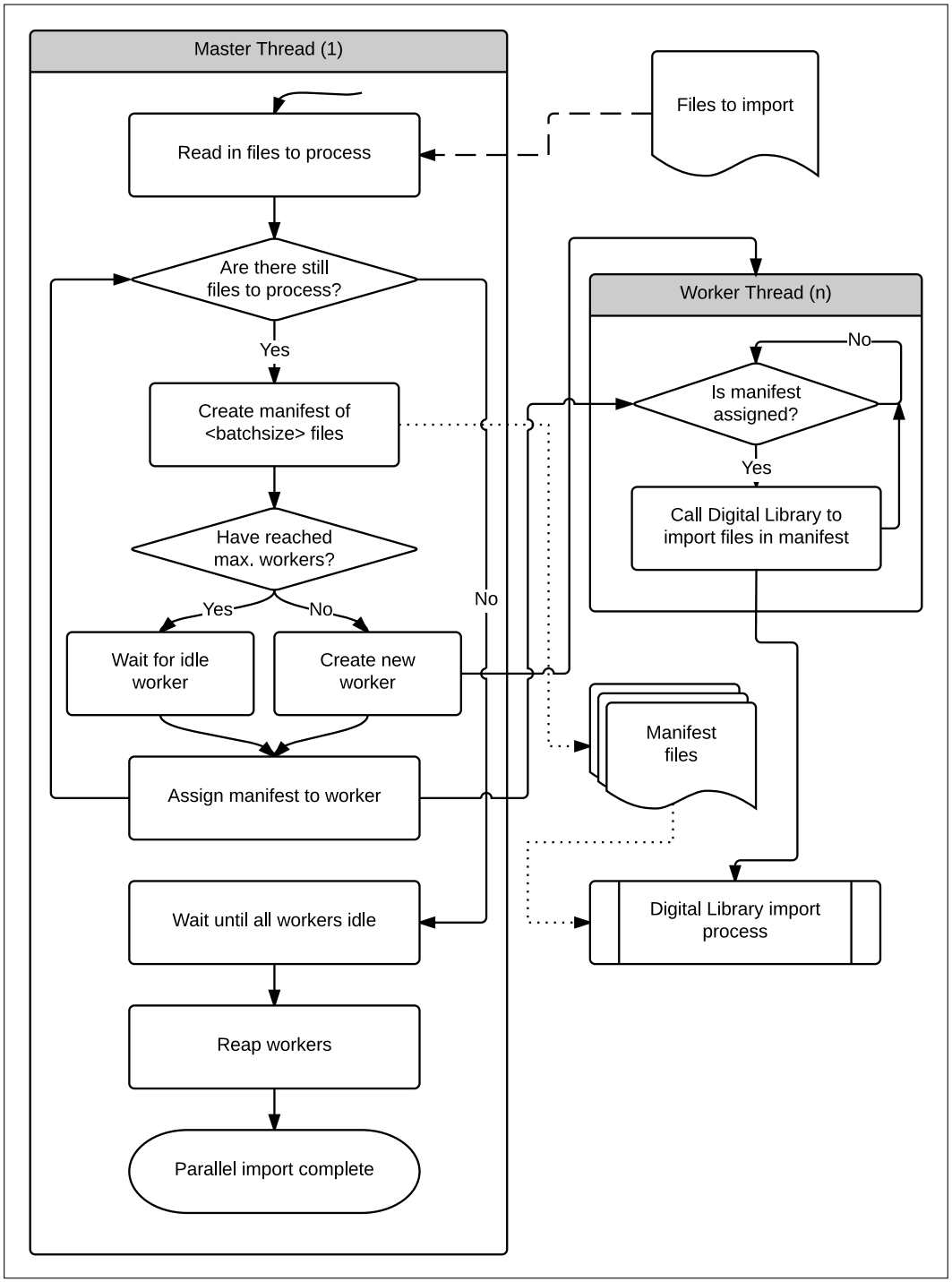


Figure 4.3: Open MPI applied to generic document import process

also that the batch size can be varied on a per-collection basis to ensure the optimal balancing of tasks amongst the workers. Without this balance it is easy to reach a situation of resource-contention where one or more workers are left with nothing to do (starvation) while others still have many files to process.

4.1.4 Parallel Greenstone using Open MPI

This section details how the Open MPI framework was specifically applied to the import stage of Greenstone in order to support parallel processing. Open MPI's SPMD technique fits well with the way importing is handled in many digital libraries including Greenstone. Figure 4.4 repeats the SPMD flowchart, but amended with specific details for Greenstone. The Greenstone import process was extended to use the Open MPI framework to allow for a single top-level import script to manage a number of worker import processes, each of which would ingest one list of files using a standard serial Greenstone import process. The parallel import script allows configuration in terms of number of worker threads (parallel jobs) and batch size for files to be processed. Meanwhile the underlying Open MPI can be configured in terms of how many processing cores to utilise and how to locate compute nodes when run in a cluster environment.

Greenstone contains mechanisms for importing a collection's documents in batches. The size and content of each batch can be controlled by use of a file listing in XML format called the *manifest*. Each manifest file explicitly lists the file paths to documents to be ingested, and offers the ability to control what sort of import action should be applied: add document, update existing document, or remove document. This functionality was originally added to support incremental collection building, but means it is also suited for parallel importing in an message passing architecture too. Each worker thread is given its own, distinct, manifest of documents to process drawn from the total number of documents in the import directory.

Referring to Figure 4.4 the process flow devised for a Greenstone parallel processing import using Open MPI can be broken down into nine key steps:

1. A top-level Perl script initiates parallel importing, providing arguments instructing the import to use multiple worker threads and specifying the number of files to be included in each manifest file. The script

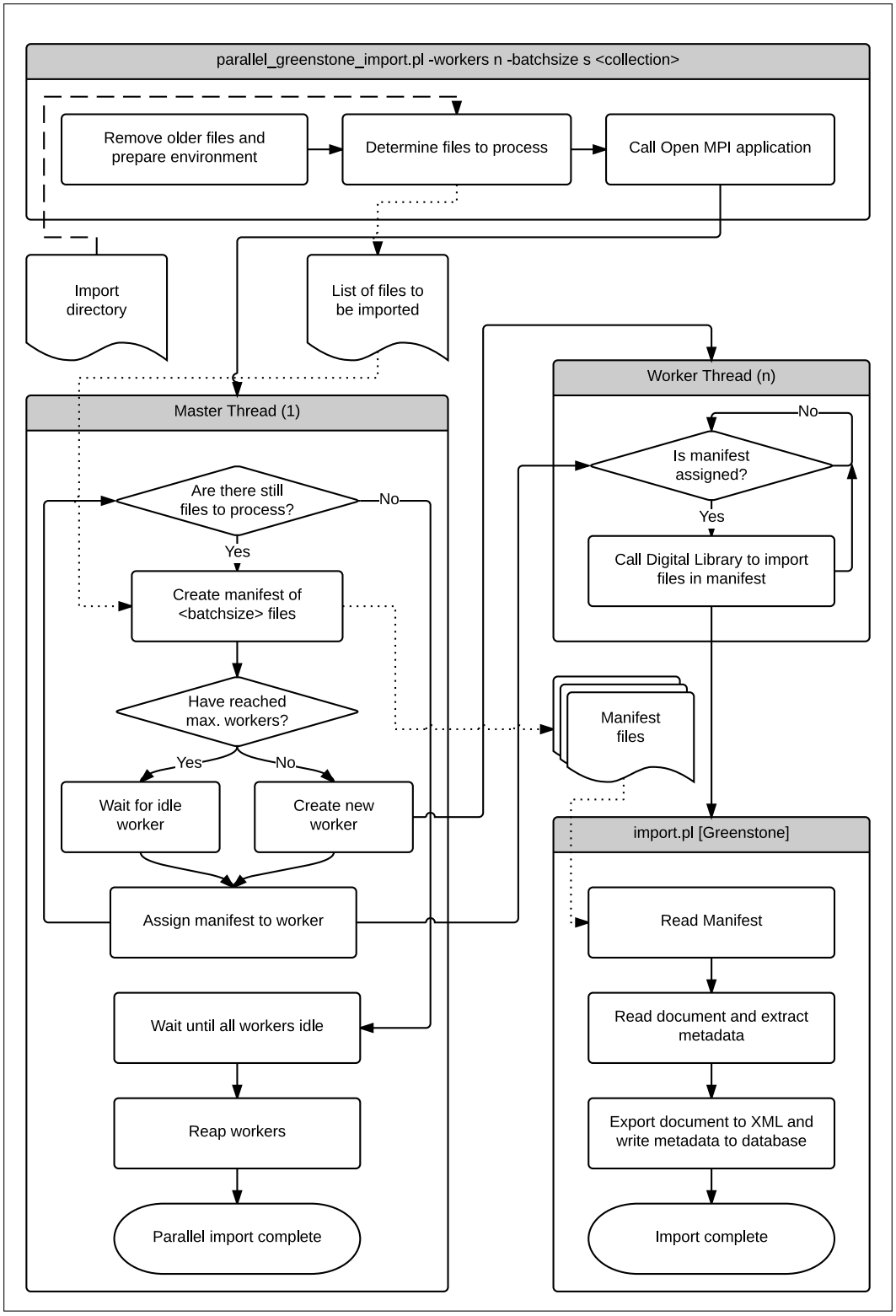


Figure 4.4: Open MPI applied to Greenstone document import process

also prepares the environment, removes any existing imported files, and flushes disk caches.

2. The script then scans the import directory generating a listing of all the files to import.

3. At this point Open MPI is called and asked to execute a parallel import. It starts a master thread, whose first task is to split the complete listing of files into smaller manifest files according to batch size

4. Worker threads are started and are handed a single manifest file to process

5. Each worker then makes a recursive call to the Greenstone import script providing the manifest file as an argument, but this time without the flags that indicate a parallel import.

6. Greenstone import, now running in serial mode, imports just the files listed in the manifest file.

7. Once the serial Greenstone is complete, focus returns to the Worker thread, which then becomes idle waiting for the master thread to allocate it another manifest file.

8. Processing continues until all of the manifests have been imported, after which the master thread shuts down all of the worker threads and returns focus to the Greenstone import script.

9. The Greenstone import script finishes writing log files, performs any final clean-up tasks, and completes.

It is worth noting that a typical, non-manifest, Greenstone import may actually perform more processing; for example, it performs a non-trivial pre-scan of all documents to be imported to locate any metadata files to be applied to the collection. Thus manifest files were passed to both the serial and parallel imports during experimentation so as to account for these algorithmic differences.

In terms of implementing parallel processing code within Greenstone, this research leveraged the recent addition of a new, flexible framework for adding extra functionality known as extensions. By following the required architecture when creating an extension, it can be configured, compiled, installed and executed all as part of the appropriate Greenstone processes. Enabling an extension is as straight-forward as placing its package in the `ext` directory, with

Greenstone being able to detect and reconfigure environment paths automatically during subsequent runs of the software. In this case a new extension, named `parallel-building`, was created.

At the heart of the extension are two executables, compiled from C++ code, that provide parallel support for document importing and index building respectively. The former, `mpi-import`, follows precisely the process shown in Figure 4.4. Meanwhile the latter, `mpi-buildcol` replaces the manifest file creation with code to generate recipe-like instructions explaining the steps and required ordering of building the collection's index. The extension also includes:

- the libraries necessary to compile the executables,
- changes to the Perl scripts used by Greenstone during build-time, and
- customised versions of the importing and index building scripts designed to delegate the actual work to the appropriate MPI-aware executable.

Implementation using the the Open MPI framework was fairly straightforward, building upon an initial prototype [173]. The framework also proved easy to move between a single multi-core computer and a Beowulf cluster [19]. There was only one significant issue encountered when deploying on the cluster caused by Open MPI binding to the wrong network interface, solved by explicitly stating the interface to use when executing `mpirun`.

The eventual performance improvement of parallel Greenstone is highly dependent on the number of processing cores allocated and the load of the processing during import, as detailed in Chapter 5. However, early indicative testing results were promising. For example, under optimal conditions, building on a cluster of w compute nodes, and where a serial import would take elapsed time t, the parallel processing elapsed times approaching Equation 4.7 were recorded. In practical terms, when considering the problem of importing an hour's time period worth of video recorded by the ReplayMe! digital library, a task that would take 60 hours on a particular computer may be reduced to approximately 4 hours when parallel processed on a cluster of 15 such computers.

$$ParallelElapsedTime = \frac{1}{w-2} \times t \tag{4.7}$$

4.1.5 Greenstone using Hadoop

In order to provide comparisons between differing parallel processing frameworks, a second implementation of parallel processing was developed for each digital library utilising the Hadoop framework [198] working in tandem with the Hadoop File System (HDFS) [26]. Hadoop, being built on the *MapReduce* paradigm, has a significantly different approach to parallel processing data but one that fits well within the nature of Greenstone's import process. This research started development with Hadoop version 0.20.205 (included as part of the Rocks cluster [144] distribution at the time) but later upgraded to Hadoop version 1.1.0. There were significant changes to function calls between these versions, but the improvements present in the latter version—specifically HDFS performance improvements—were worth the cost of re-factoring the code.

Practically, the processing work Hadoop needs to execute is defined by the implementation of an application that adheres to the Hadoop API. The application in question would typically need to include classes implementing the *Mapper* and *Reducer* interfaces, along with several other important classes (the grey coloured classes in Figure 4.5). In this case the application included a customised Mapper phase, to extract metadata and other information from the documents being imported. It also required a customised Reducer phase, to aggregate information from the mappers and generate the interim collection files. When this application is launched, Hadoop acts as the parallel framework, manager, and scheduler, distributing the application out to processing cores.

The files to be imported by the Hadoop-enhanced Greenstone software are placed into the HDFS using command line tools, allowing the application to leverage the benefits of data locality. While the *replication factor* (as discussed in Section 3.4.3) was left at the default of 3 for most experiments, it was straightforward to alter this, then rebuild the HDFS, in order to measure the effect of replication factor on performance. Moreover, since this implementation included a customised FileInputClass, it was possible to experiment on the "worst-case" scenario of imports that had no data locality; where processes always required files located on other compute nodes.

The import process followed the same general flow as for Open MPI, but with Hadoop specific changes:

1. A top-level Perl script prepares the environment and file-systems for testing.

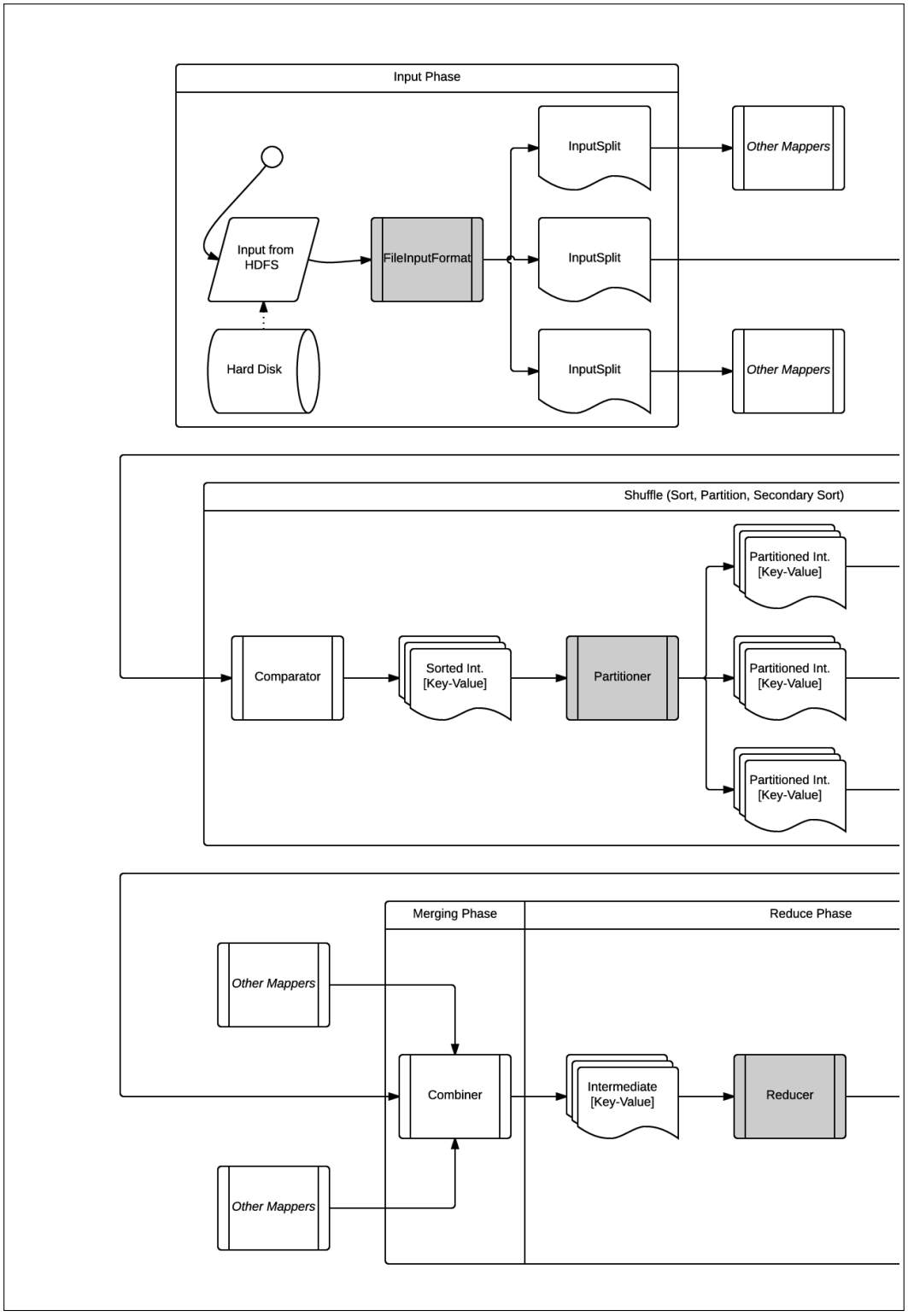
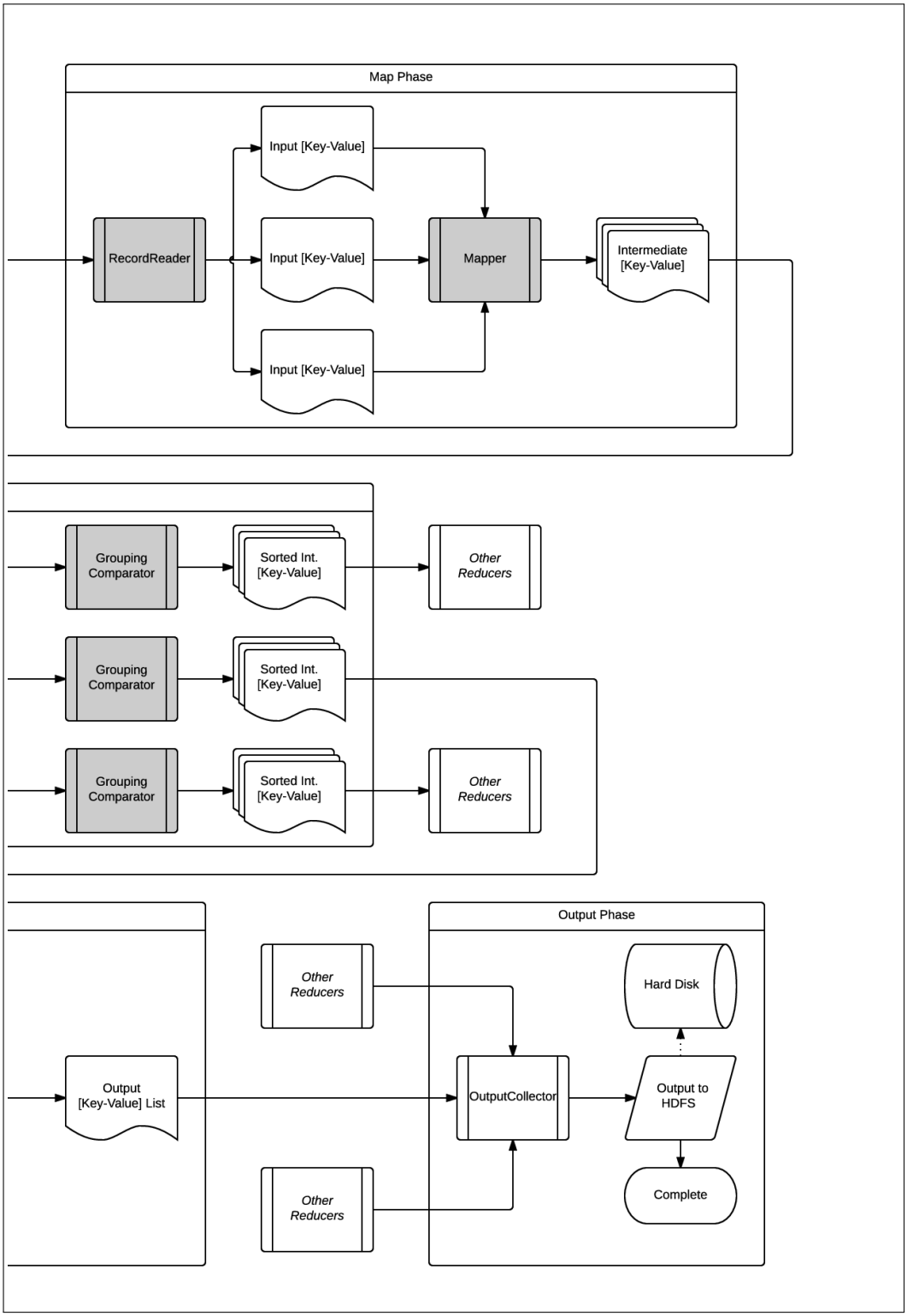


Figure 4.5: Relationship between customised classes, in grey, for Greenstone using Hadoop (continued next page...)


Map Phase
Input [Key-Value]
RecordReader
Input [Key-Value]
Mapper
Intermediate [Key-Value]
Input [Key-Value]
Grouping Comparator
Sorted Int. [Key-Value]
Other Reducers
Grouping Comparator
Sorted Int. [Key-Value]
Grouping Comparator
Sorted Int. [Key-Value]
Other Reducers
Output Phase
Other Reducers
Hard Disk
Output [Key-Value] List
OutputCollector
Output to HDFS
Other Reducers
Complete

2. The import directory is copied into HDFS in preparation for import, with the replication factor being configurable.

3. Hadoop is launched and asked to execute the Hadoop-enabled application.

4. Within the application a customised FileInputFormat class extends the default input class but treats each file found in the specified HDFS directory as a 'split'. These are marked as not being able to be split further. This new FileInputFormat class provides a hint to Hadoop as to where this split should be processed, given knowledge on which compute nodes replicated copies of the file reside. Hadoop then attempts to abide by these hints to improve data locality, depending on availability of idle Mappers.

5. A simplified RecordReader tokenises the single key/value pair contained in the single record in the split.

6. One or more Mapper implementations are then instantiated by Hadoop and are given one split to process. Map prepares the environment, writes a manifest file, and calls Greenstone, specifying that all information from Greenstone should be passed back to Hadoop via a pipe from the standard output stream. Each line of information is wrapped in XML that includes an attribute stating where the data should end up. For example, a line of information about an association between a document and its metadata will be marked as needing to eventually end up in the document information databases.

7. A customised Partitioner then groups the information returned by the Mapper, putting all information marked as the same destination together. Each of these collections of information will be eventually passed to the Reduce phase.

8. As part of the previous partitioning, a customised GroupingComparator class is applied to sort each group of information. This is only essential for the last information destination, namely those ordered by timestamp. The other three information categories are not time order sensitive.

9. Hadoop then initiates one or more Reducer implementations. The input to a Reduce class phase is a key/value pair where the key is the information destination and the value is an ordered list of the information bound for that destination. The Reducer looks at the destination of its list of information, generates the output container as required, be it a

database, flat file, or log file, and then writes the information to the output. Database generation is supported through helper classes that interact with the underlying database.

10. The import is complete once Mappers have processed all files found by the FileInputFormat class and all Reducers have written their value lists to the appropriate destination.

When partitioning information (step 7) there are four potential destinations in a default Greenstone import:

- **"doc"** - indicates this information is about a relationship between a document and metadata.
- **"src"** - indicates the information is about the association between a document and its files located on the file system.
- **"rss"** - indicates this information is an update message to be relayed to users of this collection using the standard RSS mechanism.
- **A timestamp** - indicates a log message to be written as debug information and to be ordered using the aforementioned timestamp.

There are several further implementation concerns to note. First of all, Hadoop's automatic retrying of failed tasks was effectively disabled by configuring the maximum retries to 1. This artificial constraint was applied for the purpose of testing; the test either succeeded or failed as a whole without the possibility of extra time spent retrying failed tasks. The Java specific issue of prematurely full standard output buffers causing the program to stall was addressed by the use of a separate thread to periodically empty the buffers either to file for debugging purposes or to null otherwise. It was also necessary to allocate another thread to report Greenstone's import progress back to the Hadoop system. Without this periodic progress report Hadoop would assume the process had stalled and time-out the import. A final extra feature, available only in the Hadoop implementation, was the ability to stagger the start of worker processes in order to determine if the single NameNode was a bottleneck to processing, especially in periods of high congestion.

By making use of both the Map and Reduce phases of Hadoop, the responsibility of synchronising the output from several parallel processes was moved back to a single centralised point allowing more flexibility with choice

of database. While the TDB for clusters database interface was used for consistency, the Hadoop implementation could have made use of less featured but generally faster GDBM variants. Full support of the MapReduce paradigm is a trade-off as the implementation can fully exploit the capabilities inherent in Hadoop thus providing a more efficient import, but the solution becomes more tightly coupled to the underlying digital library. Indeed this implementation prompted a significant addition to Greenstone, by way of a new plug-out that rendered collection import information as XML to the standard output stream so as to be machine-parsable by the later Reduce implementation. See Figure 4.5 for a graphical overview of the classes in Hadoop, how they relate to each other and to the customised classes and interfaces implemented as part of this research (shaded in grey).

4.2 DSpace

The second general purpose digital library software that was extended to take advantage of parallel processing was DSpace. This software was initially discussed in Section 2.3.1. DSpace is digital library tool, implemented in Java, that is closely modelled on traditional library processes and organization structure, presenting a rigorous structure to document creation, metadata, and hierarchical placement. DSpace was chosen to represent a digital library that stores all of the collection's information in a database. Text search in DSpace, by default, uses indexing built into the database rather than having a specific indexing stage. Such functionality can be added via Solr by enabling the optional *Discovery* feature [125] or even integration with Greenstone [202]. This research made use of DSpace version 1.8.2 source code package, as released in February 2012 [56].

For the purpose of this research, the challenges were, firstly, to determine how to approach document importing in a fashion similar to the other two digital libraries to be tested, and secondly, how to extend that approach to support parallel import via Open MPI and Hadoop.

DSpace supports the addition of new documents into its collection via two different mechanisms. Both approaches result in a document and its associated media file, or *bitstream* in DSpace terminology, becoming available within a certain collection in the system. The first, and most commonly used, is a web-based, multi-form process that guides a user, step-by-step, through the creation of a new document container and the upload of the target document and any associated derived files. This process assumes that the library creator

has performed any metadata creation or derived document generation prior to uploading the document.

The second import mechanism is a command line application providing batch upload functionality. This research focused on this method as it allowed for better timing of document creation and is better geared for large scale operations; it is also most similar to the way Greenstone and Terrier behave. This batch upload does not execute any processing on the uploaded documents: for that, another command line application called `filter-media` is used.

DSpace includes a number of media filters [57]; these provide the ability for various media to be further processed in order to derive bitstreams from the original media. Each filter takes an input bitstream, applies some form of transform, creates a single, new bitstream, and then attaches it to the original document's container. For example, there is the *HTML Text Extractor* filter that extracts the text from HTML formatted documents in order to allow for full-text searching. Multiple filters can be applied in a pipeline. Filtering is applied to the DSpace collection by calling the aforementioned `filter-media` application. The DSpace designer's intention was for the media filter tool to be called periodically—via an operating system task scheduler—in order to generate new content for collections containing media other than plain text. In this research, this tool was instead called by the same script that calls the batch import to closely mirror the import processes of Greenstone and Terrier. This effectively means the content in a DSpace installation would become fully available at the same moment of import, rather than some aspects appearing at a later point (once the OS task scheduler runs the filter).

This research required the creation of a new DSpace media filter, one that implemented the video processing as outlined in Section 3.4.4. This filter generates more than one derived bitstream as it transforms the video into a streamable format, but also generates a large number of keyframe images with accompanying timing information by way of an XML file. However, a DSpace media filter is only allowed to generate a single new bitstream. To work around this limitation the new filter stores the keyframe images as files in a temporary directory. These files are accompanied by a file named `contents` that contains one line per image associating them with the document container. Once media filtering is complete a second call is made to the batch importing tool, except this time the argument `item-update` is specified along with the path to the temporary directory set as the directory to import. This command goes through the contents file associating and uploading each of the keyframe images as a new bitstream against the parent document container.

Note that all of these derived bitstreams are added to the `ORIGINAL` bundle. At the time of this research, *Bundles* [169] were an emerging mechanism in DSpace that allowed the relationship between individual or groups of bitstreams to be specified. Extra functionality could then be attached, either manually or automatically, to certain bundle types. This technology was judged not mature enough to leverage as there were still questions about the implementation, scalability, portability, and security of this feature.

A challenge common to Java-based applications, and thus applicable to DSpace, was that the redirection of standard output and error buffers must be carefully handled. DSpace outputs several messages during the process of filtering media, but when running in parallel—possibly on multiple remote compute nodes—issuing these messages to standard output to appear on screen is problematic, especially if messages from different processing threads should not get mixed up. It becomes necessary to redirect these messages into distinct log files so as to be available to be reviewed during debugging, otherwise output buffers become full and Java will hang indefinitely waiting for room to appear in a buffer. This issue was addressed by the creation of a separate thread to periodically flush these buffers; either to file on debug runs, or to a null device during timed experiments.

A final implementation challenge was in suppressing the large number of files generated by DSpace in order to support the "undo" of the batch import, media filtering, and subsequent metadata importing. This was done in order to avoid the significant file transfer cost incurred by the generation of these numerous and small-sized files.

To generate the content for DSpace's batch file format we took advantage of Greenstone's export capability, which—via the *Plugouts* framework—allowed the export of an existing Greenstone collection in such a fashion that it was immediately importable into DSpace without further manual preparation.

At this point in the research, document importing in DSpace was at a point that it was directly comparable with the batch importing phase in both Greenstone and Terrier. The software was installed on both a multiple core processor and the Beowulf cluster in order to measure performance over a number of different collections of documents. Experiments in Chapter 5 will show that the performance is comparable with the other two libraries, although this is likely due to the most expensive cost—that of transforming and extracting metadata from the documents—being essentially common to all three libraries.

4.2.1 DSpace using Open MPI

Having implemented and tested a serial batch document import for DSpace the next step was to extend the process to support parallel processing via the Open MPI framework. The general process flow detailed in Section 3.2 can once again be applied, with each worker expected to process a subset of the total collection, as controlled by a file listing, in order to distribute the work load. Serendipitously, the `filter-media` tool takes a number of command line parameters including one that allows the caller to specify exactly which DSpace documents to apply the filtering too.

The process flow is, understandably, very similar to that of Greenstone with Open MPI (Figure 4.4), with the most significant differences occurring in the worker thread as visualised in Figure 4.6.

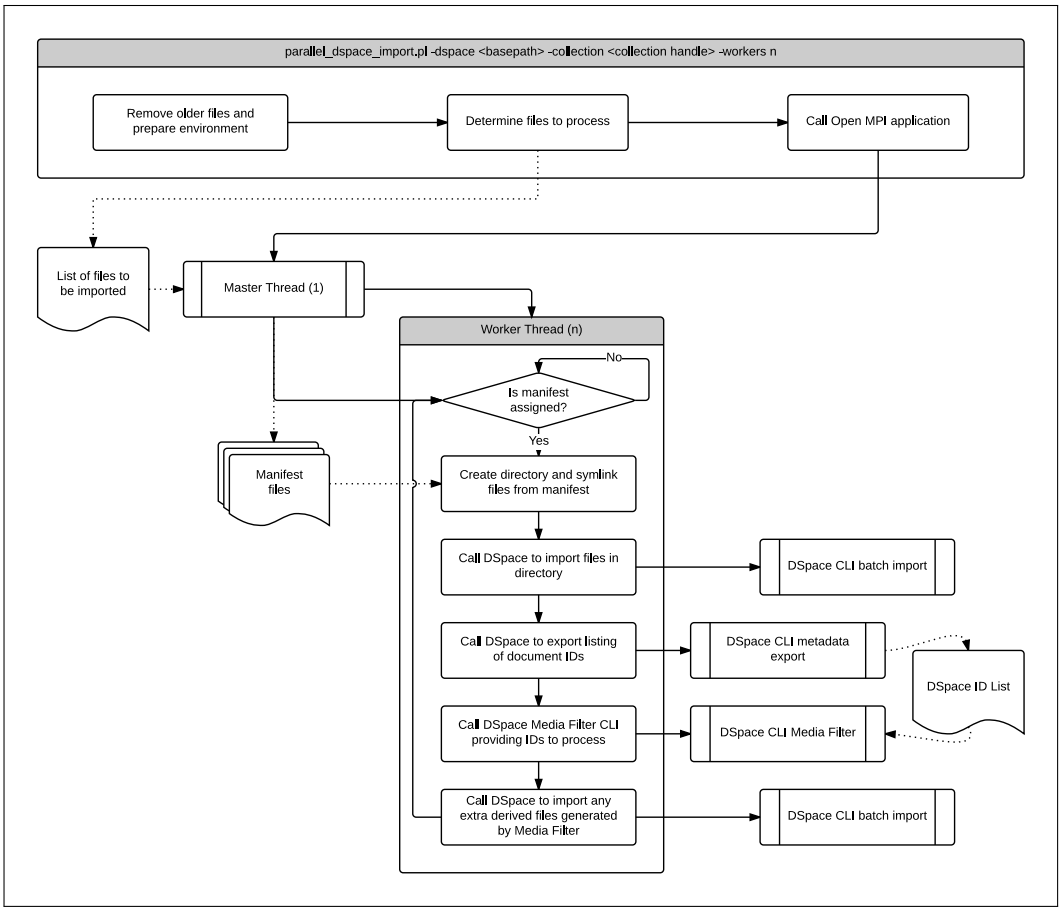


Figure 4.6: Open MPI applied to DSpace document import process - modified worker process

A parallel version of DSpace using the Open MPI framework has the following sequence:

1. A top-level Perl script prepares the environment for import, clearing out

old files and flushing the disk cache.

2. The script creates a listing of the files in the specified import directory.
3. At this point an Open MPI enabled application implemented in C++ is launched.
4. The master thread splits the complete listing of files into smaller batches.
5. Worker threads are started, and are handed a single batch to process.
6. The worker creates a new temporary directory with its batch of import files.
7. The worker calls DSpace's batch import application.
8. Once complete, the worker calls a second DSpace application to generate a list of the IDs of the imported files.
9. Now the worker calls DSpace's media filter application, passing in the IDs just determined.
10. The media filter creates the expected derived file, as well as any other consequential files, then associates them using the batch import application again.
11. Processing continues until all of the files have been imported.

It should be noted that there was an evolution to this implementation during the research, with an earlier prototype that only distributed the media filter stage of the processing in a parallel fashion (as reported on in our earlier parallel video processing paper [172]). The second version given above was subsequently implemented in order to more accurately reflect the potential benefit of parallel importing collections—especially those that have low processing costs—and more closely match the Greenstone and Terrier implementations.

Again, the performance of this system is superior to the serial one but is highly dependent on processing load and number of workers and/or compute nodes assigned to the task. Refer to Chapter 5 for detailed performance results.

4.2.2 DSpace using Hadoop

Unlike Greenstone, which leveraged both the Map and Reduce phases of Hadoop, DSpace only provided the means to test the former. This "lightweight" version uses the Map phase of Hadoop, and even then only to the

extent of calling a Perl script to do the actual DSpace processing. This version also makes no use of the sorting or aggregating of the Reduce phase, nor can it leverage the advantages of data locality.

Hadoop is only responsible for splitting up the initial listing of files into single offset plus file path pairs, and allocating them to *Mapper* instances. The Mapper instances in turn call a Perl script. The script in question essentially fulfils the role of a *worker* in the Open MPI process detailed above. It receives the path of a single file, creates a temporary directory, calls the batch import tool, the media filter tool, and the metadata updating version of the batch import tool, before returning. Once complete the Mapper returns a dummy value, which subsequently remains unchanged by the Reducer (configured to be the sterile IdentityReducer class) before being effectively ignored by the rest of the Hadoop process.

As the experiment results will show, this approach was not as efficient as the Open MPI one. This is arguably due to the implementation gaining all the negative costs implicit in running Hadoop (and Java) while doing little to leverage the potential benefits.

4.3 Terrier IR Platform

This section will discuss the application of parallel processing to the Terrier IR Platform—the third general purpose digital library considered by this work. This open-source digital library was introduced in Section 2.3.3, with the pertinent details being that it emphasises an index-based approach to data storage, was designed to be very large scale and high performance, and that it is a work-bench with modular structure allowing for extensive reconfiguration. The implementation work for parallel importing detailed in this research made use of Terrier version 3.5.

From a technical point of view, Terrier is a work-bench that requires a user to specify and configure the indexing implementation before it can be used. The downloaded package includes several example implementations for importing from traditional information retrieval data sets such as TREC's WT2G [84] and DOTGOV [83]. Of particular interest was the `SimpleFileCollection` indexer. This implementation is run as part of Java-based desktop application that provides full-text searching over one or more directories of files. From this starting point, a new customised indexer was developed that closely mirrored the batch importing approach seen in the other two digital libraries. The in-

dexer is invoked via the command line and has three distinct manifestations depending on the arguments given:

- The first, prepare mode, scans the specified import directory for files that it knows how to process and creates one or more file listings, with the number of files in each list controlled by another argument, `batchsize`.
- The second, index mode, takes the path to a file listing as an argument, along with an optional string given as the argument `prefix`, and generates an index in the output directory. This prefix allows multiple simultaneous imports, with each resulting in a unique index by way of appending the prefix followed by a period to the index's file path.
- Finally the third, merge mode, finds all index directories with prefixes in the output directory and merges them into a single, non-prefixed, index using the *StructureMerger* classes included as part of Terrier.

As well as allowing a serial batch import similar to Greenstone and DSpace, our custom file indexer contains enough functionality to support parallel processing as explained in the following sections.

To be operational in the experimental environment developed in this thesis, implementation work also required several new document types to be supported by Terrier and so various classes adhering to the Terrier interface for documents were implemented, including *ImageDocument* for importing images and generating thumbnails, and *VideoDocument* that performed a Terrier compatible version of the video processing as detailed in Section 3.4.4. The recurring Java issue with standard output and error stream buffers was encountered and addressed.

Ensuring that the collection has imported successfully and was correctly displayed required some minor changes to the default Terrier web interface. Specifically, support was added for display images and their thumbnails, and videos with a number of key-frame images displayed and links to the original and streamable video formats.

4.3.1 Terrier using OpenMPI

The customised indexer described in the previous section was implemented with the further goal of operating in parallel. Figure 4.7 shows how the custom file indexer implementation was mapped into the SPMD model.

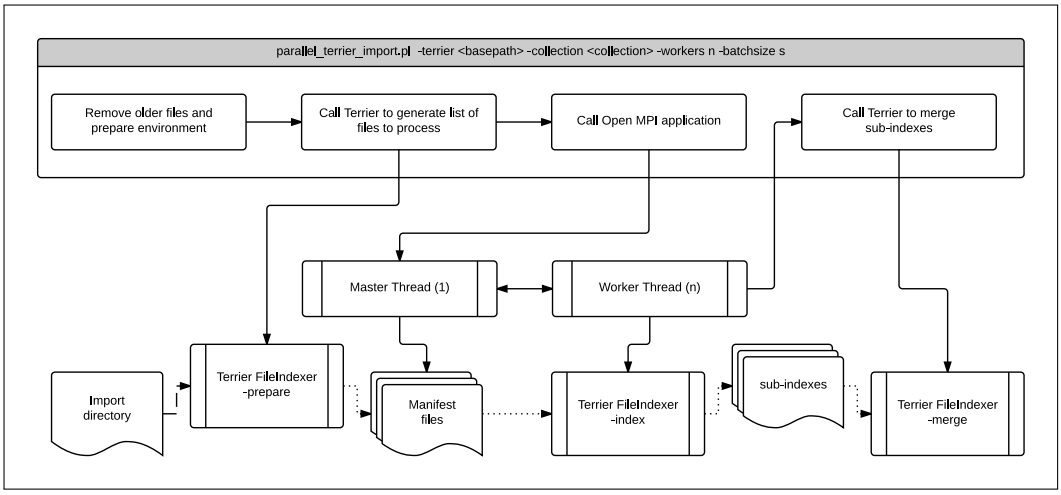


Figure 4.7: Parallel Terrier import process in Open MPI

It was thus possible to write a C++ application that made use of the three manifestations of the customised indexer to import the documents in parallel. In particular, the application could assume that the unique count included in each file listing's name was to automatically be used as the prefix for each individual index, allowing for multiple indexes to be generated in parallel and left to be merged in a final pass. The process flow was as follows:

1. A top-level Perl script prepares the environment, clears any existing index files, and flushes the disk cache.

2. The FileIndexer is called in "prepare" mode to create several numbered file listings containing the file paths for the batch of documents to be imported.

3. The Open MPI application is executed passing in the total number of batches created.

4. The master thread instantiates each worker thread passing through the filename of the file listing to be processed.

5. The worker thread once again calls FileIndexer but this time in "index" mode, stipulating both the path to the file listing to process, and the prefix to use so as to match the numbered manifest. Each document in the batch is imported, using the appropriate document-type implementation for processing.

6. Parallel processing continues until all files have been imported, whereupon the Open MPI application exits.

7. Control is returned to the top-level script, which immediately calls FileIndexer one last time, this time in "merge" mode, so as to combine all of the worker produced indexes into a single index.

Note that the merge phase was not written so as to allow parallel processing. As the act of merging two indexes is independent of other merges, this stage would benefit from parallel processing as well, although this would have required complex logic within each worker or a second Open MPI application designed to take a list of index names and/or prefixes and recursively merge them. Instead the merge step was keep simple to more closely reflect the processes used in the other two digital library implementations.

4.3.2 Terrier using Hadoop

Terrier has built-in support for Hadoop utilising the MapReduce mechanism to allow for an efficient full-text index built in a single-pass on multiple processing cores. Utilising this support, along with careful configuration of Reducers and replication factor, resulted in significant decreases in elapsed indexing time [129]. However, to be compatible with experimentation of the other digital library systems and present a generalised approach, the work presented in this thesis did not make use of this feature. It added a new light-weight implementation that utilised Hadoop in the role of parallel scheduling manager, similar to the process flow in DSpace, which did not fully leverage Hadoop's features. Unlike in Greenstone a more complete Hadoop solution was not implemented as it would need to be coded specifically for, and tightly coupled to, Terrier.

The process flow is very similar to that shown for the basic Hadoop implementation in Section 4.1.5, but with several changes to support Terrier:

1. A top-level Perl script prepares the environment, clears out older indexes, and flushes disk caches.

2. The import directory is scanned and a listing of suitable files written to HDFS.

3. Hadoop is launched to run the Hadoop-aware application.

4. The application reads in the list of files using a FileInputFormat implementation, with each file found being treated as a *split*.

5. One or more Mappers are initiated with each presented a single split to process.

6. The Mapper writes the file path to be processed as a file listing and then provides this to the index mode version of the Terrier FileIndexer. Each manifest file processed by a Mapper generates a distinct sub-index.

7. Any output from the map phase is passed, unmodified, to the reduce phase and is—for all intents and purposes—ignored.

8. Once all the splits have been processed by Mappers the Hadoop framework returns control to the script.

9. The top-level script calls the FileIndexer one last time, this time in merge mode, to combine the multitude of indexes into one.

10. The import is complete once all sub-indexes are combined into a single index.

One significant difference from the Open MPI implementation is that the second step, whereby a listing of suitable files is prepared and stored in HDFS, precludes the use of the FileIndexer in prepare mode. This implementation also did not allow for experimentation with batch size as each split is a single file from the initial listing. Further speed increases may be possible by re-implementing merge as a parallel process. For example, HadoopTerrierIngest could have been extended by customising the hidden sort phase of Hadoop—where the outputs from the map phase are sorted and merged—to also merge the sub-indexes.

4.4 Discussion

As evidenced in the above flowcharts and descriptions, this research shows that it is possible to extend a number of general purpose digital libraries with parallel processing capability. While there are some differences due to the way each library imports documents, there are also commonalities between the three import processes regardless of framework. All implementations fit within the broad Single Program Multiple Data (SPMD) model, and could be arranged to have a similar sequence of steps with file costs in terms of temporary and manifest files. While a complete Hadoop implementation was only completed for Greenstone, and so stands-alone from Map-phase only DSpace and Terrier

implementations, with further work it would be possible to have completely common process for all three digital library and over both frameworks.

Specifically, all parallel processing implementations supported the documents types found in the corpora for this research, and exposed important configuration options to allow for experimentation. Using command line arguments each import could be run with a varying number of worker threads, and using batches of varying size. Where applicable, the replication factor of distributed files systems could also be controlled.

Experimentation showed that all of the digital libraries tested showed some improvement from parallel importing. However the extent of this benefit was heavily dependent on the type of documents being imported. As will be shown in the next chapter, textual documents with very little processing performed during import only experienced a slight speed-up and were bound by file transfer limits. In contrast, Video documents undergoing significant import processing exhibited speed-ups in line with those predicted as optimal by IBM's experiments on Java threads [77].

The practical implementations were deemed complete enough to proceed with the next step of the research, namely that of performing a set of experiments to capture information on certain key features of parallel processing so as to build an accurate predictive model.

Chapter 5

Experimental Results

> After Mike McCandless increased the limit of unique words in a Lucene/Solr index segment from 2.4 billion words to around 274 billion words, we thought we didn't need to worry about having too many words. We recently discovered that we were wrong!
>
> — *Tom Burton-West, "Too Many Words Again!", HathiTrust*

This chapter details the experiments conducted in order to gather information used in the creation of the predictive model. It is divided into three sections. The first contains a number of serial import experiments over a range of document types, exploring the relative performance of several general purpose digital libraries and provides compelling evidence of the need for alternate methods for creating very large-scale digital libraries. The second section contains the results of parallel processing experiments, repeating the import of the aforementioned document types, and is key to this thesis. Finally, the third section contains the results of additional experiments, carried out during development of the parallel implementation. These experiments are not essential to support the central hypothesis, however are chosen for presentation as they exhibit properties that are of interest to the field of large-scale digital libraries specifically, and parallel processing in general.

Experiments run using Greenstone (agnostic to the database backend and indexing tool), unless otherwise stated, were configured to be Lucene and a version of TrivialDB customised for clusters respectively. This combination is the only configuration available across the range of hardware. Depending on the experiment, the hardware in question is either a single, modern, eight-core computer, or a Beowulf cluster [19] formed from sixteen older pentium computers.

5.1 Serial Importing

Serial importing is the focus of the initial experiments in this chapter, designed to show the relative performance, in terms of time elapsed, of the three chosen digital library software systems when importing increasing numbers of documents. Each combination of number of documents, document type, and digital library system was repeated several times to limit the effect of outliers. In order to meaningfully compare the actual processes undertaken by the three digital libraries, and given that a collection built in Terrier results in a searchable index, both Greenstone and DSpace included a minimal indexing phase, albeit run in serial. In this way all three digital libraries reach the point, after import, of containing documents along with a full-text searchable index.

These experiments were conducted on a modern multi-core computer and so care was taken to ensure only a single processing core was utilised. This was achieved through the use of the Linux tool *taskset*, used to indicate to the operating system which core or cores to run a process on. Given how modern operating systems operate, this is best described as a suggestion—something the operating system will attempt to obey, balancing the other running processes notwithstanding—and is referred to as *processor affinity*. In running other experiments, there were few network resources in use, aside from client-server communication traffic in those database configurations utilising a server daemon, and thus file input/output costs were mostly due to local hard-disk access. Disk synchronisation and flushing were used to mitigate the effect of caching. Despite this, there will still be some speed-up due to file caching as they may be read several times during the import process, and subsequent access attempts will benefit from the initial access.

5.1.1 Serial import of text documents

The first experiment shows the relationship between the number of documents and import time during the normal serial import of text documents. The collections were created from Lorem Ipsum text specifically modelled to reflect the measurements characteristic of text that has resulted from an OCR process (see Section 3.4.5). The collections ranged in size from ten documents through to one million documents.

In this experiment the processing of text files during import was kept purposely lightweight, with no metadata beyond file size being extracted. Terrier

exhibits this property by default, but for Greenstone this required configuration to avoid automated detection of file encoding and hashing functions involved in automatically assigned unique identifiers, while for DSpace this required the provision of small, manually created, Dublin Core metadata files.

Table 5.1: Elapsed time of serial import of Lorem Ipsum text collection as number of documents increase

Count	Greenstone (seconds)	σ	DSpace (seconds)	σ	Terrier (seconds)	σ
10	4	1.29%	13	2.32%	6	6.33%
100	13	0.80%	48	4.44%	12	2.56%
1K	117	1.34%	407	6.29%	88	2.82%
10K	1,019	1.21%	3,873	6.19%	696	3.58%
100K	14,160	1.13%	39,101	6.79%	6,058	2.42%
1M	310,972	0.47%	330,900	2.44%	54,498	1.13%
5M	1,602,543	-na-	*1,700,00*		*300,000*	
10M	3,207,209	-na-	*3,400,000*		*600,000*	

The result of this experiment is shown in Table 5.1 and visualised in Figure 5.1. The main findings of this experiment are:

- **Overall Terrier has the best import performance as number of text documents increase.** Terrier IR displays the best scaling performance, with timings one-sixth that of Greenstone and DSpace for larger collections. While DSpace's performance is initially inferior, it and Greenstone become comparable as Greenstone experiences a notable slowdown around the one hundred thousand document mark.

- **The total elapsed serial import time is approximately linear to the number of documents.** All three digital library software systems exhibit linear performance as the number of documents increase. Thus, this trend appears common to general purpose digital libraries regardless of the technology used for back-end storage. This is a confirmation of experimentation detailed earlier in Table 4.1 and provides evidence that serial import of larger scale collections will inevitably become impractically expensive in terms of time.

There is evidence that both Greenstone and DSpace encounter a point where performance drops off rapidly. For the former this occurs somewhere between one hundred thousand and one million documents, while DSpace experiences this effect somewhat earlier between ten thousand and one hundred

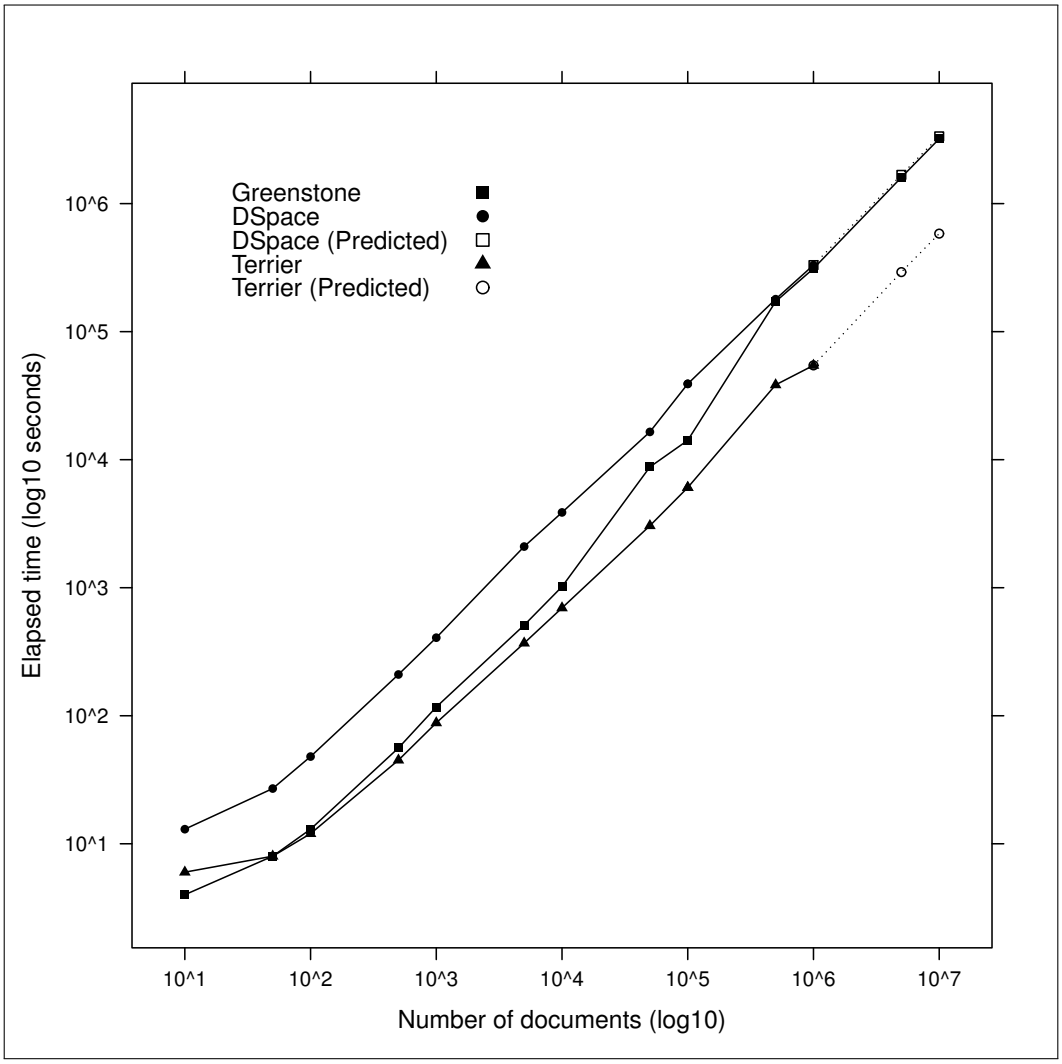


Figure 5.1: Serial import of Lorem Ipsum text collection as number of documents increase

thousand. After this event, however, the performance of the two digital libraries mirror each other. As the experiments below will show, this appears to be related to the sheer number of files involved rather than any feature of processing load or even bytes processed. As this effect is not seen in Terrier IR, which does not employ a database, this threshold may be caused by some property of the underlying database technology once a certain number of records is reached.

Table 5.1 includes indicative timings for five and ten million document collections: sizes on a par with the larger collections mentioned in the case studies in Section 2.2. These estimates were extrapolated from the other results using linear regression forecasting as running multiple experiments of this scale proved difficult. A single import of a ten million document Lorem Ipsum collection, for example, would have taken more than forty days to import using DSpace—assuming there were no operating system glitches requiring

the process to be re-run.

One further observation, as exhibited by the relative standard deviation measures in Table 5.1, is that elapsed timings from larger sized collections generally showed less variance than the smaller sized ones. DSpace showed the most overall variance, while Greenstone was the most consistent. Terrier exhibited the greatest decrease in variability as document number increased. There are two likely reasons for this. Firstly, the influence of indeterminate operating system factors is far more pronounced in collections with less total elapsed time. One second of variance in the time it takes for a database connection to close has a greater effect on an import that takes several seconds as compared to one that runs for several hours. Secondly, the input documents exhibit significant variance themselves as detailed in Table 3.1; they are quite different in terms of total number of characters, words, and unique terms. Thus, as the number of such documents in the collection grow larger they are more likely to approximate the average document and consequently reduce the disproportionate affect of outliers.

5.1.2 Serial import of multimedia files

The second experiment has a similar configuration to the first, however considers the serial import of multimedia files and targets the velocity of data rather than the volume. The files for this experiment are sourced from the Greenstone ReplayMe! system, with each file being a ten minute duration segment of MPEG2 Transport-Stream encoded video [191]. The experiment records the serial collection import time, over the three chosen digital libraries, as the size of the collection increases from a single video segment, through sixteen segments representing a ten minute period of recording across New Zealand's free-to-air digital channels, to a total of 96 video segments accounting for a complete hour of recording. A linear regression forecast is then used to predict how long a full day of ReplayMe! material, over two thousand segments, would take to process.

For this video collection to be usable in real-time, there is an implicit requirement that the elapsed time taken to import a ten minute segment must be ten minutes or less. The inability to achieve this requirement would mean that the digital library would become progressively more out-of-date.

The results for this experiment are shown in Table 5.2 and Figure 5.2. Unlike in the text collection, the results for the three digital library software are

Table 5.2: Serial import of ReplayMe! video collection as number of documents increase

Count	Greenstone	σ	DSpace	σ	Terrier	σ
1	831	1.43%	838	5.17%	701	3.74%
16	13,263	0.77%	13,357	2.97%	13,190	0.02%
32	26,530	0.38%	26,638	2.22%	27,064	0.04%
48	39,893	0.43%	39,937	2.02%	38,167	0.94%
64	53,070	0.26%	53,036	0.99%	50,940	0.75%
80	66,292	0.25%	66,570	1.21%	63,635	1.16%
96	79,726	0.27%	79,896	1.24%	75,889	0.79%
2,304	*1,911,959*		*1,915,685*		*1,814,234*	

more closely matched. Terrier has the smaller elapsed times as the number of video segments increases, processing in approximately 95% of the elapsed times of either Greenstone or DSpace. Greenstone and DSpace exhibit similar performance, with elapsed times often overlapping when taking variance into account. Also note that the chart again displays a linear relationship between number of documents to process and total processing time elapsed; however, this trend is more noticeable than in the text collection example. A third point of interest is that the variability in elapsed times is also generally lower, notwithstanding the smallest collections as explained in the previous experiment.

These three points can be explained by the fact that there is significantly more processing load directly related to the video files. Indeed this cost dwarfs any overhead introduced by the digital library software. Thus it can be concluded that the processing load incurred in importing the documents themselves is an important predictor of processing time and, by extension, the potential benefits of parallel processing.

Finally, note that at no point do any of the digital libraries achieve real-time processing. Video processing at real-time offers a scale challenge not easily addressed with serial processing. As with the previous experiment, Table 5.2 includes estimates for a larger scale collections—in this case the total number of video segments recorded in a single day—predicting it would take twenty days to process in serial on this particular computer. It is exactly this soft of situation—in this case real-time (or better) processing of the video documents—that parallel processing gains are possible.

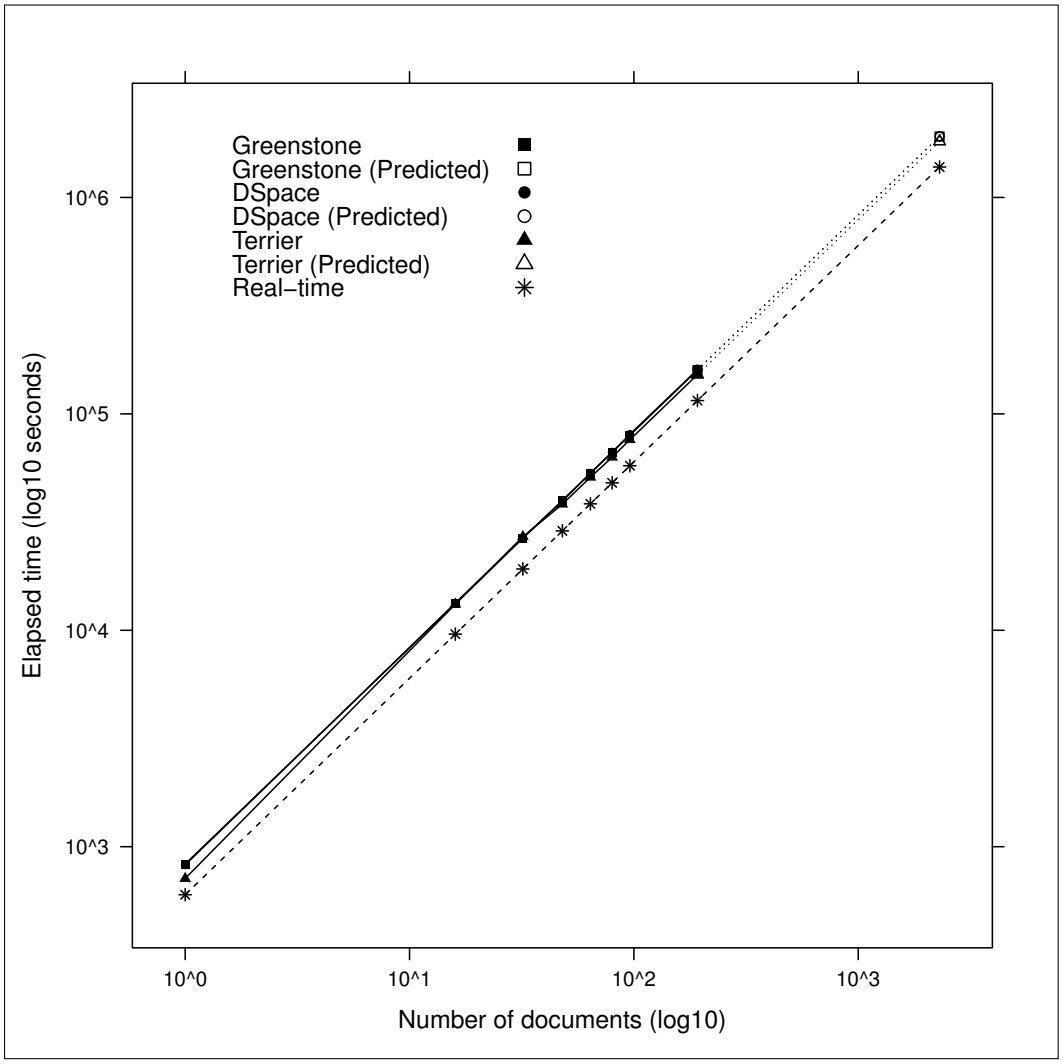


Figure 5.2: Serial import of ReplayMe! video collection as number of documents increase

5.2 Parallel Importing

This section details several experiments involving parallel processing and focuses on the effect on the total elapsed import time of an increasing number of processing threads over a fixed number of documents. The goals of this series of experiments are to show that:

- Parallel import has been successfully implemented across a number of general purpose digital libraries,
- Increasing numbers of processing threads decrease processing time,
- The magnitude of the time decrease is heavily influenced by the processing cost of importing each document, and

- There is a break-even point beyond which adding further processing threads does not improve performance.

Again, the experiments shown below were run on one of two hardware configurations:

- Multi-core made use of a modern eight-core (4-CPU, hyper-threaded) desktop computer. Where necessary, tools were used to assign processes to run on certain cores (set affinity) or to flush the file caching system to an initial state.
- Cluster involved a Beowulf [19] cluster with a total of fifteen commodity computers. For these experiments the head node of the cluster was not utilised as a worker node as it is responsible for network file transfers (via NFS) and we wished to minimize interference between digital library processes and system level processes. Thus the maximum number of available compute nodes was fourteen.

As a reminder, the parallel processing work published by IBM [77] suggests that the optimal performance for processing intensive tasks should occur when the number of worker threads is one more than the number of compute nodes.

5.2.1 CPU Utilisation

The CPU Utilisation experiment was designed to expose the potential benefits of running a parallel import on a modern, multi-core, computer. This experiment involves running a Greenstone import of the plain-text Lorem Ipsum collection on an eight-core computer. The Linux tool `mpstat` was polled to determine the processing load on each core every second over a period of four minutes, this being the total elapsed time taken to perform the typical serial import. The load is expressed as a percentage. There were three configurations experimented on:

- A serial import process allowing the operating system to move the process between cores;
- A serial import process attempting to restrict processing to a single core; and
- A parallel import process supported by the Open MPI framework.

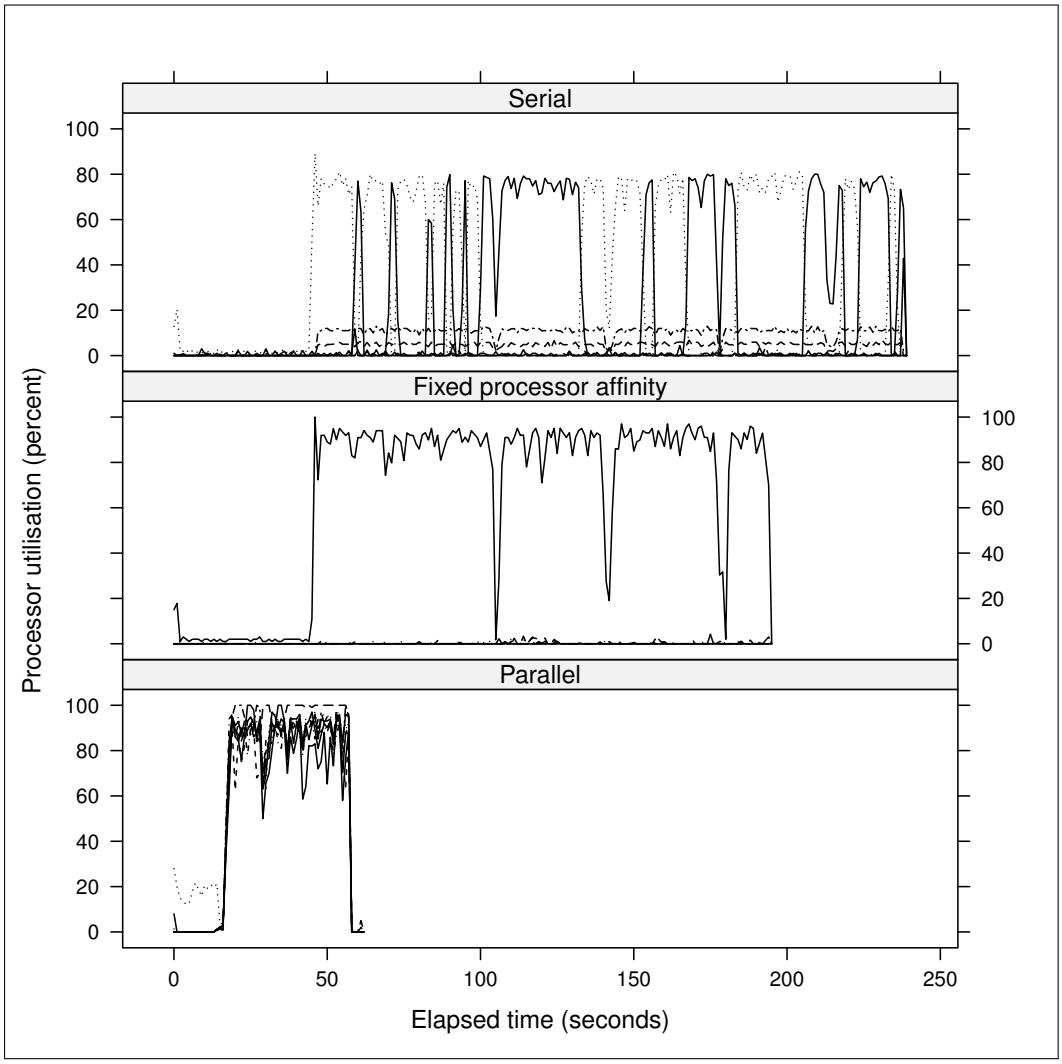


Figure 5.3: CPU utilisation on eight-core computer during serial, serial with fixed processor affinity, and parallel import

The second configuration once again made use of `taskset` to restrict the process to a single processing core. Care was also taken to reset the system environment, including disk cache, between each experiment run.

A typical serial Greenstone import is shown at the top of Figure 5.3. An initial burst of processing (peaking at 20%) is followed by a quieter period, as Greenstone performs file input/output searching for metadata and determining which files to process, and then the process uses around 80% of a processing core for the remaining time. While activity is evidenced on several cores, what is being observed in the chart is most likely the operating system switching the processing load between cores as a side effect of context-switching in the multitasking kernel. Practically, the majority of the cores exhibit low utilisation or none at all. Another observation is that there are clear troughs in processing load whenever the processing moves between cores; as explained during the next result. The serial import took 240 seconds elapsed time to complete,

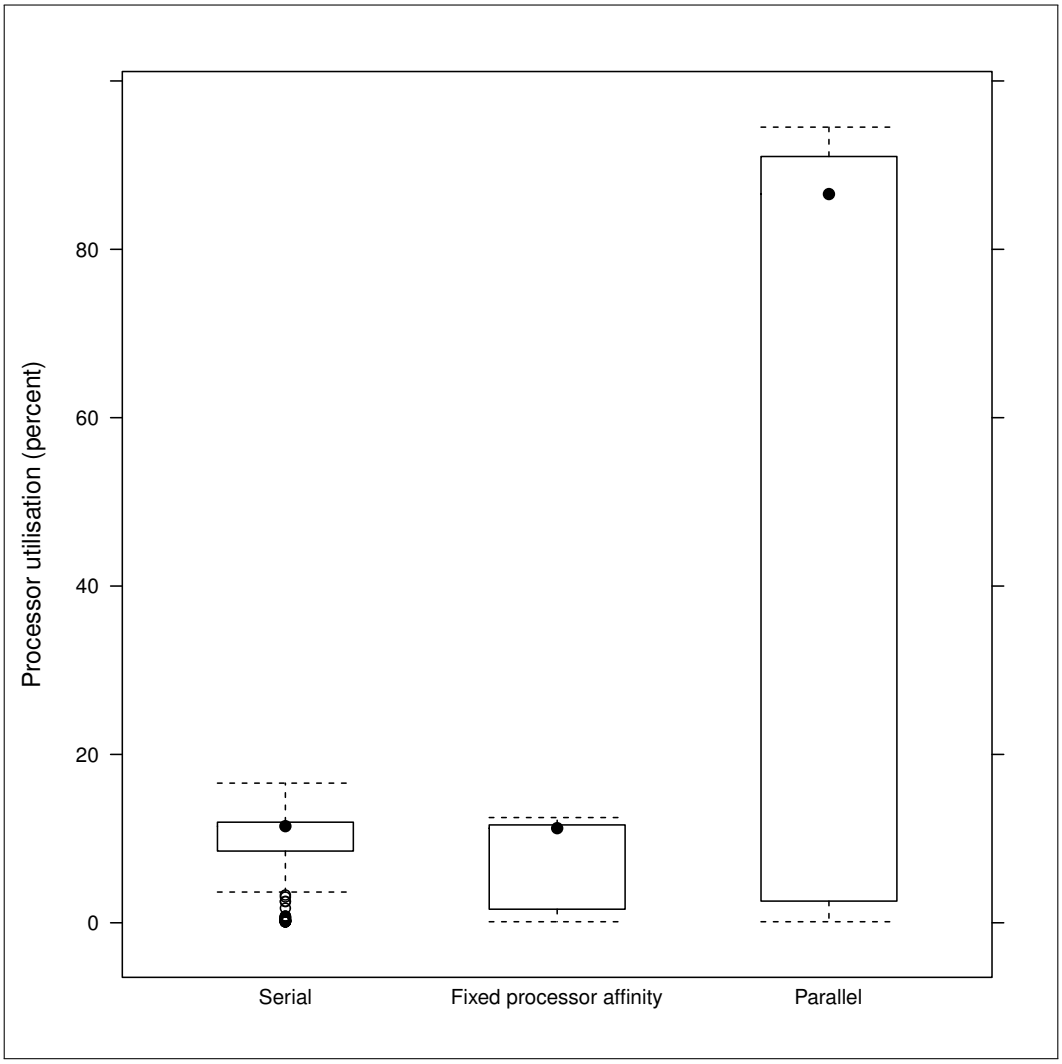


Figure 5.4: Summary of CPU utilisation on eight-core computer during serial import, serial with fixed processor affinity import, and parallel import

during which time the average load on the eight processing cores was calculated to be approximately 9%.

The fact that the serial import is only utilising a single processing core is somewhat obscured by the operating system switching the process between cores in order to try and balance load over the entire computer. The second experiment tries to clarify the properties of a serial import by fixing the affinity of the serial import to a single processing core. The middle chart in Figure 5.3 shows the result of applying a processor affinity indicating the process should run on a single core. The general shape is the same as the serial import above, but now it can be seen that the other cores are barely utilised. That there is any load shown on the other cores at all may be explained by the fact that processor affinity is only a suggestion—the operating system may still allocate other cores for processing but should be biased towards running the process on the specified core. Of immediate note is that the elapsed time has decreased

to 195 seconds, illustrating that there is an overhead cost to processing on multiple cores—in other words the cost of context switching. Whenever the process switches between cores, the state of memory must be stored and then restored on the destination core. The serial import process above incurs all the costs of running multiple cores but with none of the benefits of parallel processing. By restricting the process to a single core, the operating system avoids much of the costs of multiple cores resulting in a saving in elapsed time. Figure 5.4 shows that, while the single core selected using affinity is now busier at 86% utilisation, the average processor core utilisation over all eight cores for the duration of the import actually dropped to approximately 8%.

Finally, we consider the results of a parallel Greenstone import, as shown in the lower chart in Figure 5.3. The benefits of parallel processing are immediately apparent: the elapsed time for the import has decreased to 61 seconds. The general shape of the result is also distinct from the previous two experiments. At the start there is longer period of activity on a single core. This represents the master processing thread establishing the worker threads and assigning work to them. Next, there is a much smaller period of low processing load during which time Greenstone searches for metadata and processable files. This is due to most of that work now being distributed to the eight worker threads. Next, the processing work of importing the collection now utilises all eight cores. The average processor load during this period is 88%. Unique to this chart is a final phase where a period of low processing load is followed by one final, small, spike. This phase represents the master thread ending the worker threads and writing the final Greenstone files. These files can only be written once the parallel stage is complete. This, along with an initial scan of the collection to be imported to create a manifest, are thus done in serial. Even with this balance between serial and parallel components—see Amdahl's Law [8]—this experiment exhibited the highest average processor load over the eight cores for the entire duration of the import process, approximately 58%.

5.2.2 Parallel import of text documents

The parallel import of text documents experiment considers the import of a collection of plain text Lorem Ipsum documents. Lorem Ipsum was chosen as it was practical to generate in large quantities and in such a way as to closely resemble the measurements of english text extracted by optical character recognition (Section 3.4.5). For the metrics and measurements of the collection review Table 3.1. The experiment involves importing a collection of one million such documents, carried out on each of the three digital libraries,

utilising the Open MPI frameworks, and run on a cluster of 15 nodes total. The number of worker threads was varied from zero—essentially a serial import—through to 28 threads. This upper bound is double the number of compute nodes and results in an average of two worker threads per node, although the actual distribution of threads is left up to the parallel processing framework and so may vary from node to node.

In general, a digital library built on plain text files undergoes very little processing during import, but in order to gather a range of information three differing configurations were used to import text at three distinct processing levels:

- **Minimal processing load** - which entails the text being copied from the input file into the applicable file format—interim or otherwise—for the chosen digital library. DSpace performs lightweight processing of text by default, whereas Greenstone and Terrier were both configured to do so.
- **Typical processing load** - which extends the above by also putting the text through a series of functions to extract measurements, such as the Flesh-Kincaid readability score [100], and encrypting random paragraphs, as might happen with documents containing sensitive and secure information.
- **Intensive processing load** - which performs both of the above but further augmented the metadata with key-words as extracted by relatively expensive Weka data mining algorithm [91].

By exploring a selection of processing loads the intent is to measure the influence of the balance between processor load and file transfer cost on the potential performance gain offered by parallel processing.

The results of measuring the elapsed time of parallel import as number of worker threads increase is shown in Figure 5.5. The main thing to note is that increasing the number of worker threads does lead to a reduction in processing time. All three digital libraries and all three processing loads follow the same trend, reaching a local minima approximately 15 worker threads. Accounting for the master thread, these numbers are close to IBM's suggested optimal point. However, rather than the addition of further worker threads resulting in a decrease in performance—due to the significant overhead of over-subscription, where multiple worker threads on a single node—the experiment indicated that adding further threads garnered further performance increases.

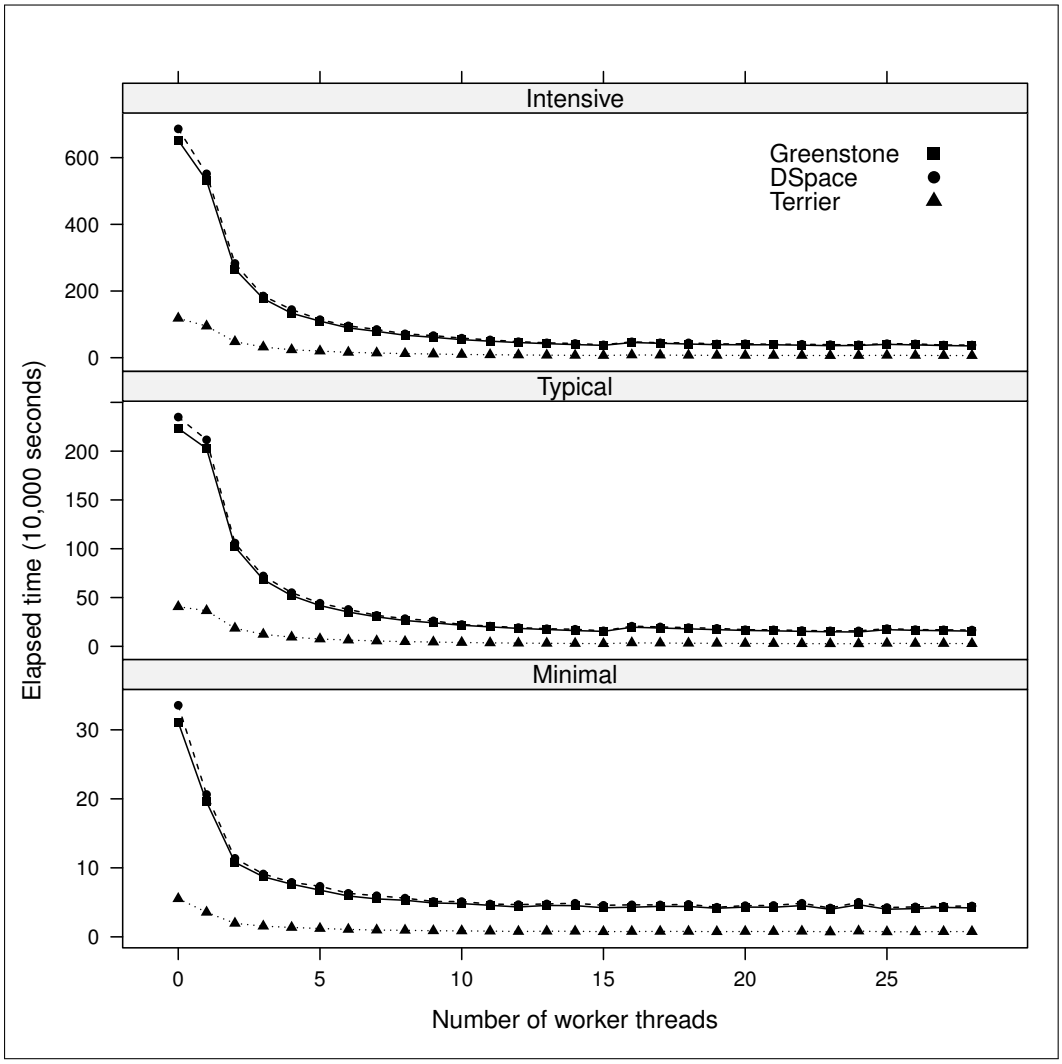


Figure 5.5: Elapsed time of import of text collection as number of parallel worker threads increase

This suggests that the import is more likely I/O bound than processor bound; if worker threads were stopped waiting on I/O the spare processing power could be used to oversubscribe computer nodes resulting in a slight decrease in import time. This result for low processing loads is unsurprising given that text import has significantly lower processing costs than other media types.

Consider the minimal processing load collection where, at the first minima point, the elapsed time of parallel processing is 13.56% of the elapsed time of a serial import. Compare this to a naive prediction of performance, which simply divides the total elapsed time by the number of worker threads. It would predict a performance approaching 6.67% of serial. This prediction assumes the import is completely processor bound, and thus the difference between the naive prediction and actual performance is due to interaction with the operating system, most likely the cost of file transfer being the limiting factor.

For typical processing load, the performance at fifteen workers is 6.86%, very close to the naive prediction. In the case of intensive processing load parallel processing performance surpasses the naive prediction, reaching performance of 5.69% of serial import time. Such a performance is only possible if the use of the parallel framework is somehow decreasing some other cost—such as file transfer cost—over-and-above dividing the cost of processing.

One final point of interest is the difference in performance between a serial import and a parallel import with a single worker thread. In theory, for a processor load bound import, the worst performance should be encountered when worker threads equals one, as this case has all of the overheads of the parallel processing framework but with none of the benefits. Figure 5.5 exhibits no such feature. On the contrary, by utilising the parallel framework when processing text file, the results indicate an elapsed time decrease of approximately 20%. As mentioned above, the suspected cause is that the task is not processor bound, and thus the file transfer cost is somehow co-incidentally mitigated by the use of the framework.

5.2.3 Parallel import of image documents

The parallel import of image documents experiment repeated the previous one but with a key difference: the collection to be imported is made from one thousand random images that utilise a mixture of compression techniques. The images, drawn from the Wang Corpus [195], are predominantly in JPEG format but include a number of GIFs and PNGs as well. Typical image collections make use of image transform or metadata extraction techniques that are significantly more processor intensive than those common to text collections. In order to reflect this in the experiment, each image in the collection was processed by several functions from an image library to extract identifying features and had a thumbnail preview generated. Calls to underlying libraries were specifically configured to occur on a single processing core—many modern image processing libraries can utilise multiple cores or even delegate the work to video processing hardware.

Once again, three configurations were explored so as to measure differing loads of processing cost:

- **Minimal processing load** - which copies the images into the collection adding the file path and mime-type as metadata.

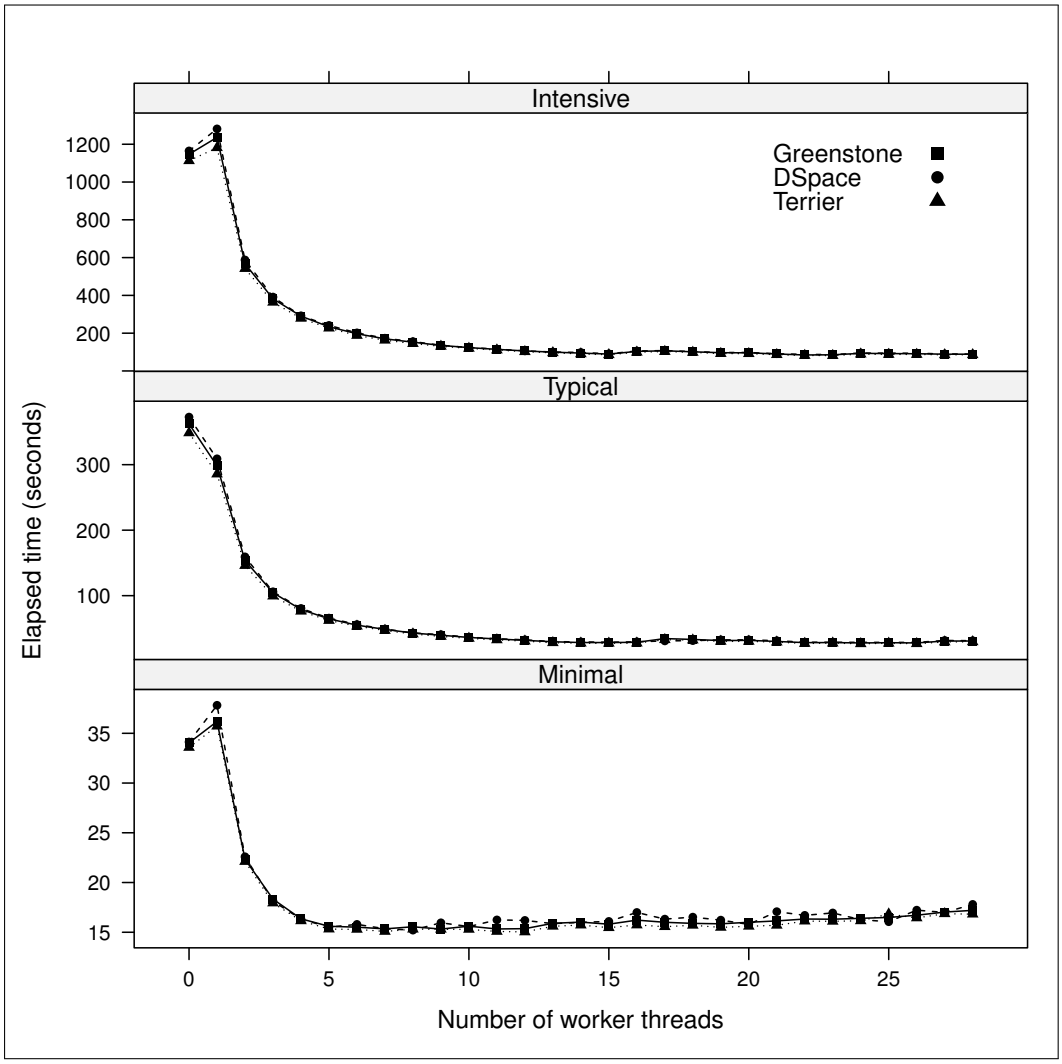


Figure 5.6: Elapsed time during parallel import of image collection as number of worker threads increase

- **Typical processing load** - which built upon the above by generating a thumbnail-sized preview of the image. This reflects the most typical image collection configuration.
- **Intensive processing load** - which greatly increased the processing cost by using machine learning vision tools to extract SIFT features [119] from the images.

Figure 5.6 shows the result of parallel importing the image collection on the same cluster configuration as before. Results for all three digital libraries are shown. They are broadly similar in terms of performance, as is to be expected as all make use of the same image manipulation processes.

The chart exhibits the same trends as present in the previous text collection but with three notable differences:

- Common to the three processing loads, all three digital libraries have similar performance—with Terrier generally faster when importing. Due to the number of documents involved, neither DSpace nor Greenstone are in danger of exhibiting sudden increases in elapsed processing time as encountered in the previous experiment.

- For the intensive processing load experiment, moving from a serial import to one with a single worker thread now results in the predicted initial decrease in performance due to the extra overhead of parallel processing with none of the benefit. This is due to the heavier processing load involved in importing images; despite the individual files being larger (in bytes) the overall file transfer cost is lower and there is less chance that it is unduly influenced by the change to a parallel processing framework.

- For this same chart, the performance increase at fourteen worker threads is a local minima and completes in approximately 8% of serial elapsed time. This is closer to the naive prediction garnered by dividing the total elapsed time by the number of worker threads, indicating the process is likely processor bound and would greatly benefit from parallel processing.

While the chart exhibits a slight trend of decreasing performance after the predicted optimum number of threads, there is not a single clear minima, and for typical and intensive processing loads there was a second, faster, performance reached around 23 worker threads again.

5.2.4 Parallel import of audio files

The third in this series of experiments repeated the parallel import set-up outlined above, but as applied to a collection of 96 audio (music) files in WAV format [65]. The typical processing involved in transforming and extracting metadata from audio files is another step up in terms of cost, and so was expected to exhibit better performance under parallel processing. The three processing configurations tested in this experiment were:

- Miminal processing load, which copied the WAV file into the collection extracting minor metadata using the *FFMPEG* [21] tool.

- Typical processing load, which performed the same tasks as the first load, but also converted the audio file to FLV format suitable for web streaming.

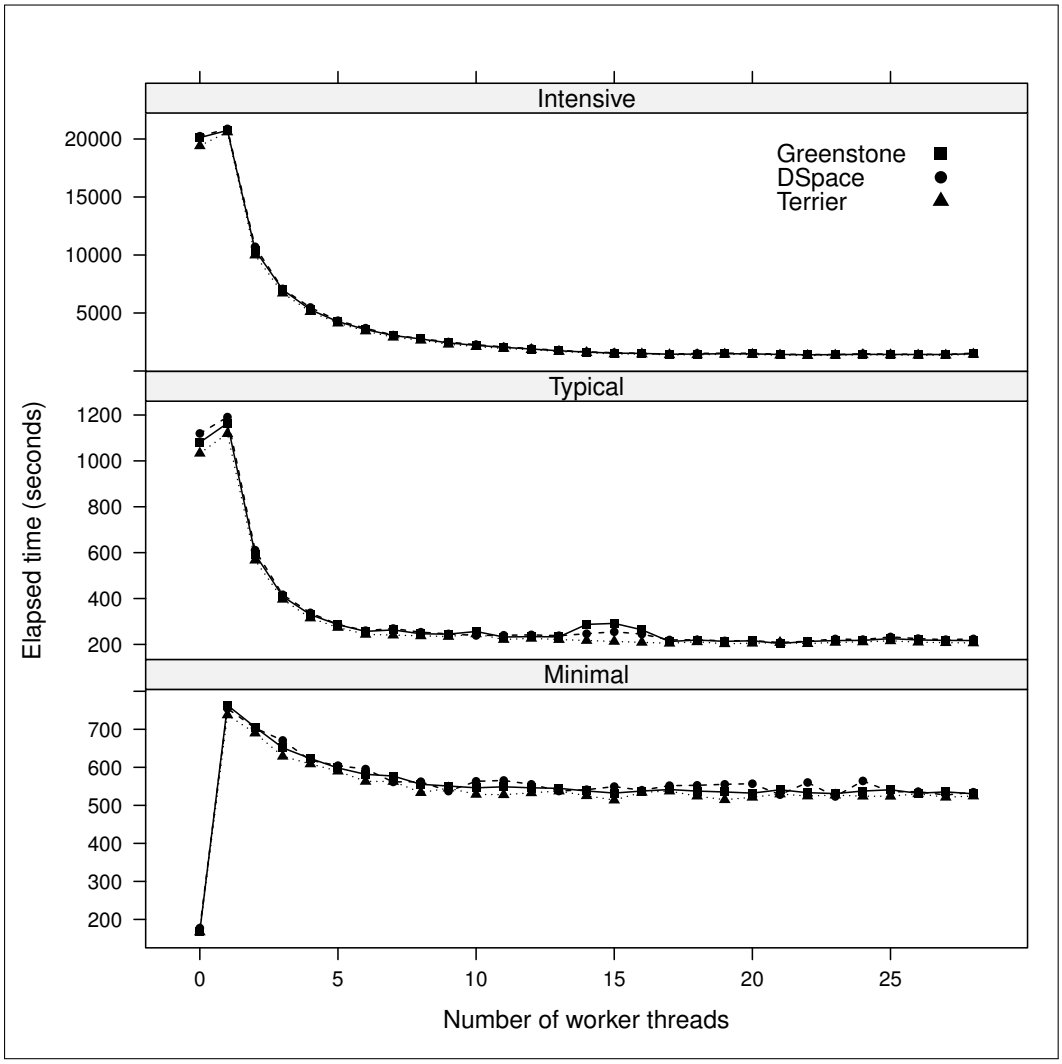


Figure 5.7: Elapsed time during parallel import of audio collection as number of worker threads increase

- Intensive processing load, which performed metadata extraction and FLV conversion, but also used the FFTW algorithm [68] and other third party tools to calculate music information retrieval features.

The results shown in Figure 5.7 reflect that increasing the number of worker threads decreases the time taken to import the collection. All three libraries exhibit the initial expected increase in time as the overhead of parallel processing comes into effect, followed by a substantial decrease in processing time up to a local minima point. For the first time, and common to all three processing loads, once the number of workers exceeds 24 there is a gradual but constant decrease in performance. Both typical and intensive processing loads reach an optimal processing time around fourteen worker threads, although the typical load exhibited substantial differences in timings between the three libraries; the cause for this could not be traced.

The minimal processing load chart stands out, and is unique amongst the experiments run. In this experiment alone the serial processing elapsed time was far better than any parallel processing elapsed time regardless of the number of threads added. This was due to a combination of the size of the files and the low level of processing needed for each document. Reviewing the system logs revealed that the NFS server experienced substantial contention between the various compute nodes causing delays in writing files. This can be seen either as evidence that some collection configurations are optimally imported in serial or, alternatively, an argument for the use of better distributed file systems when parallel processing (Section 3.4.3).

5.2.5 Parallel import of multimedia files

The final in this series of experiments, parallel import of multimedia files, repeated the set-up above but measures performance increase when applying varying numbers of worker threads to the task of importing video files. The video files are those sourced from the Greenstone ReplayMe! system, with the other two digital libraries being configured to process the video in as similar a way as possible. Due to the significant processor load posed by transforming the video—including the extraction of high value key frames by use of artificial intelligence techniques—it is expected that the performance between the three libraries will be similar. Further, the optimal worker thread number should be fourteen, and the best performance should approach the naive prediction of 7% of serial elapsed time.

The three processor load configurations used during this experiment were:

- **Minimal processing load** - which copied the video files into place and used *MediaInfo* tool to extract metadata.
- **Typical processing load** - as above except but generated a web-streamable version of the video segment in the widely-supported MP4 format.
- **Intensive processing load** - which performs both of the above and also used *Hive2* [85] to automatically extract high-value keyframes from the video segment.

Figure 5.8 displays the results produced by running multiple imports at varying processor loads and over a growing number of worker threads. When

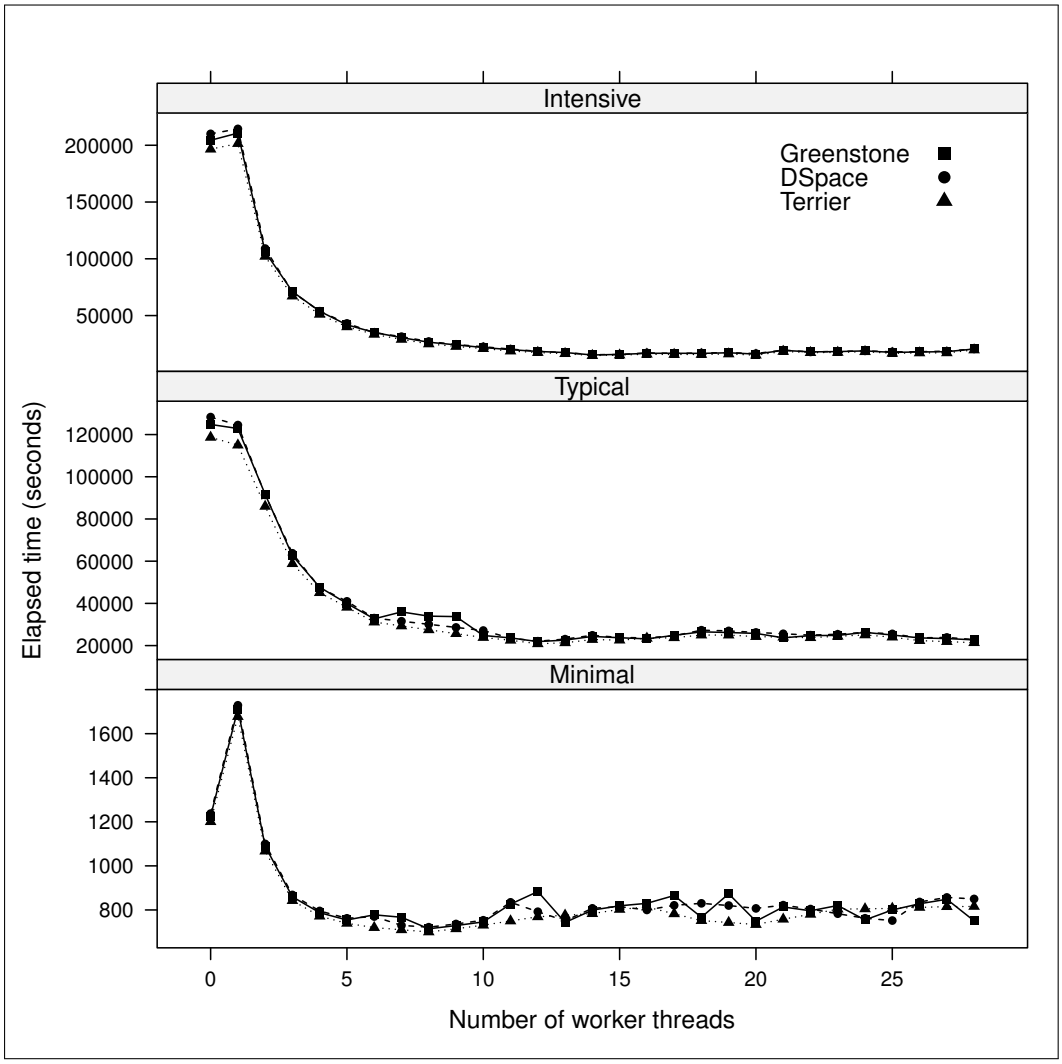


Figure 5.8: Elapsed time during parallel import of video collection as number of worker threads increase

intensive processing loads are involved in processing video files, the trend closely matched the outcome predicted by the IBM research. The global minima occurred when 14 worker threads are applied, and the performance at that point is approximately 7% of serial elapsed time. To put this in perspective, a video collection that would take 60 hours to import in serial would take a little over 4 hours on the 15 node cluster. There is less of an increase in the initial cost of applying parallel processing—especially in regards to DSpace and mostly likely due to the relatively low amount of file transfer required—but the shape of the chart exhibits the typical signature for a processor bound import when parallel processing is applied. For the intensive load, there is a clear (albeit slight) decrease in performance after 14 workers, while minimal load exhibits this increase in general although with more visible variation in times between digital libraries. Only typical load processing seems to lack a definitive trend, and only by looking at the numbers can we confirm a minima

at 12 worker threads.

The results for typical load are similar, but there is a second later minima at 25 worker threads. Unexpectedly absent is the initial increase in processing cost, along with an equally unexplained variability in elapsed time for Greenstone and DSpace occurring around 6, 12, and 19 worker threads. Neither re-running these tests, nor a review of the logs for these specific test runs, illuminated any obvious cause. Finally, while following the parallel import curve generally common to this battery of experiments, the results for minimal process load importing of video files exhibited more variation than the other two configurations. This is due, in part, to the disproportionate influence of the volatility of the underlying file system.

5.2.6 Hadoop Framework

The follow section presents an experiment comparing the two parallel processing frameworks considered by this research, Open MPI and Hadoop. While several prior experiments explored Open MPI's preformance, we now turn our attention to the Hadoop framework, and present the key findings from operating collection import experiments in this environment. As detailed in Section 3.4.3 there were several different interfaces between Hadoop and the underlying HDFS implemented. This experiment will compare Open MPI's performance to several HDFS interfaces, namely: the command line shell, the Thrift protocol with a single central server (located on the head node of the cluster), the Thrift protocol using one server per compute node, and the NFS proxy interface to HDFS.

One essential caveat is that the HDFS NFS proxy project is in its infancy, and this research's implementation never reached a stable state. Several times the import completely failed or NFS issues caused the cluster to stop responding requiring a system restart. Even when an import claimed to have completed the resulting parallel processing timing chart presented evidence that several video file segments had not been converted properly, as shown in Appendix B. Therefore the timings plotted in Figure 5.9 below were extrapolated by determining the average time taken to process the non-suspect video segments and then multiplying that by the expected number of videos processed by a single worker: in this case 7. This provides a ball-park estimate of time elapsed, but based upon several large assumptions and would most-likely change should a more stable implementation of NFS proxy become available.

The experiment essentially repeats the high processing load ReplayMe!

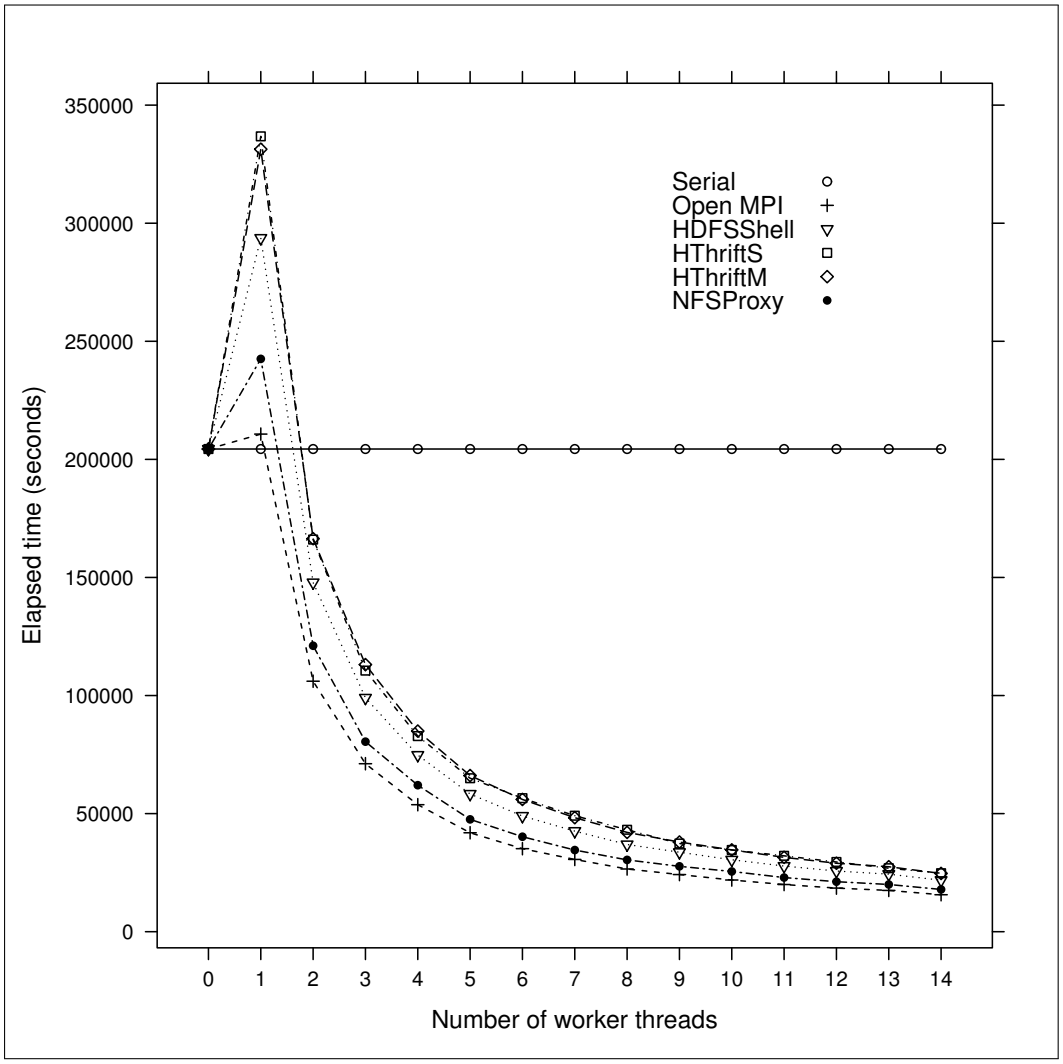


Figure 5.9: Comparison of Open MPI versus several Hadoop frameworks when applied to a Greenstone import of a video collection

video collection import shown in Section 5.2.5. This time the maximum number of workers is limited to 14 as over-subscription on a Hadoop cluster is a non-trivial configuration problem due to the specific hardware, Java resource constraints, and Hadoop's balancing mechanisms [156, 209, 190].

As Figure 5.9 shows, the general trend remains the same. The initial overhead cost of parallel processing is quickly absorbed by the growing number of worker threads. The chart does not exhibit the minima seen in the previous experiments without the inclusion of over-subscription, but given the similarities evident between Open MPI and Hadoop performance it would seem safe to assume such a minima does exist. The best Hadoop performance exhibited in the results shown is approximately 7% of serial import time, very close to the figure calculated from a naive prediction.

Of the Hadoop implementations, the one using the NFS proxy to the un-

derlying HDFS has the best performance, but remains suspect due to the stability issues inherent in that implementation. Direct calls to HDFS tools from the command line had the second best overall performance. This finding is somewhat surprising, as the command line requires repeated initializations of the short-lived Java instances. The alternate interfaces were implemented expressly to mitigate this, allowing for a single Java instance to be persisted throughout the collection import. However, the poorer performance of the Thrift variants suggest that the overhead cost of the Thrift protocol exceeded any performance gained by limiting Java instances. Indeed, the Thrift implementation utilising a single server had the worst performance, due mostly to the unintended bottleneck of a single computer handling all of the writing (database or otherwise).

The experiment above shows that, in this research, the Open MPI parallel framework offered better performance than could be extracted from the Hadoop framework. This is not something specific to Hadoop, but is due to limitations applied by the common SPMD approach. Moreover, Hadoop's performance was heavily influenced by the overhead of running Java on the (relatively old) compute nodes, compounded by a lack of appropriate interface to the underlying file system. We predict that alternate hardware and file system software, selected so as to be more appropriate for Hadoop and Java, could yield significantly better performance.

5.2.7 Compute Nodes

In all of the parallel processing experiments run thus far the number of compute nodes in the cluster has been fixed. Thus we present an experiment where the number of compute nodes as varied and the effect on processing time measured. Logically, there should be no noticeable difference between the configurations initially, but there should be a appreciable drop-off in performance in each once the number of worker threads exceed the optimum subscription point and the system becomes over-subscribed.

The experiment involved repeated imports of a one thousand image collection, using an intensive metadata extraction process, and exclusively utilising the Open MPI framework. This framework reads the compute nodes to include from a configuration file, and so makes it easy to alter the number of utilised compute nodes between test runs. The experiment was run on a cluster with a maximum 15 compute nodes.

The results for this experiment are shown in Figure 5.10. It is immediately

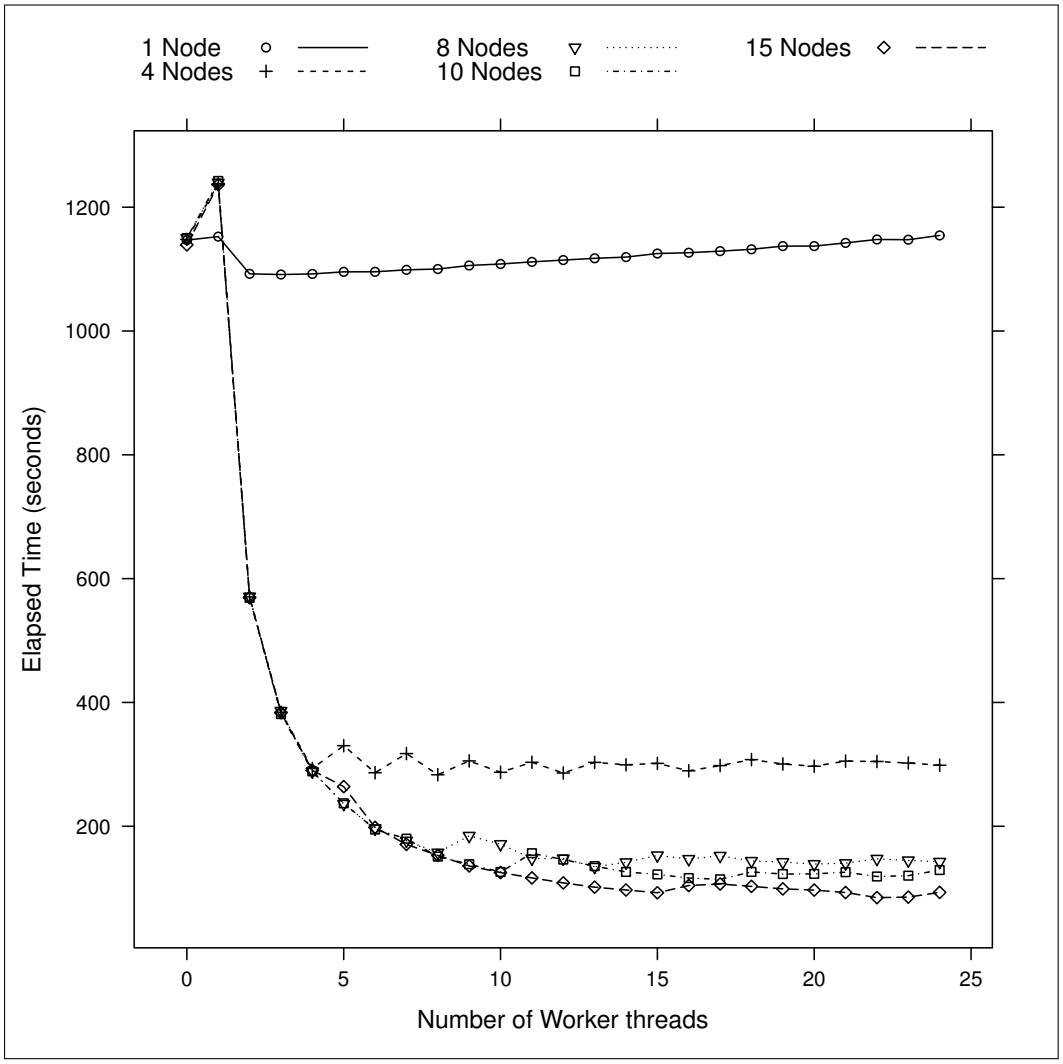


Figure 5.10: A comparison of the elapsed time of a one thousand image parallel import as number of worker threads and compute nodes vary

clear that all imports follow the same initial shape—after an initial increase due to the overhead of parallel processing, an increasing number of worker threads leads to a decrease in elapsed time. This trend continues until the number of worker threads exceeds the number of available compute nodes, at which point the elapsed time readings level off and may even increase slightly. The worst performance appears to occur when worker threads is equal to number of compute nodes plus one, with a definite spike deviating away from the curve. After this there is some evidence of a cyclic pattern, most likely related to numbers of worker threads that are multiples or share factors with the number of compute nodes.

The trend line for a single compute node also reveals two interesting features. Firstly, there is far less of an initial spike due to the overhead of parallel processing. Since the spike is similar for all other numbers of compute nodes, this suggests Open MPI contains an optimisation for a single compute node,

rather than some sort of cost linear to the number of compute nodes. Secondly, this trend line provides the clearest evidence for the increasing overhead of running more worker threads. There is a single compute node, and the amount of work remains unchanged, so the increase in processing time is directly attributable to the increasing number of worker threads and the cost incurred by Open MPI in establishing, managing, and scheduling them.

In conclusion, the experiment validates the initial expectation—the performance of parallel-processing imports on varying number of compute nodes is similar until the system becomes over-subscribed. Once over-subscribed the performance trend eventually plateaus, with the first over-subscribed test run exhibiting a significant decrease in performance over the previous, optimally subscribed one.

5.2.8 Summary

The experiments carried out above not only demonstrate that the parallel processing implementations of the chosen digital library software work, but that parallel processing can prove beneficial across a range of collection imports. Performance between the three digital libraries is overall similar, with DSpace initially slower but approaching Greenstone for larger collections, and Terrier consistently faster and with generally better scaling performance. The experiments all indicate a decrease in elapsed time when importing collections containing several document types, although the actual performance differs greatly. Collections with low processing requirements, such as a basic text collection, exhibit less performance increase than those with significant processing cost, such as video, and in one case was actually inferior to a serial import. Medium and high processing load collections, perhaps unsurprisingly, show greater gains in performance when processed in parallel. Overall there is a common trend across the experiments, with an initial decrease in performance—due to the overhead of parallel processing with none of the benefit—followed by a rapid increase in performance up to a minima point. This optimal point generally occurs when the number of threads is approximately the number of available compute nodes plus one, although the exact number was found to vary depending on file type and process load. While there may be other minima points after this first one, there is also an argument for diminishing returns, made obvious in the case of high processor load tasks such as importing video.

The results also document a similar trend in performance between the Open

MPI and Hadoop frameworks. While Hadoop performance was poorer, this can be attributed to a failure to properly leverage the *Map-Reduce* model and the lack of a suitable—to this research's cluster hardware configuration—and stable interface between Hadoop and the underlying Hadoop distributed file system.

Of greater implication for the development of a predictive model, as sought by this thesis, is that the ratio between processor and file transfer was found to have the most substantial affect on the performance. Minimal processing load configurations result in performance values substantially different from the naive assumption where the elapsed processing time is the serial time divided by the number of worker threads. This observation prompted the experiments detailed in the next section.

5.3 Processor versus File I/O Costs

During the course of this research it became apparent that the processing cost of importing a certain document type was likely to be the most important factor in whether parallel processing would provide measurable benefit. High processing cost activity, such as video processing, would obviously benefit from more processors being allocated to share the work. However, there is still some break-even point, as parallel processing incurs an overhead cost of managing a parallel execution environment as well as higher file transfer costs. These file transfer costs include:

- File locking and contention for multiple file readers and writers,
- Specific to cluster configurations, moving files between compute nodes, and
- Added inter-thread and inter-machine communication costs of the parallel framework.

The ratio of processing cost to file transfer cost when importing in digital libraries—or indeed in computer processing in general—is difficult to predict. It depends on many factors such as the type of document, the configured processes to apply to the document, and the file input/output involved in handling the document. In practice there are only two ways to predict this for a collection of documents in advance: measure a representative sample collection, or choose the most applicable value from among exemplar ratios.

Table 5.3: Statistics of elapsed time spent on file transfer for exemplar media types and configurations

Media type	Process load	Average	σ
Audio	Low	91.96%	1.81%
Audio	Medium	6.58%	0.04%
Audio	High	1.36%	0.02%
Image	Low	56.51%	1.17%
Image	Medium	16.72%	0.52%
Image	High	11.45%	0.11%
Text	Low	48.61%	0.19%
Text	Medium	9.88%	0.09%
Text	High	9.02%	0.02%
Video	Low	77.03%	0.85%
Video	Medium	0.83%	0.01%
Video	High	0.62%	0.01%

For the former, Section 3.4.1 described how measurement using *strace* was implemented in Greenstone; this could be repeated for other digital libraries. For the latter, a number of example collections—based on common use cases—were created for the research presented in this thesis, and their ratios measured as detailed below. A complete listing of these measurements is included in Appendix A.

Thus, a battery of experiments were designed to quantify processing costs of typical document collections and provide a breakdown between the processing and file transfer costs, the goal being to create a list of exemplar values that can be used as input into a predictive model. We utilised the same four collections shown in the experiments above—text, images, audio, and video—and once again applied configurations that required three distinct processing loads. Imports are run one hundred times for each combination, with the *strace* tool used to capture system level calls to file input/output functions. This capture provided not only a total elapsed time but also the percentage of that time spent in file transfer related activity. However, it should be noted that the introduction of system level tracing also has an effect on the elapsed time. We proceed under that assumption that this cost should be fairly evenly spread between file and non-file system level calls and thus the overall breakdown should still be accurate. Every attempt was made to reset the state of the environment, and in particular any memory-based file caching, between each run.

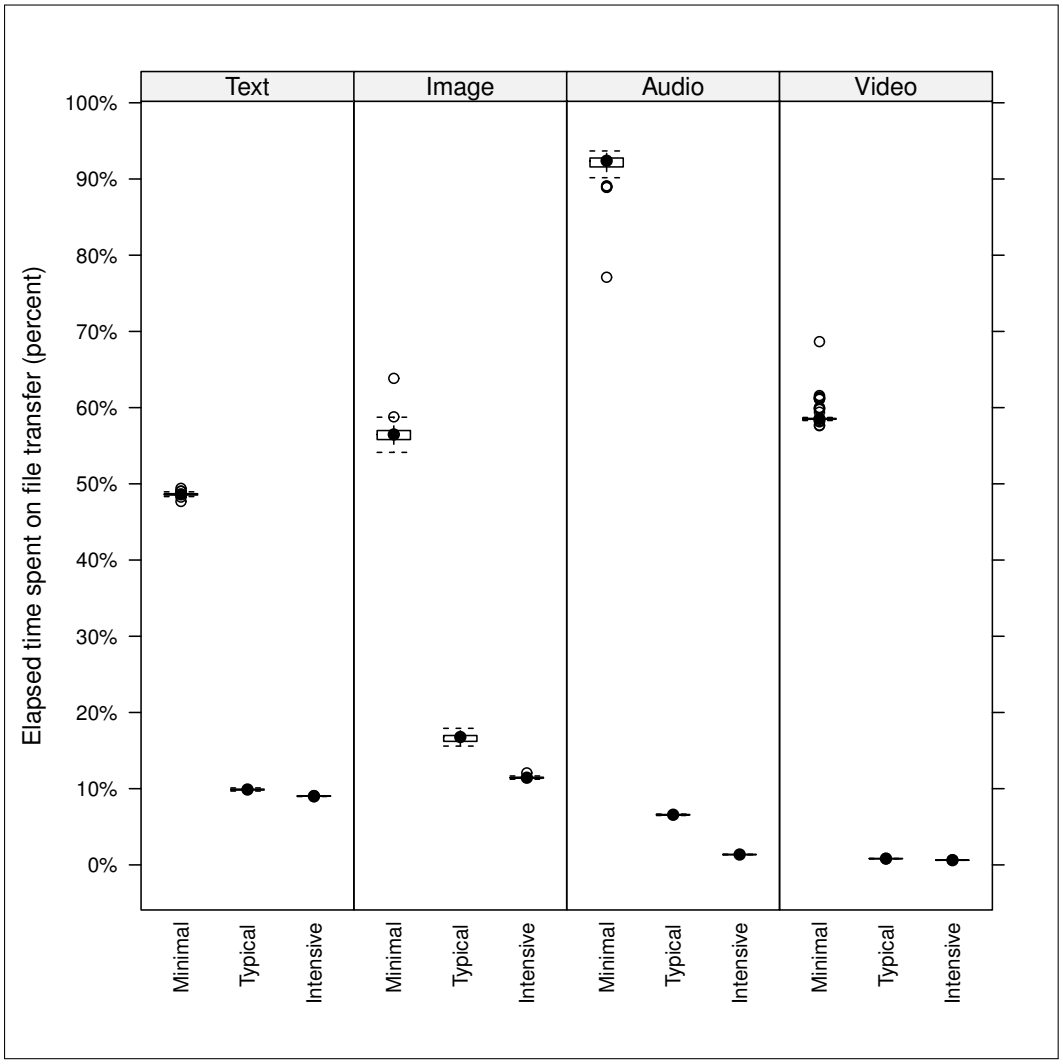


Figure 5.11: Percentage of elapsed time spent on file transfer for exemplar media types and configurations

From Figure 5.11 it is clear there is a significant difference between low process load imports and high process load imports. For example, in the case of the audio collection this difference is almost two magnitudes. There is also clear difference between the file transfer costs of the media types, most likely due to the relative ratio of file size and amount of metadata to be extracted (and hence cost of processing the media). Finally, the experiments showed more variation for the low process load imports as well, reflecting the influence of underlying file transfer factors. Medium and high processing loads are much closer together in terms of file transfer cost, from which can be drawn the conclusion that even small processing loads quickly surpass the file transfer costs.

It is important to note that these results represent typical but otherwise arbitrary configurations of Greenstone import. Low processing configurations do little more than copy the media and annotate with easily found metadata.

Medium processing configurations typically involve transforming the media into a more suitable format for web-based delivery. Finally, high processing configuration utilise data mining or other artificial intelligence techniques to derive more complex metadata. As mentioned these configurations were inspired by the case studies, and represent the typical settings a digital librarian might apply given the document type. However, the best way to determine the actual processing load versus file transfer cost ratio for a particular collection is to import a representative sample collection while measuring statistics using system level tools.

5.4 Miscellaneous Experiments

During the course of this research several other experiments were carried out, somewhat tangentially from the main thrust of research, but often with interesting results. This section explains the experiments found to be pertinent, along with the findings from the experiment and their impact on the research presented here or in parallel processing in general.

5.4.1 Database performance

The first experimental result in this section compares the timing performance of seven different databases configurable as the underlying data store in Greenstone. A full description of these databases can be found in Section 3.4.2, but for reference the databases are: GDBM, SQLite, GDBM using file locking (GDBML), TrivalDB (TDB), GDBM supporting multiple writers by utilising a client/server configuration (GDBMS), TrivialDB in a client/server configuration (TDBS), and TrivialDB with multiple files and a final merge suitable for running on clusters (TDBC). The first two database implementations only supported serial processing, the second two supported parallel processing on a single, multi-core computer, and the remaining three supported parallel processing in a cluster.

The experiment involves repeated imports of a subset of the Lorem Ipsum collection. In this case the subset consisted of one thousand OCR plain-text files only. For each run the environment—including any previously imported collection and disk cache—is reset, one of the aforementioned database types is selected at random, and the import run and timed.

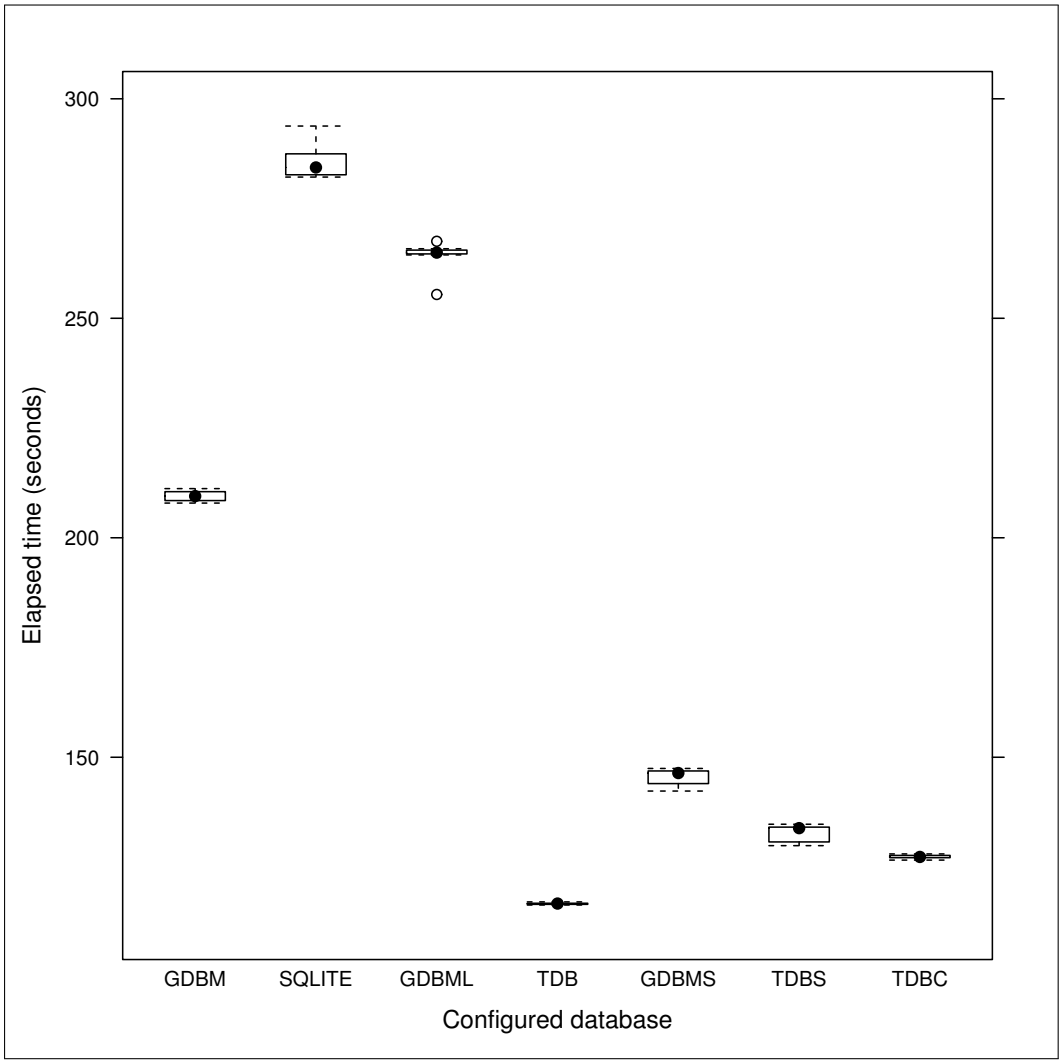


Figure 5.12: Comparison of elapsed time summaries for various database implementations

A forerunner for this experiment was carried out early in the implementation of parallel-enabled Greenstone, comparing the first four databases listed, with the results presented at the 2011 VLDL workshop [174]. With the addition of several new database implementations, as prompted by the need to support cluster computing, the experiment was rerun with the results presented in Figure 5.12.

We now compare the performance of the various database implementations relative to the performance of the default GDBM database. Proceeding left to right, the SQLite implementation took the longest, and exhibited the most variance, as expected given the overhead of the more powerful but expensive SQL interaction and implicit indexing. GDBM with file locking was a third slower than GDBM and recorded several outliers, most likely influenced by the underlying file system on which the locking depends. TrivalDB was the fastest database, mostly due to reuse of a single, persistent, database connection.

The first database utilising a client/server model, GDBMS, is significantly faster than vanilla GDBM despite returning to a bottleneck situation of a single writer. This speed increase is due to the server being able to open and persist a single database writer, and so simultaneously provides compelling evidence that most of TrivalDB's apparent superiority is due to the single reusable database connection. TrivialDB as a client/server provides more evidence of this fact as it, once saddled with the overhead costs of client/server communication, is similar in speeds to the server version of GDB. Note that both server versions exhibit some variance, again most likely due to influences from the underlying network and file systems. The final database implementation, TrivialDB for clusters, provides almost as significant speed increase as the original TrivialDB and is faster than client/server implementations, even with the delay caused by a final merging phase that, unoptimised, can only occur as a serial process.

The results show that a TrivialDB database would be significantly faster for Greenstone imports run on a single machine (regardless of number of cores). As mentioned, this is certainly caused by the way Greenstone necessarily interacts the GDBM database using multitudes of short-lived connections, and the fact the TrivialDB, with its multiple synchronous writers, can avoid this without major code refactoring. Unfortunately TrivalDB is not readily transferable to a distributed environment due to the file locking implementation (as explained in Section 3.4.2). The second fastest database implementation, TrivialDB designed for clusters, exhibits many of the same benefits of TrivialDB and provides a suitable and stable database for testing parallel processing on both multi-core computers and in a cluster.

5.4.2 Batch Size

Common to all parallel processing digital libraries implemented is the concept of import batches. Each processing thread is allocated a file listing to process, the batch, with each list specifying a number of full file paths to be imported. Early in the research, experiments were carried out to determine if there was an optimal number of files, or batch size, to include in each batch. For this exercise Greenstone was used to import the Lorem Ipsum collection subset, containing one thousand text documents, over a range of batch sizes and number of parallel worker threads. As well as recording the total elapsed time to import the collection, we also took note of the worker threads that actively performed processing as it was obvious that some batch sizes would prematurely starve worker threads.

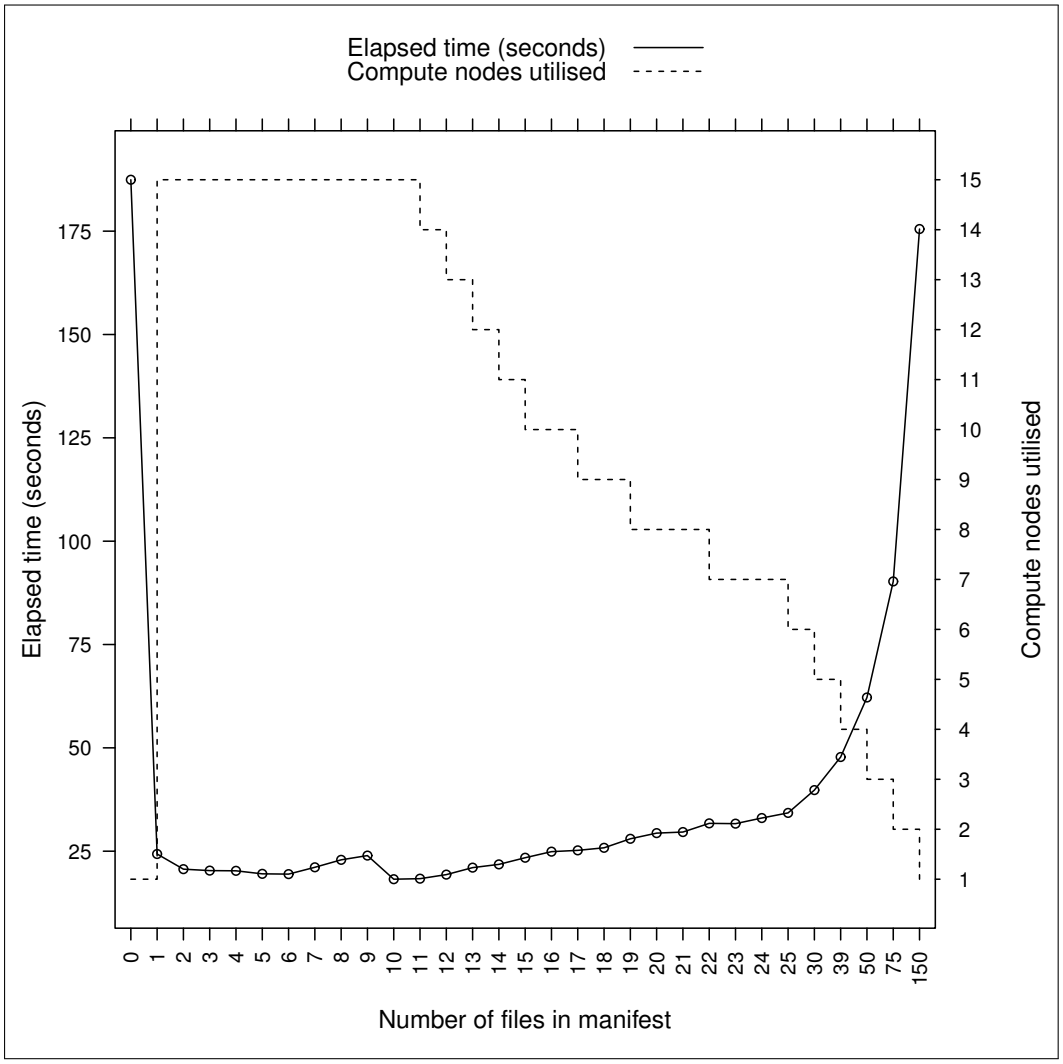


Figure 5.13: The effect of manifest file batch size on elapsed processing time and processor utilisation

Figure 5.13 shows the result of increasing batch size when parallel processed using 14 worker threads. The chart approximates a quadratic curve exhibiting a global minima point. This point occurs where file listing size exceeds the ratio calculated by the total number of documents divided by the number of workers. In this figure the batch size of 10 is shown to be optimal. At this point each worker thread has a balanced share of the work to perform, whereas larger sizes beyond this point mean some worker threads are under-utilised. Interestingly, smaller batch sizes—even as small as a single file per worker thread—still exhibit good performance. While there is an extra file transfer cost associated with processing more file listings, there is also the advantage that smaller manifest files help "fill in the gaps" as worker threads finish earlier due to variations in file sizes and processing times.

While the above finding had an impact of the Open MPI implementations, Hadoop implementations of parallel processing did not require manifests and

thus were not affected. The Map/Reduce algorithm includes built-in load balancing to avoid starving worker threads.

5.4.3 HDFS Replication Factor

The final reported experiment explores the Hadoop Distributed File System by measuring the effect of replication on data locality and the potential gains from achieving data locality. One of the key reasons for implementing parallel processing in the Hadoop framework was to leverage the property of data locality. Recall that this is the act of running a processing task on the same compute node that contains the data for the task. Data locality is directly related to replication factor: the number of compute nodes that hold a copy of a particular piece of data.

The experiment involved multiple imports of a collection of 54 ReplayMe! video files using parallel processing Greenstone and utilising the Hadoop framework. The experiments were performed on a cluster with 14 compute nodes. Before each test run, the distributed file system was reconfigured with a different replication factor, rebuilt, and restarted so as to minimize the effects of any file caching. While the default replication factor for Hadoop is 3, the experiment used replication factors ranging from 0 through to 14. A replication factor of 0 is a special case achieved through customisation of the scheduler and indicates that a task should never be run on a node that contains the data to be processed—in other words 0% data locality. Meanwhile a replication factor of 14 will ensure the distributed file system allocates each compute node a complete local copy of the data, resulting in 100% data locality.

The results shown in Figure 5.14 represent the averaged results of nine test runs at each replication factor, plotting both the percentage of processing tasks that encountered data locality and the elapsed import time as the replication factor increased. There are several findings to note:

- Increasing the replication factor resulted in a proportional increase in data locality and thus a decrease in elapsed time,
- Even at a replication factor of one, data locality occurred roughly a quarter of the time, indicating Hadoop's scheduler is fairly good at matching task and data,
- The greatest increase in data locality occurred at replication factor three, at which point data locality occurred two out of three times, and

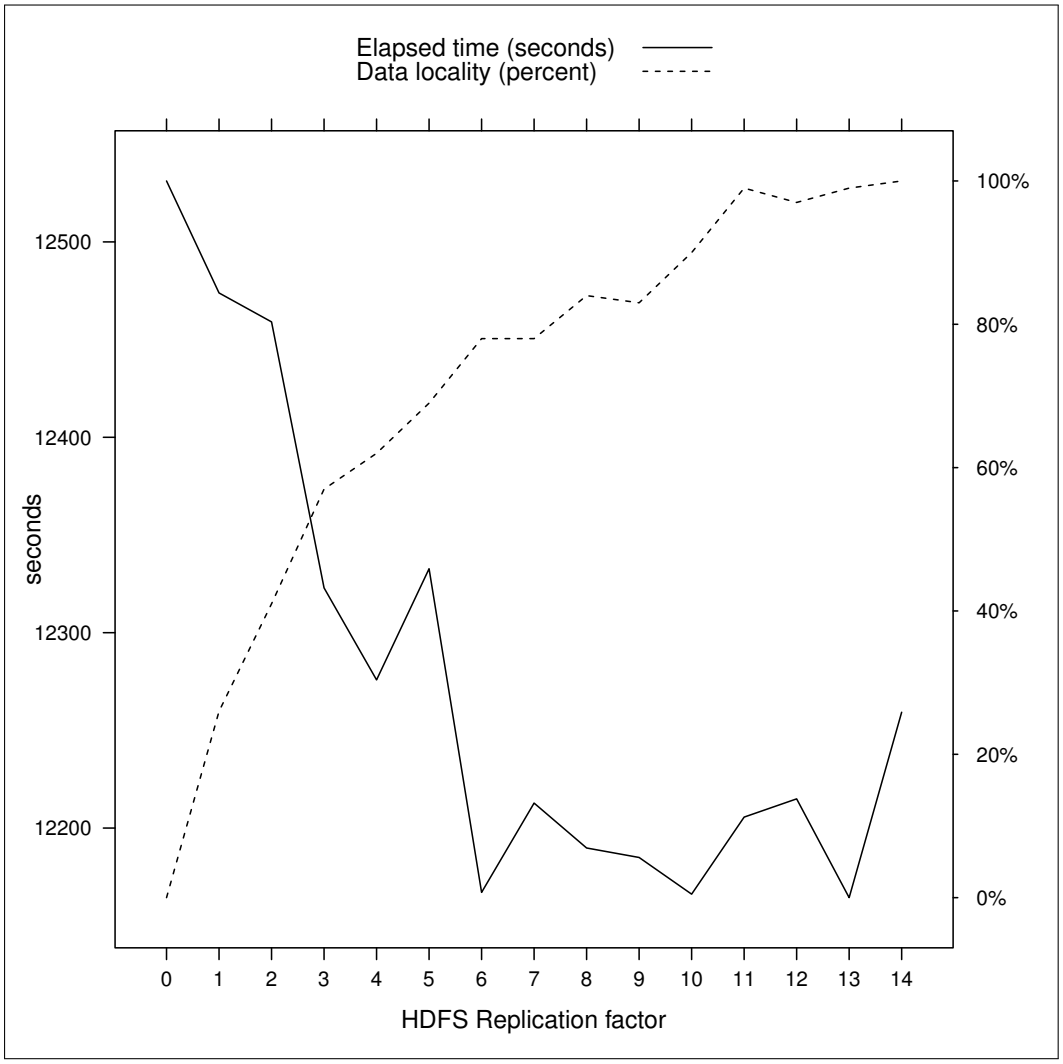


Figure 5.14: Effect of HDFS replication factor on data locality and processing time

- Despite increasing data locality, note the scale on the processing time axis. While processing time does decrease, the overall difference is very small compared with the total processing time——this is mostly likely due to the nature of the files processed, being low file transfer, high processing load, video segment files. Overall there was less than six minutes difference between the fastest and slowest elapsed processing time on an import that took over three hours.

The above implies that replication factor is not a significant feature given the research's hardware configuration and the nature of processing videos files. For the former, a fast switch with little contention and older, slower, local hard drives, meant that the file transfer speeds are still comparable between local and network locations. While for the latter, the file transfer accounts for less than 2% of the time spent in the configured video processing. This is evidenced

in earlier experiments showing processor versus file transfer ratios, as detailed in Section 5.3. Like many features in parallel processing, careful consideration and matching of task, hardware, and configuration may yield more favourable results when applying replication.

5.5 Discussion

This chapter presented details and results for a number of experiments designed to gather information about the performance of parallel processing. The experiments were grouped into three main sections.

The first section of experiments explored the effect of an increasing number of documents on serial importing times. Two media types were considered—the OCR-like plain text of the Lorem Ipsum corpus and the video files of the ReplayMe! corpus—as imported by the three general purpose digital libraries used in this research. The experiments showed a linear relationship between number of documents and processing time. The serial performance of Greenstone and DSpace were comparable, while Terrier had slightly better performance.

The second section began with an experiment that provided compelling evidence of the potential of parallel processing to better utilise processing capability on multi-core or cluster hardware. Immediately following this were a battery of experiments investigating the performance of the parallel processing implementations of the digital library systems over four differing collections of media. For each combination, three different intensities of metadata extraction (and hence processing load) were explored. While parallel processing generally provided some improvement in each experiment, the most significant gains were exhibited by those media that had low I/O and high processing costs (comparatively) where performance approached the theoretical maximum for the hardware configuration (this begin 7% of elapsed serial processing time). One configuration—that of applying a minimal metadata extraction process to audio media—stood out as an example of a collection where the parallel processing performance was worse than the serial performance.

The section continued with three further important experiments. The first compared the performance of the Open MPI framework to that of several variations of Hadoop/HDFS framework, and found that our implementations of the latter remained inferior despite several improvement attempts. The second experiment measured the effect of over-subscription of worker threads to com-

pute nodes, and confirmed the expectation that performance decreased past the optimal point suggested by existing literature [77]. The third experiment—the findings of which proved critical to the final predictive model—measured the ratio of time spent on processing versus I/O for the various combinations of media and metadata extraction intensity.

The final section of experiments included several miscellaneous results of general interest. An experiment that measured the performance of the various databases used in this research—as introduced in Section 3.4.2—was presented, with TrivialDB for Clusters being the best all-round result (and without the hardware limits of TrivialDB). Also presented was an investigation on the effect of varying manifest file sizes, with the key finding being that smaller manifests provided better utilisation of compute nodes while larger ones lead to more thread starvation. The experiments concluded with an experiment based on the HDFS environment and testing the effect of replication factor on data locality. The findings showed that the default replication, 3, provided a good trade-off between data locality and disk cost, but that the overall performance improvement in this research's implementation of HDFS was minor.

This chapter provides evidence that parallel processing can lead to performance improvements for certain media types and metadata extraction configurations, but such improvements are not universal. This further justifies the need for a model to predict the potential performance benefit/deficit. Consider the import of the ReplayMe! collection: a model could, at least, predict whether parallel processing might improve performance. Moreover, the existence of a predictive model—if it is indeed viable—will further be able to refine what is required in such a video processing digital library in terms of optimal compute nodes or, alternatively, how many television channels can be handled given a cluster of fixed size. The following chapter will make use of data gathered during the experiments above to develop the predictive model that is the primary contribution of this research.

Chapter 6

The Predictive Model

> Prediction is very difficult, especially about the future.
>
> — *Niels Bohr*

This chapter explores the creation of a mathematical model that predicts several key features of parallel processing in digital libraries, including total elapsed time, optimum number of computer nodes, and maximum processing complexity within a set time and hardware configuration. The chapter begins with a discussion of mathematical models and introduces important and contemporary related research. This is followed by a discussion on attribute selection and derivation, and concludes with an analysis of the models considered during this research, namely: the *Simple Division* model against which performance is benchmarked, and the *Parallel Processing of Digital Libraries* (PPoDL) model presented as the finding of this research.

6.1 Mathematical Modelling

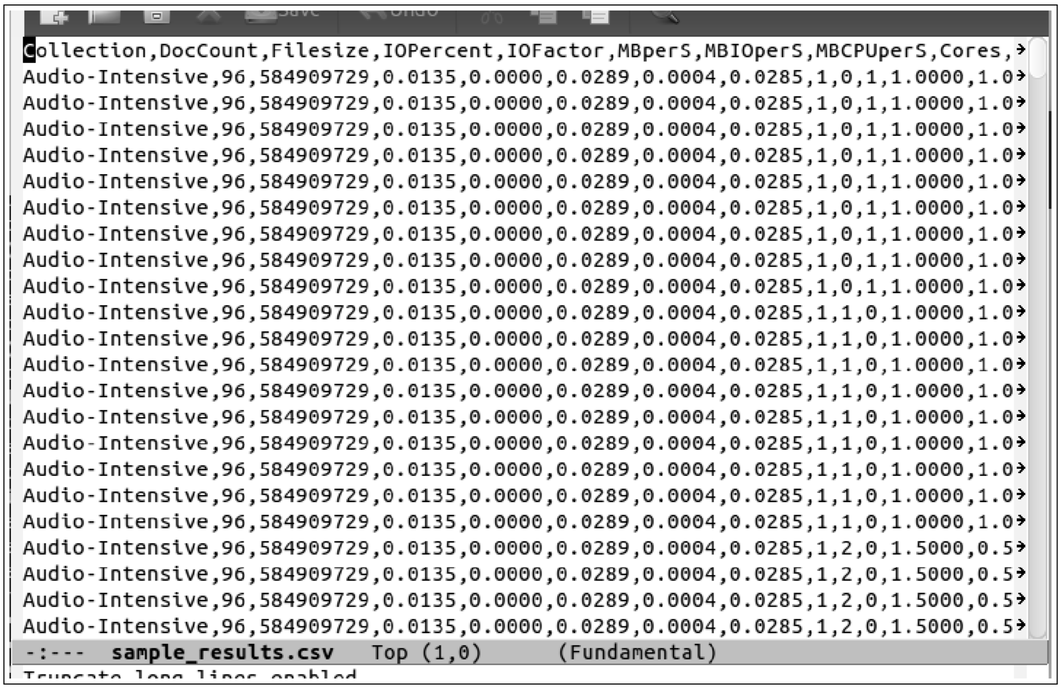
The goal of this research is to develop a mathematical model [132] capable of predicting key features of the parallel import stage of building digital libraries. This research focuses on automated approaches [204] for modelling when applied to large-scale data sets. As argued in Section 2.1, these techniques are necessary as the amounts of raw data being generated increases and our ability to understand (as things currently stand) the valuable information therein decreases.

Several tools were used during this research to automate the process of calculating mathematical models, foremost being *Weka* and *R*. Weka [91] is

a data mining workbench allowing a user to interactively select, review, and refine models for both prediction and classifying. The "Explorer" interface offers a wide selection of predefined models, can be configured for several validation techniques including cross-fold validation, and provides summary graphical representations for important features such as classifier errors. Similarly, R [149] is a powerful statistical and analysis programming language (and accompanying tool) that offers built-in predictive models and—with the inclusion of appropriate libraries such as *hydroGOF* [210] and *R/miner* [49]—provides goodness-of-fit measures and visualisations for model accuracy.

When creating mathematical models one important consideration is model selection. This task needs to balance the demand for accuracy against the model being suitable for the underlying data [150]. While the eventual use of linear regression to create the model will be justified in Section 6.3, earlier experimentation suggested that *IBk*—an implementation of the k-nearest-neighbour algorithm using instance based machine learning [2]—had comparatively superior accuracy. This model was eventually discovered to be suffering from a data sparseness problem. This issue is commonly seen in models with a higher number of attributes being modelled—the Curse of Dimensionality [22]. In essence, the resulting durations from two digital collections of different media, for example video and text, were spaced so far apart that it became trivial for the model to cluster them, resulting in very small relative errors in predictions. Unfortunately this means the model was effectively over-fitting and any future predictions—especially regarding documents or import processing with unseen properties—would be inaccurate.

A second important consideration is that of attribute selection and expression. The model was developed using the attributes, configuration, and results recorded during the parallel importing experiments over four media types (text, images, audio, and video) combined with different levels of processing load (minimal, typical, and intensive). Accounting for multiple test runs for each combination, there were $3,132$ records in the data used to train the model. Each record had 18 attributes; most were numeric but there were several textual (nominal) attributes such as collection name and media type. A sample of this data, in the form of a comma-separated file, is shown in Figure 6.1. From all the captured configuration and experimental values, we must select those that have the most significant effect on the accuracy of the model. Both Weka and R provide measures that guide this selection, for instance the significance measure provided by p-value [63]. Regardless, manual intervention is generally required to prepare the data and select amongst the attributes.



```
Collection,DocCount,Filesize,IOPercent,IOFactor,MBperS,MBIOperS,MBCPUperS,Cores,
Audio-Intensive,96,584909729,0.0135,0.0000,0.0289,0.0004,0.0285,1,0,1,1.0000,1.0
Audio-Intensive,96,584909729,0.0135,0.0000,0.0289,0.0004,0.0285,1,0,1,1.0000,1.0
Audio-Intensive,96,584909729,0.0135,0.0000,0.0289,0.0004,0.0285,1,0,1,1.0000,1.0
Audio-Intensive,96,584909729,0.0135,0.0000,0.0289,0.0004,0.0285,1,0,1,1.0000,1.0
Audio-Intensive,96,584909729,0.0135,0.0000,0.0289,0.0004,0.0285,1,0,1,1.0000,1.0
Audio-Intensive,96,584909729,0.0135,0.0000,0.0289,0.0004,0.0285,1,0,1,1.0000,1.0
Audio-Intensive,96,584909729,0.0135,0.0000,0.0289,0.0004,0.0285,1,0,1,1.0000,1.0
Audio-Intensive,96,584909729,0.0135,0.0000,0.0289,0.0004,0.0285,1,0,1,1.0000,1.0
Audio-Intensive,96,584909729,0.0135,0.0000,0.0289,0.0004,0.0285,1,0,1,1.0000,1.0
Audio-Intensive,96,584909729,0.0135,0.0000,0.0289,0.0004,0.0285,1,1,0,1.0000,1.0
Audio-Intensive,96,584909729,0.0135,0.0000,0.0289,0.0004,0.0285,1,1,0,1.0000,1.0
Audio-Intensive,96,584909729,0.0135,0.0000,0.0289,0.0004,0.0285,1,1,0,1.0000,1.0
Audio-Intensive,96,584909729,0.0135,0.0000,0.0289,0.0004,0.0285,1,1,0,1.0000,1.0
Audio-Intensive,96,584909729,0.0135,0.0000,0.0289,0.0004,0.0285,1,1,0,1.0000,1.0
Audio-Intensive,96,584909729,0.0135,0.0000,0.0289,0.0004,0.0285,1,1,0,1.0000,1.0
Audio-Intensive,96,584909729,0.0135,0.0000,0.0289,0.0004,0.0285,1,1,0,1.0000,1.0
Audio-Intensive,96,584909729,0.0135,0.0000,0.0289,0.0004,0.0285,1,1,0,1.0000,1.0
Audio-Intensive,96,584909729,0.0135,0.0000,0.0289,0.0004,0.0285,1,1,0,1.0000,1.0
Audio-Intensive,96,584909729,0.0135,0.0000,0.0289,0.0004,0.0285,1,2,0,1.5000,0.5
Audio-Intensive,96,584909729,0.0135,0.0000,0.0289,0.0004,0.0285,1,2,0,1.5000,0.5
Audio-Intensive,96,584909729,0.0135,0.0000,0.0289,0.0004,0.0285,1,2,0,1.5000,0.5
Audio-Intensive,96,584909729,0.0135,0.0000,0.0289,0.0004,0.0285,1,2,0,1.5000,0.5
-:---  sample_results.csv   Top (1,0)     (Fundamental)
Truncate long lines enabled
```

Figure 6.1: A sample of the data captured during parallel processing experiments

Consider the summary of the corpus and configuration attributes presented in Table 6.1. There is a magnitude of difference between the file size of a collection of text and one of video, and the elapsed processing time is equally disparate between minimal, typical, and intensive processing configurations. To address this, normalisation is applied to the data resulting in several new derived attributes. For example, the model predicts the parallel processing import performance as a percentage of the serial import elapsed time rather than trying to predict the elapsed time in seconds. Specific attribute normalisations applied to the model's data will be discussed in Section 6.3.

Another consideration, foreshadowed above, is that of overfitting. This means the model is accurate at predicting values similar to what it has already seen, by way of training data, but exhibits poor accuracy when predicting new and unseen values; essentially the model memorises the data, rather than 'learning' anything about the trends present in the data. While an overfit model is easy to accidentally create, either by having too many attributes or only testing using seen values, there are also techniques to reduce the risk of overfitting. One common technique for reducing the risk of overfitting is cross-validation, where some part of the observed data is set aside specifically for testing and is not used in training. This test data is then used to determine whether the model is be able to predict values it has not seen before. The models created in this section were tested for overfitting by the use of cross-

Table 6.1: Summary of test collection information

Collection	Media	Average doc size (bytes)	Metadata processing load	I/O%	Import time per file (seconds)
Lorem	Text	3,405	Intensive	11.75%	1.48
			Typical	9.89%	0.51
			Minimal	48.74%	0.07
Wang	Image	29,846	Intensive	11.45%	1.15
			Typical	16.78%	0.36
			Minimal	56.48%	0.03
Jamendo	Music	6,092,810	Intensive	1.35%	200.91
			Typical	6.57%	11.27
			Minimal	91.97%	0.06
Replayme!	Video	360,259,772	Intensive	0.62%	2,129.04
			Typical	1.86%	1,299.93
			Minimal	58.82%	12.76

validation:

- In Weka, the default 10-fold cross-validation was always applied during model generation.
- In R, the model was likewise tested by applying 10-fold cross-validation available from the *DAAG* library [122].
- Moreover, this research also estimated the effect of leave-one-out validation by applying the CV function in the *forecast* library [96]. A low CV value indicates the model is less-likely to be overfit.

Moreover, the models tested were created using a simplicity-first approach: choosing models with fewer attributes over those with more. This is recommended as a dataset may exhibit information that is not immediately obvious and there are cases where a simple model may perform as well, or even better, than far more complex models [92]. While there are several statistics available for measuring the accuracy of any given model, such as the correlation coefficient, it is often useful to include a comparison against simpler benchmark models in order to determine relative performance. Many of the typical benchmark models, for instance *ZeroR*, *OneR*, and *Naïve Bayes* [124], are not particularly useful in this case as they either cannot be used to predict a numeric class or are too simple to reflect the wide range of possible values (ZeroR,

for example, generates a single average threshold value). Thus we introduce the Simple Division model as a benchmark, in order to allow better judgement of the relative performance of the model developed during this research.

6.2 Simple Division Model

The Simple Division approach is the intuitive way of predicting parallel processing performance by dividing the total expected serial performance by the number of worker threads, as formalised in Equation 6.1. The Simple Division value forms one of the initial attributes recorded in the training data, as it relies only on other known attributes and so can be calculated prior to experimentation. The results shown in Chapter 5 indicate that such a model exhibits some accuracy despite not taking into account any features of the documents being processed nor applying the overhead cost of parallel processing. From those same results it can be seen that the Simple Division model is less accurate when the collection import requires less processing and conversely has higher file transfer requirements. When predictions are compared to the observed elapsed time values, the Simple Division model has a correlation coefficient of 0.2356, a root mean squared error of 0.8599, and an overall mean absolute scaled error of 0.1305. Later in this chapter these measures will be explained and compared with results from the PPODL model. Computed coefficients are rounded to 4 decimal places (equivalent to a percentage expressed to 2 decimal places).

$$SimpleDivision = \begin{cases} \frac{1}{Workers} & \text{if } Workers > 0 \\ 1 & \text{if } Workers = 0 \end{cases} \tag{6.1}$$

The Simple Division model overlooks two important aspects of parallel processing that we argue should be captured. Firstly, a more sophisticated model would include measures that reflect the balance between number of worker threads and available compute nodes pivoting on the optimal load point. Secondly, the model should include a ratio of the processing costs of importing versus the file transfer costs. The less metadata processing applied, the lower the processing cost; the higher the percentage of time spent on file transfer, the less performance to be gained by parallel processing.

6.3 PPoDL Model

The main contribution for this research, the Parallel Processing of Digital Libraries (PPoDL) model, is generated using a *Linear Regression* technique. This technique involves calculating the formula that produces predictions with high correlation to the trend exhibited by the observed experiment results. The trend matched does not necessarily have to be a straight line and by applying exponents to the attributes that make up the model it is possible to correlate with complex curves. Although there may be other models with higher accuracy, a linear regression model was selected as it has several qualities desirable for this research that others do not, including:

- The model can be expressed as a formula, with the influence of each attribute quantified,
- As a formula, it is possible to rearrange the resulting model to predict alternate attributes, and
- It is straightforward (unlike some models) to visualise both the residuals and the predicted values themselves.

The initial step in creating a model is to extract the initial set of attributes from the experimental results and then extend this set with several new derived attributes to focus on specific trends shown in the actual measured parallel import performances. Initial attributes of importance, taken directly from experimental data, are:

- **CollectionName** - the only textual attribute, and is used solely as a label to group predictions when producing charts,
- **Cores** - the number of processing cores in either the single multi-core computer, or in total over a cluster of computers,
- **Workers** - the number of configured worker threads for the parallel import, where zero workers indicate a serial import and the total number of workers may exceed the total number of cores (over-subscription),
- **IOPercent** - the percentage of elapsed time expected to be spent in file transfer activity for this configuration measured either from a sample import or looked-up from exemplar values,
- **CollectionSize** - the total size, measured in megabytes, of the files processed during importing,

- **ElapsedSerial** - the time, measured in seconds, of importing the collection in serial, and
- **Observed** - the actual performance value, calculated as a percent of serial elapsed time, and is included in the data in order to allow cross-validation and calculation of model accuracy statistics.

While other primary attributes were trialled, these proved to less useful in creating the model as explained shortly. To this initial set of attributes were added several derived numeric attributes, informed by the results in Section 6.2, namely:

- **IOFactor** - derived from the *IOPercent* in such way as to make the effect of higher values more pronounced. It does so by using a cubic growth curve as calculated as per Equation 6.2. Note that, since *IOPercent* is presented as a decimal it is multiplied by 100 so that cubing increases the value, while the divisor is chosen solely to make the coefficients calculated by R of similar magnitude and thus more readable.
- **MBIOperS** - (short for megabytes of I/O per second) an indication of the speed at which file transfer occurs given the collection configuration. It was introduced so as to reflect the configurations where the file size and file transfer cost combine to adversely affect parallel processing, as in the case of the Audio-Minimal. It is calculated as shown in Equation 6.3.
- **SimpleDivision** - an attribute calculated by Equation 6.1. The Simple Division model contains the basic trend exhibited by actual performance values and thus is included in the model.
- **WeightedDivision** - similar to *SimpleDivision* but takes into account that only the processing cost is divided by increasing numbers of workers, while the file transfer cost is not. The values for this attribute are calculated using Equation 6.4.
- **Subscription** - an attribute that captures the ratio between *Workers* and *Cores* in order to reflect increasing benefit of adding more worker threads while also exhibiting a local optimal or break-even point beyond which more workers do not provide a benefit (and may actually incur a penalty). Equation 6.5 results in a curve that offers poorer returns after the optimal point is reached [77].
- **IsSerial** - essentially a flag that equals 1 when *Workers* equals 0, and 0 otherwise, as shown in Equation 6.6. This attribute is intended to capture the overhead cost exhibited when moving to parallel processing.

$$IOFactor = \frac{(IOPercent * 100)^3}{100,000} \tag{6.2}$$

$$MBIOperS = \frac{CollectionSize}{ElapsedSerial} \times IOPercent \tag{6.3}$$

$$WeightedDivision = \begin{cases} \frac{1-IOPercent}{Workers} + IOPercent & \text{if } Workers > 0 \\ 1 & \text{if } Workers = 0 \end{cases} \tag{6.4}$$

$$Subscription = \frac{1}{Cores} + \left(|Workers - Cores| \times \frac{1}{Cores + 1} \right) \tag{6.5}$$

$$IsSerial = \begin{cases} 0 & \text{if } Workers > 0 \\ 1 & \text{if } Workers = 0 \end{cases} \tag{6.6}$$

Figure 6.2 provides an indication how the general trend of each selected attribute might contribute to towards the observed values. This figure helps visualise the interaction of attributes, and was considered during manually attribute selection, but is illustrative only as no measure of scale is included and so trends can not be directly combined (or compared).

This data was loaded into the statistical program R and a linear regression model formed on all of the available attributes. To refine the model the summary and ANOVA analysis for this iteration of the model is reviewed, the attribute with the highest p-value removed (failing the significance test), and the model regenerated. This process was repeated until all attributes pass the significance test and removing any further attributes reduces correlation or increases error. In R, the *F-statistic* [64] can also be used as a guideline as to whether removing another attribute will improve the model.

A summary of the linear regression model generate in R is presented as Listing 6.1. The key information from this output is that the residuals appear to be normally distributed, that all remaining attributes exhibit high

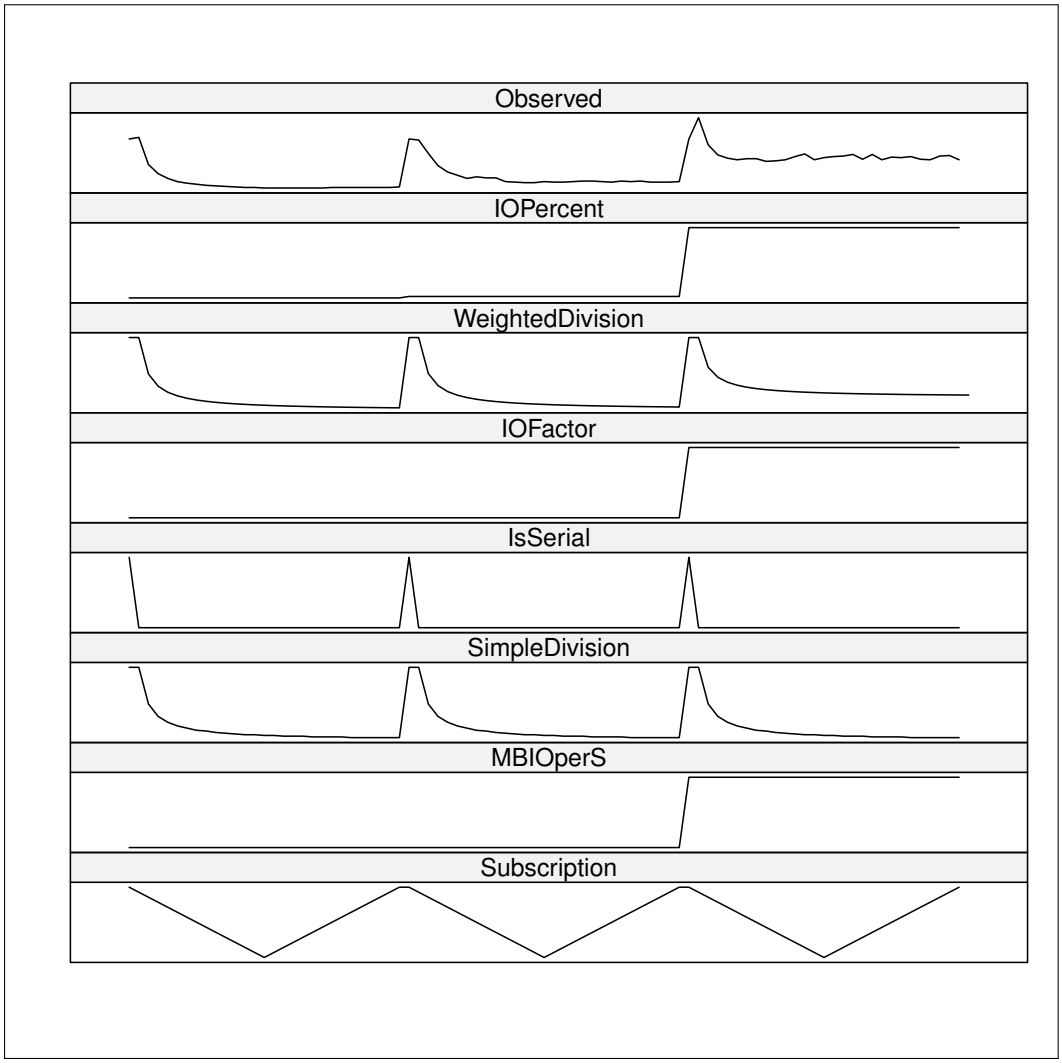


Figure 6.2: Illustrative chart showing how derived attributes combine to resemble the observed values

```
Call:
lm(formula = Actual ~ IOPercent + IOFactor +
MBIOperS + SimpleDivision + WeightedDivision +
Subscription + IsSerial, data = data)

Residuals:
Min      1Q  Median      3Q     Max
-2.25646 -0.04644 -0.01481 0.03337 1.87473

Coefficients:
Estimate Std. Error t value Pr(>|t|)
(Intercept)    0.0521227 0.001486 35.087 < 2e-16 ***
IOPercent     -1.3017862 0.013443 -96.839 < 2e-16 ***
IOFactor       0.2386331 0.003781 63.107 < 2e-16 ***
MBIOperS       0.0175935 0.000234 75.295 < 2e-16 ***
SimpleDivision 0.1920048 0.008480 22.642 < 2e-16 ***
WeightedDivision 0.9276481 0.009893 93.768 < 2e-16 ***
Subscription   0.0023164 0.000301 7.689 1.51e-14 ***
IsSerial      -0.2304582 0.004869 -47.332 < 2e-16 ***
---
Signif. codes: 0 ‘***’ 0.001 ‘**’ 0.01 ‘*’ 0.05 ‘.’ 0.1
‘ ’ 1

Residual standard error: 0.1546 on 48340 degrees of
freedom
Multiple R-squared: 0.9645, Adjusted R-squared: 0.9645
F-statistic: 1.878e+05 on 7 and 48340 DF,
p-value: < 2.2e-16
```

Listing 6.1: PPoDL model summary in R

significance (three stars), and the r^2 correlation is high at 96%. One attribute not directly used in the model is *Workers*; however recall that the number of worker threads is an important contributing factor to both the *WeightedDivision* and *Subscription* attributes. The magnitude of coefficient multipliers are slightly indicative of the weight a particular attribute has towards the actual trend; in this case *IOPercent* contributes the most, followed by *WeightedDivision* and then *SimpleDivision*. However, the actual influence also depends upon the magnitude of the attribute value. In a simple linear regression the sign of the coefficient is directly reflected in the prediction, yet on initial inspection, some of the coefficients in the PPoDL model might be surprising. In particular *IOPercent* which is negative where a positive coefficient might be expected. This is caused by multicolinearity—where primary attributes are included in several derived attributes, some of which exhibit a different rate of change—as discussed in Section 6.3.1.

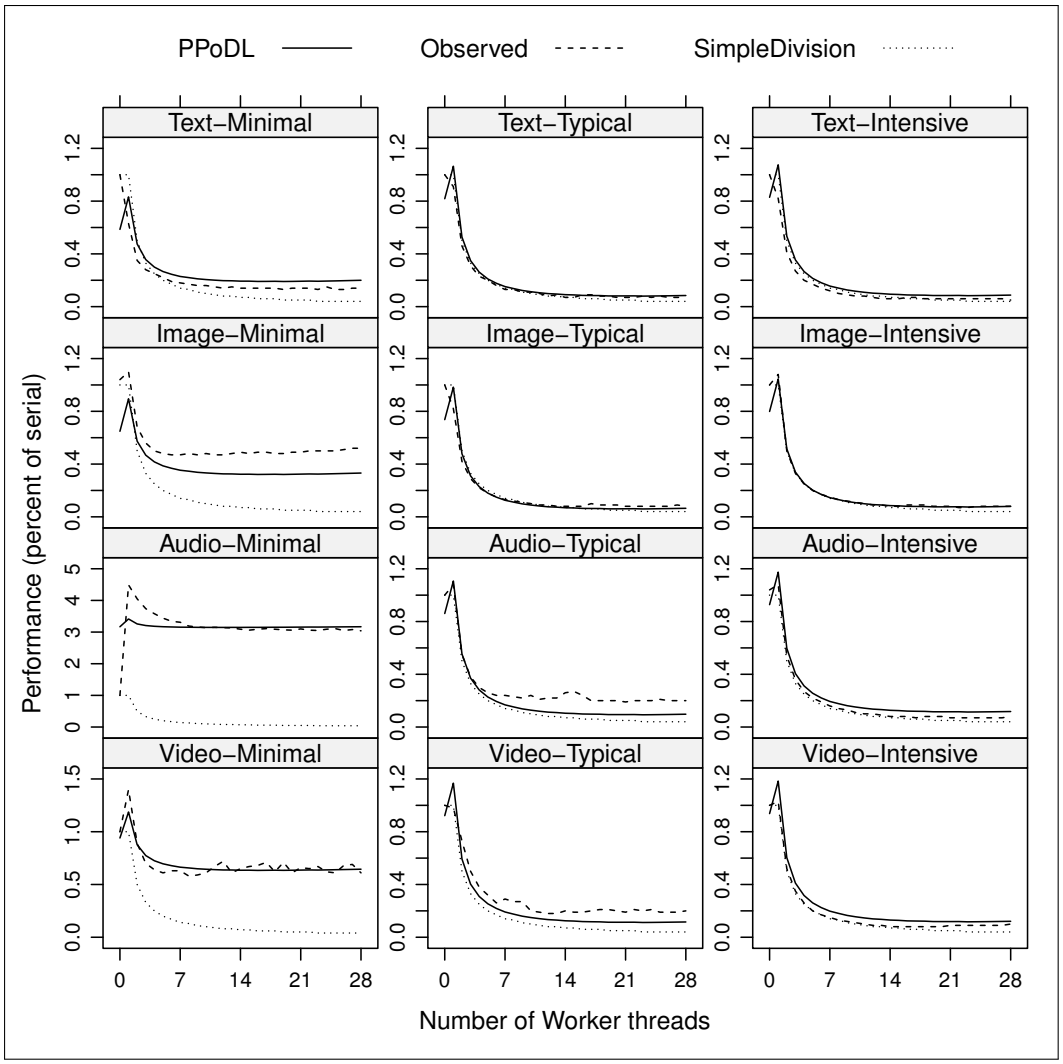


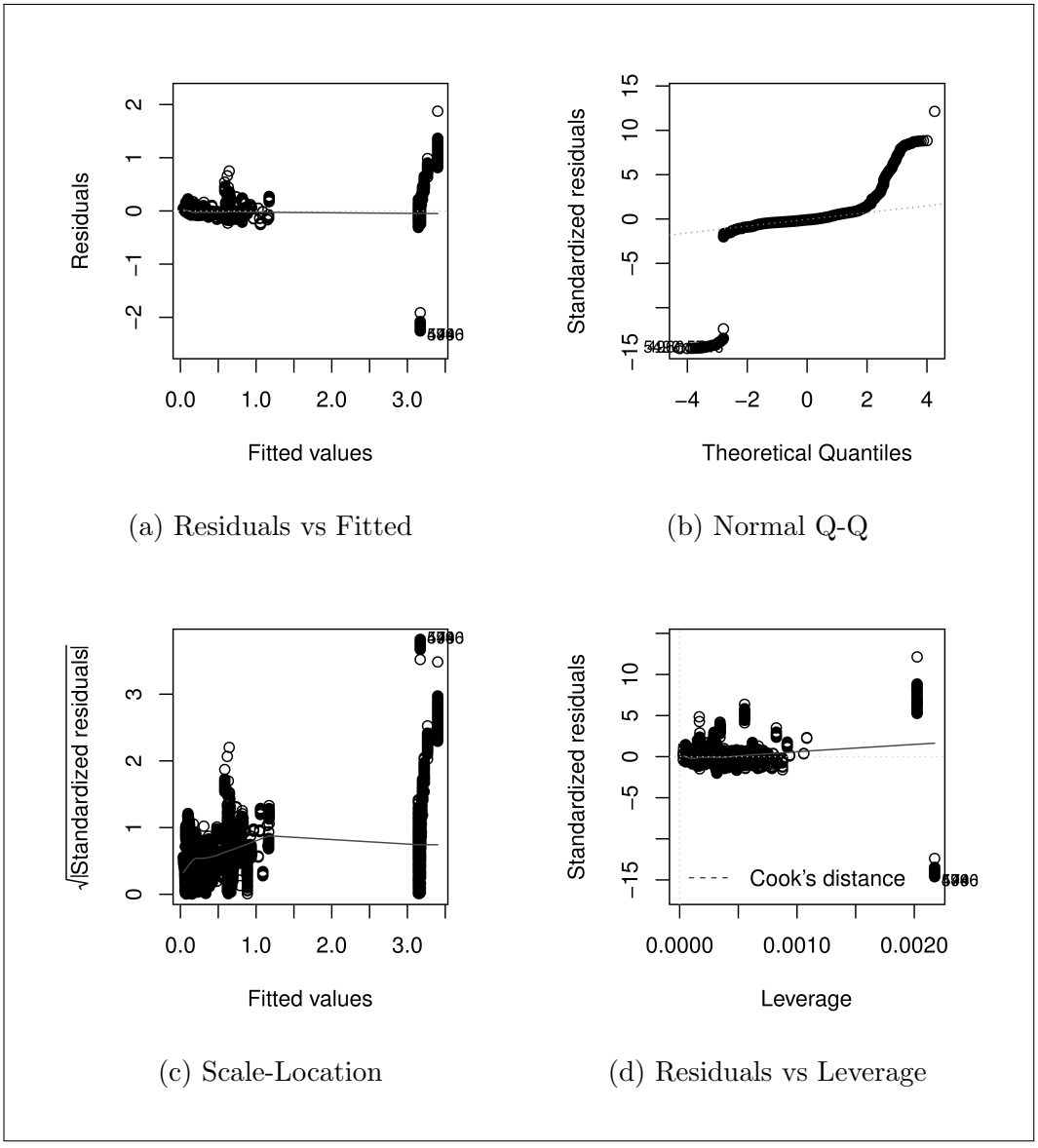
Figure 6.3: Illustrative chart of difference between observed performance and model predictions

A visualisation of the accuracy of the PPoDL model, when compared to the Simple Division model, is presented in Figure 6.3. The figure shows predictions of parallel performance, as a percentage of serial performance, over a growing number of worker threads and grouped by the various document types and configurations. Note that the chart representing minimal metadata processing over a collection of audio files is plotted to a different scale.

Considering the Standard Division model's predictions first, while they are somewhat similar to the observed performance, the prediction values are exactly the same for each experimental set-up. Consequently, high correlation is only exhibited during intensive metadata import configurations. The majority of the residuals are negative indicating the model typically overestimates performance. Furthermore, the charts exhibit a point after which the Simple Division model becomes gradually less accurate. These points are closely correlated with the optimal thread number, as explored in Section 5.2.

In comparison, the PPoDL model more accurately and consistently correlates with the observations. Note the initial decrease in processing performance due to: the overhead of parallel processing; the more accurate vertical alignment of the trend-line due to factoring in file transfer percentage; and the way the PPoDL model's predictions are slightly parabolic reflecting the cost of over-subscription. Furthermore, there appears to be a balance of over and under estimation, although the different scales involved make determining the actual effect of outliers difficult. The PPoDL model is poor at predicting performance when *Workers* is 0, in other words the serial import instances, when compared to the Simple Division. However, this is of minimal concern given the model targets parallel configurations and 0 workers results in a serial, not parallel, setting. Logically, performance should always be 100% for serial imports, so use of the model in this case is not expected nor is it required.

The construction of a linear regression model, and in particular the application of the Ordinary Least Squares algorithm [35], requires several assumptions about the underlying data. The main assumption is that the formula of the linear regression captures all of the information present in the data, and any difference between actual and predicted values is genuine random noise. One way to determine if these assumptions hold is to analyse the errors or residual values between the model's predictions and actual values. R provides a four-chart visualisation of this information, as shown in Figure 6.4. "Residuals vs Fitted" (Figure 6.4a) shows an even spread of residuals around zero, but exhibits two distinct groupings of points mostly likely caused by predictions produced for high file transfer imports having significantly larger errors than



(a) Residuals vs Fitted

(b) Normal Q-Q

(c) Scale-Location

(d) Residuals vs Leverage

Figure 6.4: Residual plots from the application of the PPoDL model

low file transfer ones. The "Normal Q-Q" (Figure 6.4b) exaggerates the tails of the residual distribution and shows evidence of heavy tails (leptokurtosis). This suggests there is more information available in the tails of the data than would be expected in a normal distribution and that the model fails to capture all of this information. Next is the "Scale-Location" (Figure 6.4c), in which the near-flat fitted *Lowess* line indicates the residuals exhibit constant variance. Finally, the "Residuals vs Leverage" (Figure 6.4d) is good in that the residuals are evenly spread, but does contain some evidence that outlier points may have an undue effect, once again mostly likely those high file transfer instances that exhibited a magnitude difference in performance to most instances. In summary, while the model exhibits high accuracy these plots indicate there may be more information hidden in the data than what is being currently leveraged.

The model building exercise was then repeated using the Weka application in order to validate the model. The data loaded into Weka consisted of nearly all of the attributes listed above excepting *Collection* which, being of data-type string, would otherwise preclude the use of regression functions. The implementation of a linear regression classifier in Weka includes two algorithms for automatic attribute selection: M5 [148] or Greedy (a backward elimination technique using Akaike Information Criterion score [3]). Table 6.2 includes results from both of these automatically built models alongside the PPoDL model calculated in R but reproduced in Weka.

Table 6.2: Comparing PPoDL attributes with those automatically selected by Weka's M5 and Greedy algorithms

Algorithm	Count	Attributes
PPoDL	7	IOPercent, IOFactor, MBIOperS, Subscription, IsSerial, SimpleDivision, WeightedDivision
M5	7	IOPercent, IOFactor, MBIOperS, Workers, IsSerial, SimpleDivision, WeightedDivision
Greedy	8	IOPercent, IOFactor, MBIOperS, MBCPUperS, Workers, IsSerial, SimpleDivision, WeightedDivision

From the work above, and Equations 6.1 through 6.5, the following linear regression model is presented as an answer to this research's hypothesis:

$$
\begin{aligned}
PredictedPerformance = {} & 0.0521 \\
& - 1.3018 \times IOPercent \\
& + 0.9276 \times WeightedDivision \\
& + 0.2386 \times IOFactor \\
& - 0.2305 \times IsSerial \\
& + 0.1920 \times SimpleDivision \\
& + 0.0176 \times MBIOperS \\
& + 0.0023 \times Subscription
\end{aligned}
\tag{6.7}
$$

Table 6.3: Summary statistics from linear regression models

Model	PPoDL	SimpleDivision	M5	Greedy
Attributes	7	1	7	8
Pearson's r	0.9821	0.2215	0.9821	0.9821
$\bar{r}^2$	0.9645	0.0488	0.9648	0.9645
RMSE	0.1547	0.8600	0.1547	0.1546
MASE	0.1348	0.6513	0.1348	0.1312

Table 6.3 compares the selected accuracy measures from the PPoDL model to that of the two automatically generated models, M5 and Greedy, and the PPoDL model. From the values for r^2 it can be observed that the PPoDL model performs substantially better than the Simple Division model and similar to those automatically built in Weka. The Greedy model has marginally better performance on most measures, while the M5 and PPoDL models have the same performance. Returning to the principal of simplicity-first, the PPoDL model is preferable to the more complex Greedy one, despite the concern of over-fit—given that the automatically generated model has a higher number of attributes—being mitigated by Weka's ten-fold cross-validation.

The accuracy measures used in the table above were selected from the many *goodness of fit* measures available in the R and Weka tools, so as to provide a balanced overview of performance. High correlation by itself does not necessarily indicate a good model—it must be viewed in context with several other metrics. The accuracy measures used in this research are:

- **Correlation coefficient**. Also knows as Pearson's r, this measure indicates the strength of the linear correlation between two trend lines and is expressed as a value from 1 (total positive correlation), through 0 (no

correlation), to -1 (total negative correlation). In general, higher values indicate a more accurate model. While this is not, strictly, a linear relationship—and Pearson's r can prove subjective in terms of how high a value to consider as accurate—it offers a good general measure of accuracy; in this case the Simple Division model is (literally) "along the same lines" 22% of the time as compared to the PPoDL's correlation of 98%. As mentioned, a high value of r does not necessarily indicate the model is a good fit for the data, just that it exhibits a similar overall trend.

- **Adjusted r^2**. The square of Pearson's r, but adjusted to account for the number of attributes used in the model, this measure is used to determine the ability of the model to explain the variation between the predictor and the response attributes. Again, higher values indicate more accurate models. Unadjusted r^2 tends to be optimistic, so the adjusted version provides a better measure of how strongly the model predicts the actual values. In this case the Simple Division model describes less than 5% of the variation in observations while the PPoDL model describes 96%.

- **Root Mean Squared Error (RMSE)**. A measure of the error between prediction and observation, with lower values of RMSE indicating a more accurate model. The errors are squared so that over and under estimating are equally bad. While mathematically easy to work with, RMSE can be more sensitive to outliers. In these experiments there are a small number of results that skew this measure; specifically the results from the parallel processing of image and audio files with minimal metadata extraction where the percentage of time spent on processing is low versus the time spent on file input/output. Still, the Simple Division model's RMSE of 0.86 indicates poorer accuracy than the 0.15 of the PPoDL model.

- **Mean Absolute Scaled Error (MASE)**. A modern, and generally applicable measure of model accuracy that avoids the issues seen in many other metrics. While initial developed for forecasting, where a time series allows historical observations to be used in future predictions, a version has been developed for cross-sectional experimental data as produced in the research [96]. Since MASE is based upon residual errors, lower values indicate more accurate models. The performance of the PPoDL model, at 0.13, is significantly better than that of the Simple Division model, at 0.65.

The high correlation coefficient and low relative errors produced by the

Table 6.4: Summary statistics of PPoDL model when applied to differing levels of metadata processing

Metadata Load	Overall	Minimal	Typical	Intensive
IOPercent	0.2613	0.6400	0.0877	0.0561
$\bar{a}$	0.5247	1.1534	0.2369	0.1840
Pearson's r	**0.9821**	0.9771	0.9572	0.9799
$\bar{r}^2$	**0.9645**	0.9519	0.8849	0.9470
RMSE	0.1547	0.2629	0.0790	**0.0552**
MASE	0.1348	**0.1060**	0.4239	0.2685

PPoDL model, indicating approximately 96% correlation, may be optimistic and is certainly being skewed by testing instances with high file transfer percentages. The performance for these instances are, in some cases, magnitudes larger than the other two categories of processing and even small residuals for these instances have a disproportionate effect on overall averages. Table 6.4 shows the same selection of goodness of fit measures when applying the model to each of the three levels of metadata processing. Included in this table is ā (the average observed value for performance) and the *IOPercent* to aid with comparisons. The main table features are the decrease in correlation for the lower file transfer percentage (higher metadata processing) categories, and the substantial jump in the two relative error measures. A 0.06 error on a performance value that averages 0.18 (intensive) should be more significant than a 0.26 error on 1.15 (minimal), but averaging across the entire data set obscures this. It may be possible to further refine the normalisation stage of preparing data to better handle this difference in scales, perhaps adding weightings to make instances with longer serial elapsed times more important. However, the Simple Division model would suffer from much the same error, with the disparity between minimal and intensive metadata processing instances even more pronounced, and so the accuracy of the PPoDL remains superior.

6.3.1 Multicolinearity

As pointed out earlier, the coefficients for the PPoDL model are different than what would be intuitively expected. The most likely cause is for this is multicollinearity [62] as the model contains pairs of combinations of attributes with high correlation. A pair-wise correlation matrix between the models variables is shown in Table 6.5 with correlation values higher than 0.8 marked using boldface.

Table 6.5: Correlation matrix between PPoDL Model attributes

	IOPercent (IOP)	WeightedDivision (WDV)	Subscription (SBS)	SimpleDivision (SDV)	IsSerial (ISS)	IOFactor (IOF)	MBIOperS (MIS)	Cores (CRS)	Workers (WKS)
IOP	**1.00**								
WDV	0.77	**1.00**							
SBS	0.01	-0.49	**1.00**						
SDV	0.00	0.59	**-0.83**	**1.00**					
ISS	0.00	0.38	-0.49	0.63	**1.00**				
IOF	**0.89**	0.68	0.00	0.00	0.00	**1.00**			
MIS	0.75	0.58	0.00	0.00	0.00	**0.97**	**1.00**		
CRS	0.01	0.02	-0.02	0.01	0.00	0.01	0.01	**1.00**	
WKS	0.01	-0.40	**0.97**	-0.68	-0.32	0.00	0.00	0.01	**1.00**

In this model the collinearity occurs because some primary attributes are re-used multiple times in derived attributes. This property breaches the underlying assumption of independence of variables and is known to cause several issues including the inability to see a direct impact of attributes on the prediction, erratic or unexpected changes in estimated coefficients, potentially exaggerated standard errors, and data redundancy in attributes (which can result in over-fitting).

Table 6.6: Subsets of PPoDL Model attributes with lower correlation

Attributes	r	r^2	RMSE	MASE
PPoDL	0.9821	0.9645	0.1547	0.1348
Cores, IOPercent, IsSerial, MBIOperS, SimpleDivision, Workers	0.9124	0.9552	0.1738	0.1639
IOFactor, IsSerial, MBIOperS, WeightedDivision, Subscription	0.9059	0.9518	0.1803	0.2133

While there are several approaches to limiting multicollinearity, such as selecting independent subsets of attributes or obtaining more data, there is also an argument that a model exhibiting multicollinearity can be more accurate than one with attribute independence strictly enforced [78]. The values

provided in the correlation matrix immediately suggest two independent subsets of the PPoDL model's attributes; independent in that attributes have low correlation. Table 6.6 provides evidence that the accuracy measurements of these two subsets are poorer than those exhibited by the complete model. Thus we decided to retain the use of the PPoDL model despite the presence of multicollinearity.

6.4 Discussion

Using a combination of the R and Weka tools, this research has constructed a linear regression model from the experimental data as presented in Equation 6.7. With careful derivation, manipulation, and selection of attributes, aided by significance testing and automated selection algorithms, the resulting model was able to predict parallel processing performance at approximately 96% adjusted r^2 correlation and a relative absolute error of approximately 16%. Using the general accuracy measure MASE, the PPoDL model was five times more accurate than the Simple Division model. The PPoDL model also had performance directly comparable to models generated in Weka using automatic attribute selection techniques. While there is evidence that the PPoDL model exhibits the issue of multicollinearity of attributes, attempts to alter the model to enforce attribute independence resulted in poorer accuracy.

Chapter 7

Application of the Model

> In theory, there is no difference between theory and practice. In practice, however, there is.
>
> — *Johannes L. A. van de Snepscheut, 1984*

This chapter presents three worked examples showing how the Parallel Processing of Digital Libraries (PPoDL) model can be applied to real-world problems. The worked examples help illustrate the potential of a linear regression model to be re-arranged so as to predict attributes other than total elapsed import time.

To recap, the derived model is:

$$\begin{aligned} PredictedPerformance = {} & 0.0521 \\ & - 1.3018 \times IOPercent \\ & + 0.9276 \times WeightedDivision \\ & + 0.2386 \times IOFactor \\ & - 0.2305 \times IsSerial \\ & + 0.1920 \times SimpleDivision \\ & + 0.0176 \times MBIOperS \\ & + 0.0023 \times Subscription \end{aligned} \tag{7.1}$$

where:

$$IOFactor = \frac{(IOPercent \times 100)^3}{100,000}$$

$$MBIOperS = \frac{CollectionSize}{ElapsedSerial} \times IOPercent$$

$$SimpleDivision = \begin{cases} \frac{1}{Workers} & \text{if } Workers > 0 \\ 1 & \text{if } Workers = 0 \end{cases}$$

$$WeightedDivision = \begin{cases} \frac{1-IOPercent}{Workers} + IOPercent & \text{if } Workers > 0 \\ 1 & \text{if } Workers = 0 \end{cases}$$

$$Subscription = \frac{1}{Cores} + \left(|Workers - Cores| \times \frac{1}{Cores + 1} \right)$$

$$IsSerial = \begin{cases} 0 & \text{if } Workers > 0 \\ 1 & \text{if } Workers = 0 \end{cases}$$

The first example considers the broad question of the potential benefits of moving to parallel processing on a multi-core computer. The second example involves looking at the import of video files. It is more specific in terms of calculating actual processing times, and uses the model to address the three constraints that typically bound digital library importing: processing load, time, and hardware (see Section 1.3). The final example returns to the challenge that motivated this thesis, that of using parallel processing to accelerate the import of a newspaper collection allowing for more frequent updates.

7.1 Jamendo Audio Collection

The first example considers the question of determining the potential benefit of running a parallel import of a large scale audio collection—sourced from the Jamendo service—utilising all processors on a modern, 8 CPU computer. The number of files to be processed is not specified for this question: instead this question is more broad and is raised to determine whether purchasing a multi-core computer is worth the potential decrease in processing time. We further stipulate that the audio files will have some metadata extracted and will be transformed into a web-streamable format.

Table 7.1: Attributes for an audio collection import with typical processing

Attribute	Value
Cores	8
Workers	8
IOPercent	0.0657
WeightedDivision	0.1825
IOFactor	0.0028
IsSerial	0.0000
SimpleDivision	0.1250
MBIOperS	0.0339
Subscription	0.1250

Since an 8 core machine is being considered, the import will be configured with 8 worker threads so as to achieve balanced subscription to compute nodes. The measurements for this collection's "typical" processing load can be taken directly from the table in Appendix A. By using these values, and the formulas given above, we can generate all of the attribute values (shown in Table 7.1) needed to apply the model.

$$
\begin{aligned}
PredictedPerformance &= 0.0521 \\
&\quad - 1.3018 \times 0.0657 \\
&\quad + 0.9276 \times 0.1825 \\
&\quad + 0.2386 \times 0.0028 \\
&\quad - 0.2305 \times 0.0000 \\
&\quad + 0.1920 \times 0.1250 \\
&\quad + 0.0176 \times 0.0339 \\
&\quad + 0.0023 \times 0.1250 \\
&\quad = 0.1614
\end{aligned}
\tag{7.2}
$$

The attributes are substituted into the model's Equation 7.2, and an prediction for the relative performance of parallel processing calculated as 16.14% of elapsed serial processing time. The SimpleDivision model would predict a performance of 12.50%—simply dividing the workload into eight parts. Experimental test runs of this collection import configuration produced observed performance closer to 19.71% of serial time. Thus the PPoDL model's prediction exhibits an error of 3.57%, versus the SimpleDivision model's larger error of 7.21%.

7.2 ReplayMe! Video Collection

This next worked example shows how the model can be applied to the task of estimating the time taken to import multiple streams of broadcast quality video generated by the ReplayMe! system [155], and converting into web-streamable format, in real-time. Recall ReplayMe! which records all of New Zealand's sixteen free-to-view digital TV channels at once and stores them in a Greenstone digital library for on-demand access. As currently implemented, the recorded content is not streamable over the internet and can only be viewed on the computer where the recordings were made. Over a one hour period ReplayMe! produces sixteen hours of content (a mixture of standard and high definition video in raw transport stream format), one for each channel. Within the parallel processing system the content is split into ten minute segments, so the import will need to process ninety-six segments.

To apply the PPoDL model requires some input values that are most easily determined by importing a sample collection using a collection configuration

and on a computer with similar properties to the destination distributed environment. The values in Table 7.2 represent the measured results from building a smaller sample collection of six video segments. An alternative approach would have been to locate the most similar import situation in Appendix A, which contains exemplar measurements, in order to select appropriate values.

Table 7.2: Measured attributes of example collection

Attribute	Value
Sample segment count	6
Sample elapsed (seconds)	7,812
Elapsed per segment (seconds)	1,302
Total segments in collection	96
Total serial elapsed (seconds)	124,992
IOPercent	0.0183
MBIOperS	0.0048
Cores	12
Workers	12

Using the PPoDL model it is possible to answer a range of questions as proposed in Section 1.3. For example, consider the question of predicting the total time elapsed to build the collection while the processing cost and hardware remain fixed. The proposed importing hardware consists of a cluster of 12 low-end commodity computers similar to the one used for processing the sample. Following the optimal values suggested for parallel threads [77], 12 worker threads will be applied, resulting in 13 threads in total. The next step in applying the model is to calculate several further derived attributes as defined in Equation 8.1. The results of these calculations are shown in Table 7.3.

Table 7.3: Derived attributes of example collection

Attribute	Value
IOFactor	0.0001
SimpleDivision	0.0833
WeightedDivision	0.1001
Subscription	0.0833
IsSerial	0.0000

$$
\begin{aligned}
PredictedPerformance = {} & 0.0521 \\
& - 1.3018 \times 0.0062 \\
& + 0.9276 \times 0.1001 \\
& + 0.2386 \times 0.0001 \\
& - 0.2305 \times 0.0000 \\
& + 0.1920 \times 0.0833 \\
& + 0.0176 \times 0.0048 \\
& + 0.0023 \times 0.0833 \\
& = 0.1374
\end{aligned}
\tag{7.3}
$$

Substituting these attribute values into the PPoDL model gives Equation 7.3. The model predicts that the performance of a parallel import with 12 workers on 12 nodes is approximately 13.74% of a serial import of the same collection. Applying this to the estimated total serial elapsed time produces a predicted elapsed parallel time of approximately $17,200$ seconds. This means that an hour of recorded ReplayMe! media—which would have taken 35 hours to process serially—will take approximately 5 hours to process on a cluster of 12 such computers. When the ReplayMe! experiment was repeated in real life, the parallel elapsed time was measured to be approximately $21,900$ seconds or 17.54% of serial time, so in this case the model has an error of approximately 4.00%. By comparison, the Simple Division model prediction of 8.33% yields an error of 9.12%.

A second question, that of determining how many compute nodes are needed to allow ReplayMe! to utilise web-streamable conversion while operating in real-time, can also be calculated using the model. In order to achieve real-time processing, given the value for total serial elapsed time in Table 7.2, the predicted performance would need to be 2.88% of serial time. Both the Simple Division and PPoDL models can be mathematically rearranged to predict the required number of compute nodes. The rearranging of the Simple Division formula is straightforward but rearranging the PPoDL model is more complex, and requires the formulas for the derived attributes to be substituted before starting. Using these equations, the Simple Division model would predict that 35 compute nodes would achieve real-time processing. However, the rearranged PPoDL model has no solution for a *PredictedPerformance* of 2.88%. Investigation using the PPoDL model and method of inspection shows the maximum predicted speed-up peaks at 4.54% (even with a thousand nodes). The PPoDL

model thus predicts—given the current processing implementation, hardware, and network configuration—that there is no number of compute nodes can reach this performance. This result is due to the PPoDL model taking into account:

- the increasing overhead cost of managing parallel processing nodes (diminishing returns in adding more nodes);
- the increasing cost of file I/O as more and smaller video segments are required to prevent worker starvation; and
- the increasing contention and wait times and more nodes access the centralised file system.

This prediction is significant and indicates the configuration/hardware is simply not up to the task. Prompted by the result of applying the PPoDL model, the designers of the ReplayMe! system may be led to rethink their approach. In this particular circumstance, the changes might include a move to more modern computers with better performance, or perhaps a decentralisation of the point where TV signal is brought into the computer network. Instead of video being recorded on a single computer—which is then farmed out to the cluster (resulting in a file I/O bottleneck)—each node in the cluster could be equipped with a digital tuner and then feed the distributed set of video files generated into the parallel importing.

Finally, consider the third question of the maximum number of channels that can be recorded given a fixed resource of 12 compute nodes and a requirement of real-time processing, a detail useful in helping prioritize which channels to record. Using the result found for the first question, we know that 12 compute nodes provides a predicted performance of 13.74% serial elapsed time. This means that in an hour this cluster could import and convert approximately 7.28 hours of recorded video. Thus seven channels could be reasonably processed in real-time using this hardware configuration.

While the PPoDL model exhibits reasonable accuracy in the worked examples above, this is not always the case. Consider a collection with attributes similar to those shown in Table 7.2 except the *IOPercent* is 0.0062 (as is encountered when videos are converted to be streamable and have key-frame information automatically extracted). In this case the observed performance of parallel importing the collection is approximately 9.02% of elapsed serial time. In this case the PPoDL model would produce a prediction of 14.28%—substantially different than the Simple Division's more accurate prediction of

8.33%. However, Table 8.1 provides evidence that the PPoDL model is superior on average despite there being several instances where the Simple Division model outperforms it.

7.3 PapersPast Historic Newspaper Collection

Return now to the motivating example of the PapersPast collection, introduced in Section 1.2. Recall, documents in the collection consisted of a METS/ALTO XML file and one or more image files. The METS/ALTO format specifies both document structure as well as metadata and OCR extracted text. Any processing of the images happens later using a run-time server, so the majority of processing load during importing involves handling the OCR text. When this collection reached one million documents, it was taking approximately 10 days to import. Let us suppose that the focus for this question will be to decrease the import time to $\frac{1}{10}$ of serial time—to allow for daily updates—while minimising the cluster hardware cost. We assume the collection will be processed on cloud-based system which allows on-demand provisioning of computing power, and that we will always assign the same number of worker threads as processing nodes.

Table 7.4: Attributes for PapersPast collection

Attribute	Value
IOFactor	0.0086
IOPercent	0.0950
IsSerial	0.0000
MBIOperS	0.0041

A reasonable estimate of the collection size at this point would be 37 GB of XML files. If we further assume that the metadata processing load configured for this collection lay somewhere between typical and intensive text processing, then a reasonable estimate for *IOPercent* is 0.0950 leading to a value of 0.0041 for *MBIOperS*. These values, along with derived attributes values, are presented in Table 7.4.

$$
\begin{aligned}
PredictedPerformance = {} & 0.0521 \\
& - 1.3018 \times 0.0950 \\
& + 0.9276 \times 0.1646 \\
& + 0.2386 \times 0.0086 \\
& - 0.2305 \times 0.0000 \\
& + 0.1920 \times 0.0769 \\
& + 0.0176 \times 0.0041 \\
& + 0.0023 \times 0.0769 \\
& = 0.0982
\end{aligned}
\tag{7.4}
$$

Rather than rearranging the formula, this answer can be found by method of inspection. By substituting combinations of values for *Cores/Workers* into Equation 7.4, we can discover that at 13 workers the PPoDL model predicts the import would take approximately 0.0982 of serial time. Thus a cluster of 13 compute nodes would be sufficient to allow for a daily import of this collection.

7.4 Discussion

This chapter has provided several practical applications of the model in order to demonstrate its usefulness. The model is applied in several instances of collection building to predict an elapsed parallel processing time. By comparing with the actual observed result of building these collections, the predictions were shown to be more accurate than the SimpleDivision model—a naive divide the work by the computers approach—would have yielded.

The chapter also illustrated how the model could be used to predict other factors. By rearranging the formula, or by method of inspection, it is possible to explore the three constraints that typically bound any digital library import, namely: time, available hardware, and processing cost.

Indeed, one key divergence between the two models' predictions was highlighted when predicting the number of compute nodes of a set hardware specification that would be required to process a certain collection in real-time. The simple division predicted there was such a number, whereas the PPoDL indicated such performance could never be achieved with the specified hardware

and processing configuration.

While the PPoDL model generally was more accurate, there was at least one case illustrated where the simple divide the work approach produced a more accurate prediction. A reflection on this, and on other threats to validity, are presented in the concluding chapter.

Chapter 8

Conclusion

> Anything that happens, happens. Anything that, in happening, causes something else to happen, causes something else to happen. Anything that, in happening, causes itself to happen again, happens again.
>
> It doesn't necessarily do it in chronological order, though.
>
> — *Douglas Adams, "Mostly Harmless", 1992*

In this concluding chapter, we review and summarise the main findings of this research. The following section reflects upon any threats to validity of the model and issues raised by its use (*caveat utilitor*). This leads to a discussion on future work. Finally, we describe how the model addresses the hypothesis posed and provide some closing remarks.

8.1 Summary

The goal of this thesis, introduced in Chapter 1, was to test whether a predictive model could be developed that accurately predicts certain key features of applying parallel building to very large-scale digital library processing. Determining the potential benefit of parallel processing is not straightforward, however, due to complexities in applying parallel processing: scheduling and messaging overhead, suitability of algorithm to data, and underlying hardware technologies. Thus, informed by specific real-world challenges encountered during commercial digital library development, this thesis aimed (Section 1.3) to develop a predictive mathematical model capable of answering the practi-

cal questions that might arise in large-scale, parallel processing, digital library design.

Chapter 2 defined in detail what is meant by very large-scale digital libraries and how they relate to Big Data. Digital libraries paired with parallel processing provide one mechanism to cope with the explosion of raw data prevalent in this modern age of computing. This is especially true as multi-core computers, clusters, and cloud computing become more established. The chapter also described a number of case studies of contemporary very large-scale digital libraries, for instance the HathiTrust. These showed that many current efforts are utilising parallel processing techniques or are looking towards distributed computing as a solution to scale issues. Without exception, the parallel processing software in use was custom-built for that particular library domain, and was often proprietary. A secondary aim of this work, in addition to the development of a predictive model, sought to go beyond the use of bespoke solutions and provide parallel processing functionality to general purpose digital libraries. This chapter also provided an overview of the general purpose digital library environment, delineating the features that were directly pertinent to this research.

Chapter 3 focused on parallel processing, providing a definition of salient properties, and an overview of the two frameworks explored in this research: namely Open MPI and Hadoop. This chapter also discussed the general challenges faced when working with parallel processing, from synchronisation and file systems through to the difficulties that arise with something as seemingly straightforward as measuring time. An overview of the document corpora collected for use in experiments was also presented.

Chapter 4 provided implementation details for the three general purpose digital library software systems that were augmented with parallel processing capabilities. It underscored the match between the highly parallel nature of initial document import and the standard SPMD model for parallel processing. It also detailed the practical steps in implementing parallel building support within the chosen digital library software systems: Greenstone, DSpace, and Terrier. Several challenges specific to the digital libraries or frameworks involved were discussed.

In the experiments presented in Chapter 5, the practical application of parallel processing to very large-scale digital library building produced promising results. While this was most notable for media or collection configurations requiring substantially more processing work than file transfer, even in the case of high file transfer configurations there was still some improvement in

performance. The use of well established frameworks, OpenMPI and Hadoop, allows for parallel processing digital libraries to be readily deployed on various distributed computing platforms, including Beowulf clusters and the Cloud. Even in the case of standalone computers, multi-core hardware will continue to drastically improve for the near future, with both Intel and AMD planning 128 core chips before 2020. The ability to leverage this technological advance with appropriate use of software is vital when addressing the ever increasing amount of content being stored digitally.

As mentioned, the principle goal for this research was to create a practical model for predicting features of parallel processing: features such as the likely elapsed time for a given number of parallel processors as a percentage of the serial elapsed time. Chapter 6 detailed the creation of this model from the experimental data. It started by defining a baseline model, the Simple Division model, against which to benchmark a variety of developed models. The chapter then highlighted the pertinent attributes extracted from the data and the calculation of several derived attributes so as to better expose trends in the data. The tools Weka and R were utilised during this phase of development, and the resulting model took the form of a linear regression. The resulting PPoDL model is shown in Equation 8.1.

$$\begin{aligned} PredictedPerformance = {} & 0.0521 \\ & - 1.3018 \times IOPercent \\ & + 0.9276 \times WeightedDivision \\ & + 0.2386 \times IOFactor \\ & - 0.2305 \times IsSerial \\ & + 0.1920 \times SimpleDivision \\ & + 0.0176 \times MBIOperS \\ & + 0.0023 \times Subscription \end{aligned} \tag{8.1}$$

where:

$$IOFactor = \frac{(IOPercent \times 100)^3}{100,000}$$

$$MBIOperS = \frac{CollectionSize}{ElapsedSerial} \times IOPercent$$

$$SimpleDivision = \begin{cases} \frac{1}{Workers} & \text{if } Workers > 0 \\ 1 & \text{if } Workers = 0 \end{cases}$$

$$WeightedDivision = \begin{cases} \frac{1-IOPercent}{Workers} + IOPercent & \text{if } Workers > 0 \\ 1 & \text{if } Workers = 0 \end{cases}$$

$$Subscription = \frac{1}{Cores} + \left(|Workers - Cores| \times \frac{1}{Cores + 1} \right)$$

$$IsSerial = \begin{cases} 0 & \text{if } Workers > 0 \\ 1 & \text{if } Workers = 0 \end{cases}$$

Table 8.1: Summary statistics from SimpleDivision and PPoDL models (with metadata processing subsets)

	SimpleDivision	PPoDL
IOPercent	0.2613	0.2613
$\bar{a}$	0.5247	0.5247
Pearson's r	0.2215	0.9821
$\bar{r^2}$	0.0488	0.9645
RMSE	0.8600	0.1547
MASE	0.6513	0.1348

Table 8.1 provides the values for several typical accuracy metrics when applied to the Simple Division and PPoDL models, where the aim is to maximize correlation (r and r^2) while minimizing error (RMSE and MASE).

Comparing the first two columns shows that the PPoDL model exhibits higher correlation and lower error than the Simple Division model; the former

has 98% correlation compared to the latter's 22%. This observation needs to be tempered somewhat, as the correlation and relative metrics are arguably optimistic. There is an order of magnitude difference between performance values depending on a collection import's *IOPercent*—despite attempts to normalise the data—and these may be having an undue effect on accuracy. Evidence of this is seen when comparing the overall accuracy of this model with that of subsets formed from the three differing configurations of metadata extraction (and thus *IOPercent*) as shown in Table 6.4. For certain subsets there was a significant increase in relative error measures.

In summary, the accuracy of the PPoDL model was consistently superior when benchmarked against the intuitive, but naive, Simple Division model (Section 6.2). When compared by the generally applicable MASE measure, the PPoDL model's predictions were five times more accurate than the Simple Division's ones.

Finally, Chapter 7 demonstrated the application of the PPoDL model to three real-world digital library imports suffering from some form of large-scale challenge. While the PPoDL model once again out-performed the Simple Division model, a more interesting result occurred when attempting to predict the number of computers required to ingest a certain collection within a certain time. While the Simple Division model predicted there was some number of computers that could accomplish this, the PPoDL model—which factors in the diminishing return of adding more nodes—suggested that such an import was not possible given the properties of the computers to be utilised.

8.2 Threats to Validity

During the development of any model, certain assumptions are made or limitations encountered. This section discusses some of the potential threats to validity of the derived model and suggest ways to minimise the resulting risk.

At first glance an obvious weakness is that, for a model developed to tackle the question of very-large scale digital library creation, the experiments run did not seem to push into the very-large scale. For example, there are digital libraries already processing petabyte-sized data sets. However the largest collection here was small by comparison; only one million text documents totalling 2.86GB. However, we would address this perceived weakness in several ways:

- In the absence of other interfering factors—such as hardware concerns—the cost of processing simple text files will increase linearly with scale. We are confident that, assuming the text file processing remains highly partitionable, the PPoDL linear regression model will continue to be accurate as text collections scale.

- Experimentation on larger corpora would certainly have provided more convincing evidence of scalability, however there was a limiting factor in the time and hardware available to perform multiple imports of each collection. A serial import of the ten million Lorem Ipsum document collection took more than a month for each run. While some experimentation was carried out at this scale, not enough test runs were performed to produce stable and convincing numbers and thus this material was omitted from the model training data.

- As we attempted to build collections on this scale we began to uncover some unfortunate limitations in the digital libraries themselves, or in interactions with the underlying operating system. For example, the way Greenstone named the temporarily transformed documents during ingest happened to trigger a limitation on the number of subdirectories imposed by the file system. Rather than get side-tracked by the complexity of actually building collections of this scale, the decision was made to press forward with the results in hand.

A related weakness centres on the fact that only four media types were tested, and so the model may only perform well on those types. Arguably the media content has more impact on processing cost than any other feature; indeed earlier prototype models—developed during this research—would fixate on media type. Moreover, all media tested could be processed independent on one another, and thus were highly partitionable. We would concede that the model may not perform as well on other media types, and suggest that exemplar collections would need to be created and their *IOPercent* and *MBIOperS* metrics measured in order to determine the suitability of applying the PPoDL model.

Another potential weakness lies in the choice of hardware for performing the experimentation. This research focused on a Single Process, Multiple Data approach, which works best on a network cluster of homogeneous hardware. In contrast, the target audience for this model are less likely to have multiple computers with the same specifications, instead utilising a 'scavenger' grid made up of varying computing resources. The model developed purposely

avoiding including too many attributes related to the underlying hardware, and again we propose that exemplar collections would need to be imported and the metrics measured to determine *average* figures for the *IOPercent* and *MBIOperS* attributes to represent the overall performance of mixed machine clusters.

A more technical critique involves the question of whether the digital library implementations developed in this research correctly matched the intended use of the three chosen digital libraries. Certainly this research has already highlighted that Hadoop's MapReduce was not used to its full potential. That said, care was taken when implementing parallel processing to keep the implementations similar across all three libraries. The importing phase made use of the same external tools and extracted the same metadata, and the task of locating files to import—a function that differed significantly between the library software—was restructured to avoid skewing results.

A second technical question is raised by the worked examples in Chapter 7, which used a rearranged PPoDL model to predict or optimise other parameters. While the process of rearranging a linear regression formula can be completed in a stepwise fashion, the resulting formula can be surprisingly complex especially when accounting for the derived attributes. This is due to the repetition of primary attributes in the derived ones.

The choice of the SimpleDivision model to be used as a benchmark in comparisons could be seen as establishing a "straw model"; one that is easy to out-perform. In defence, this model was intended to capture how someone might naively consider the effects of parallel processing: a rule of thumb, or as in the quote "Ten people can pick cotton ten times as fast as one person." Indeed, the closer the processing load comes to perfectly partitionable, the more accurate a simple division of work would be. However most processing loads follow Amdahl's Law, with non-partitionable/serial portions, and so a simple division is overly optimistic. Imagine, for instance, if the ten cotton pickers only had one basket. Section 6.1 further justified the SimpleDivision's use since the typically-used benchmark models (ZeroR, OneR, Naïve Bayes etc), could not predict a numeric class or were too simple to reflect the wide range of possible values.

Another caveat for the current PPoDL model is that it is less accurate for those collection imports making use of intensive metadata processing and thus exhibiting a lower percentage of file transfer time. This is unfortunate as these imports take the most elapsed time, and thus are the ones that have the potential to benefit most from parallel processing. The issue is obscured

because these instances exhibit a significant difference in performance times from typical instances (see Section 6.3 for details). Possible ways to address this issue include:

- Revisiting the equations for normalising the data, weighting the instances with longer elapsed serial processing time as more important.
- Applying some form of pre-clustering or hierarchical modelling to separate these stand-out instances, although care would need to be taken to avoid over-fitting.
- Replacing the Ordinary Least Squares algorithm used for building the linear model with one that takes into account the substantially greater effect of errors in those imports with longer elapsed serial processing times.

Despite this, the PPoDL model is more accurate than a naive prediction of processing time, and thus remains a valuable contribution.

As previously mentioned, some attributes in the model are multicollinear. As well as obscuring the effect of attribute changes on the prediction, this property can cause issues with increased standard error, redundant data and over-fitting, and complicates the calculation of the inverse matrix used in some computing algorithms. However, Section 6.3 also provides evidence that the PPoDL model remains more accurate than models derived from subsets of its attributes so as to mitigate attribute correlation. Thus we choose to present the PPoDL model despite this potential drawback.

There is a general question about how applicable this model is to the target audience of smaller institutions, given the hardware and operating system level requirements for clustering of machines. While we would concede that establishing a cluster would require some technical skill, we would argue that it is becoming progressively easier. The Rocks-based cluster used in this research was created from a single self-installing CD image, and came with Hadoop pre-installed. Alternatively, cloud-based provisioning of parallel processing computers is becoming as easy as filling out a web form as per Amazon's Elastic MapReduce service. Moreover, the model can be used by those thinking about developing a cluster in order to determine if it will provide a practical solution to their problem. Consider the model's answer to the second part of the example in Section 7.2; this result would provide good reason to rethink the hardware in the cluster before it was even created.

Finally, a second general question about this model is, could it be used to predict the performance of other parallel systems? While the motivation and focus of this work is digital libraries, it is in fact the case that none of the attributes utilised in the model are specific to digital libraries. The model may very well prove useful for other parallel processing tasks with similar general properties: no dependency between processes, highly partitionable, and relatively basic use of the parallel framework with little inter-node communication or other overheads.

8.3 Future Work

The parallel processing framework implementation for Greenstone is available through a public code repository server,[1] configured as an extension to the main code base. Already, the parallel Greenstone implementation is being used for processing data extracted from the HathiTrust collection, and is being considered for use in the ReplayMe! and other multimedia-based systems. However, the implementations produced during the research are experimental in nature, and would benefit from further refinement and externalisation of configuration options. These improvements could, in turn, be used to further refine the predictive model.

While the model answers the most broad of hardware questions—"How many computers should be utilised?"—it purposely took a narrow view on other hardware issues such as networking and file system configuration. The investigations into the Hadoop Distributed File System (Section 3.4.3) illustrate that choice of file system and access protocols are critical to parallel performance. Moreover, the experiments show that the PPoDL model is currently less able to accurately predict the variability present in collection configurations with lower metadata processing and thus a higher percentage of elapsed time spent on file transfer. The variability in the underlying file systems had a significant effect on performance, and many attempts to mitigate this effect were made. Future work should incorporate research on networking and file transfer models to increase accuracy and provide hardware based attributes for more flexible configuration.

The experiments shown in this research focused on the task of parallel importing of the documents as an area that has not been studied as intensively as other aspects of very large-scale digital libraries (Section 4.1.2). Initial implementations of parallel indexing within Greenstone showed that the current

[1] http://svn.greenstone.org/

indexing process is not structured in a suitable way for extensive parallelisation. Indeed, the only indexer configuration that benefited significantly from a parallel implementation (MG) did so because it was less efficient and offered fewer functions. That is not to say that the indexing processes could not be altered so as to offer greater potential for parallel processing. Nor does it exclude the use of other indexing technologies that might prove more compatible with the SMPD model of parallel processing. These are areas worthy of further study.

Another avenue for future research would be to integrate more contemporary database/indexing solutions targeted at scalable architectures, and experiment on other distributed networks. For the former, some of this is seen in TrivialDB and Terrier, but further modern technologies such as Solr [160] or NoSQL databases [59] such as Apache Cassandra [106] hold the promise of more efficient operation. For the latter, utilising the portable features of the Open MPI framework opens the door to deploying parallel digital library building into scavenger networks [140] or cloud computing configurations.

8.4 Concluding Remarks

This research has explored the application of parallel processing to the import phase of several general purpose open-source digital libraries. From experiments run on these implementations, enough data was gathered to allow a predictive model to be built using automated tools (Weka and R). The resulting model's predictions have high correlation with the trend exhibited by experimental observations, with a Pearson's r of 0.98, while limiting relative average error to approximately 0.16 across a range of processing tasks.

The PPoDL model—whose accuracy when predicting key features of very large-scale digital libraries has been confirmed through empirical testing—is presented as a tangible outcome of this research in answer to the central hypothesis:

> A predictive model can be developed that accurately predicts when it is appropriate to apply parallel building to very large-scale digital library processing.

While there remain issues to address and refinements to be made to the model, this research has provided a valuable and practical tool when designing and building very large-scale digital libraries.

References

[1] Adams, D. (2007). Papers Past: Browse Access for Online New Zealand Newspapers. *Microform & Imaging Review*, *34*(1), 22–27.

[2] Aha, D. W., Kibler, D., Albert, M. K. (1991). Instance-Based Learning Algorithms. *Machine learning*, *6*(1), 37–66.

[3] Akaike, H. (1973). Information Theory and An Extension of the Maximum Likelihood Principle. In B. N. Petrov, B. F. C. (Ed.), *Second International Symposium on Information Theory*, pp. 267–281. Academiai Kiado: Budapest.

[4] Albanese, A. (2008). HathiTrust Is Launched. *Library Journal*, *133*(18), 13.

[5] Allison, J., Vernooij, J. R., Terpstra, J. H., *et al.* (2003). *Samba Manual, Chapter 17: File and Record Locking.* The Samba Project. [Last accessed: November 2015].
`http://www.samba.org/samba/docs/man/Samba-HOWTO-Collection/locking.html`

[6] Amati, G., Van Rijsbergen, C. J. (2002). Probabilistic Models of Information Retrieval Based on Measuring the Divergence from Randomness. *ACM Transactions on Information Systems (TOIS)*, *20*(4), 357–389.

[7] Ambacher, B., Ashley, K., Berry, J., *et al.* (2007). *Trustworthy Repositories Audit Certification (TRAC): Criteria and Checklist.* The Center for Research Libraries/Online Computer Library Center, Inc. [Last accessed: November 2015].
`http://www.crl.edu/content.asp?11=13&12=58&13=162&14=9`

[8] Amdahl, G. (1967). Validity of the Single Processor Approach to Achieving Large-Scale Computing Capabilities. In *AFIPS Conference Proceedings (30)*, pp. 483–485.

[9] Arlitsch, K. (2002). Digitizing Sanborn Fire Insurance Maps for a Full Color, Publicly Accessible Collection. *D-Lib Magazine*, *8*(7/8).

[10] Ashby, E. (1966). Collaboration Between Universities and Government Laboratories. *Minerva*, *4*(3), 406–407.

[11] Assisi, F. C. (2005). *Anurag Acharya Helped Google's Scholarly Leap*. INDOlink. [Last accessed: November 2015].
`http://www.indolink.com/SciTech/fr010305-075445.php`

[12] Atallah, M. J. (1998). *Algorithms and Theory of Computation Handbook*. CRC Press.

[13] Aulwes, R. T., Daniel, D. J., Desai, N. N., *et al.* (2004). Architecture of LA-MPI, a Network-Fault-Tolerant MPI. In *Proceedings of the 18th International Parallel and Distributed Processing Symposium*, pp. 15–24. IEEE.

[14] Bainbridge, D., Don, K., Buchanan, G., *et al.* (2004). Dynamic Digital Library Construction and Configuration. In Heery, R., Lyon, L. (Eds.), *Research and Advanced Technology for Digital Libraries*, vol. 3232 of *Lecture Notes in Computer Science*, pp. 1–13. Berlin: Springer-Verlag.

[15] Bainbridge, D., Witten, I. H., Boddie, S., *et al.* (2009). Stress-Testing General Purpose Digital Library Software. In *Proceedings of the 13th European Conference on Research and Advanced Technology for Digital Libraries*, ECDL'09, pp. 203–214. Berlin: Springer-Verlag.

[16] Barkes, J., Barrios, M. R., Cougard, F., *et al.* (1998). *GPFS: A Parallel File System*. IBM International Technical Support Organization.

[17] Barroso, L. A., Dean, J., Hölzle, U. (2003). Web Search for a Planet: The Google Cluster Architecture. *IEEE Micro*, *23*, 22–28.

[18] Baru, C., Moore, R., Rajasekar, A., *et al.* (1998). The SDSC Storage Resource Broker. In *Proceedings of the 1998 Conference of the Centre for Advanced Studies on Collaborative Research*, p. 5. IBM Press.

[19] Becker, D. J., Sterling, T., Savarese, D., *et al.* (1995). BEOWULF: A Parallel Workstation for Scientific Computation. In *Proceedings of the 24th International Conference on Parallel Processing*, pp. 11–14. CRC Press.

[20] Beguelin, A., Dongarra, J., Geist, A., *et al.* (1991). *A User's Guide to PVM Parallel Virtual Machine*. University of Tennessee.

[21] Bellard, F. (2005). *FFmpeg Multimedia System*. FFmpeg. [Last accessed: November 2015].
`https://www.ffmpeg.org/about.html`

[22] Bellman, R. (1957). *Dynamic Programming*. Princeton University Press.

[23] Bhandarkar, M., Castain, R., Tan, W. (2011). *Hamster: Hadoop and MPI on the Same Cluster*. The Apache Software Foundation. [Last accessed: November 2015].
`https://issues.apache.org/jira/browse/MAPREDUCE-2911`

[24] Boddie, S. (2007). *Veridian: The Software That Brings Digital Collections to Life*. DL Consulting Ltd. [Last accessed: November 2015].
`http://veridiansoftware.com/`

[25] Boddie, S. (2013). *Elephind: Search the World's Historic Newspaper Archives*. DL Consulting Ltd. [Last accessed: November 2015].
`http://www.elephind.com/`

[26] Borthakur, D. (2007). The Hadoop Distributed File System: Architecture and Design. *Hadoop Project Website*, *11*(2007), 21.

[27] Box, D., Ehnebuske, D., Kakivaya, G., *et al.* (1999). SOAP: Simple Object Access Protocol. *HTTP Working Group Internet Draft*, p. 42.

[28] Boyer, E. B., Broomfield, M. C., Perrotti, T. A. (2012). GlusterFS One Storage Server to Rule Them All. Tech. rep., Los Alamos National Laboratory.

[29] Brachman, B., Neufeld, G. (1992). TDBM: A DBM Library with Atomic Transactions. In *Summer '92 USENIX*, vol. 92, pp. 63–80.

[30] Breeding, M. (2006). PINES Sets Precedent for Open-Source ILS. *Smart Libraries Newsletter*, *26*(10).

[31] Brin, S., Page, L. (1998). The Anatomy of a Large-Scale Hypertextual Web Search Engine. In *Seventh International World-Wide Web Conference*. Computer Science Department, Standford University.

[32] Bunce, T. (2008). *New York Times Profiler*. CPAN. [Last accessed: November 2015].
`http://search.cpan.org/perldoc?Devel%3A%3ANYTProf`

[33] Burns, G., Daoud, R., Vaigl, J. (1994). LAM: An Open Cluster Environment for MPI. In *Proceedings of Supercomputing Symposium*, pp. 379–386.

[34] Burton-West, T. (2011). *Forty Days and Forty Nights: Re-indexing 7+ Million Books (part 1)*. HathiTrust Digital Library. [Last accessed: November 2015].

http://www.hathitrust.org/blogs/large-scale-search/forty-days-and-forty-nights-re-indexing-7-million-books-part-1

[35] Buse, A. (1973). Goodness of Fit in Generalized Least Squares Estimation. *The American Statistician*, *27*(3), 106–108.

[36] Butler, B. (2013). *Talk about Big Data: How the Library of Congress can index all* 170 *billion tweets ever posted.* Network World. [Last accessed: November 2015]. http://www.networkworld.com/news/2013/010813-loc-tweets-265627.html

[37] Byron, A., Berry, A., Haug, N., *et al.* (2008). *Using Drupal.* O'Reilly Media, Inc.

[38] Caffaro, J., Kaplun, S. (2010). Invenio: A Modern Digital Library for Grey Literature. Tech. rep., European Organization for Nuclear Research. No. CERN-OPEN-2010-027.

[39] Candela, L., Akal, F., Avancini, H., *et al.* (2007). DILIGENT: integrating digital library and Grid technologies for a new Earth observation research infrastructure. *International Journal on Digital Libraries*, *7*(1-2), 59–80.

[40] Candela, L., Castelli, D., Ferro, N., *et al.* (2008). *The DELOS Digital Library Reference model. Foundations for digital Libraries (Version 0.98).* ISTI-CNR at Gruppo ALI.

[41] Candela, L., Castelli, D., Pagano, P. (2007). A Reference Architecture for Digital Library Systems: Principles and Applications. In *Proceedings of the 1st International Conference on Digital Libraries: Research and Development*, DELOS'07, pp. 22–35. Berlin: Springer-Verlag. ISBN 3-540-77087-9, 978-3-540-77087-9.

[42] Caplan, P., Guenther, R. S. (2005). Practical Preservation: The PREMIS Experience. *Library Trends*, *54*(1), 111–124.

[43] Castelli, D. (2004). DILIGENT: A Digital library infrastructure on grid enabled technology. *ERCIM News,(59)*, pp. 26–27.

[44] Chang, F., Dean, J., Ghemawat, S., *et al.* (2006). Bigtable: A Distributed Storage System for Structured Data. In *Seventh Conference of USENIX Symposium on Operating Systems Design and Implementation*, vol. 7, pp. 205–218.

[45] Chapman, B., Jost, G., Van Der Pas, R. (2008). *Using OpenMP: Portable Shared Memory Parallel Programming*, vol. 10. MIT press.

[46] Chen, P. M., Lee, E. K., Gibson, G. A., *et al.* (1994). RAID: High-Performance, Reliable Secondary Storage. *ACM Computing Surveys*, *26*, 145–185.

[47] Christel, M., Kanade, T., Mauldin, M., *et al.* (1995). Informedia Digital Video Library. *Communications of the ACM*, *38*(4), 57–58.

[48] Claverie-Berge, I. (2012). *Solutions Big Data*. IBM Information On Demand. IBM. [Last accessed: November 2015].
`http://www-05.ibm.com/fr/events/netezzaDM_2012/Solutions_Big_Data.pdf`

[49] Cortez, P. (2010). Data Mining with Neural Networks and Support Vector Machines Using the R/Rminer Tool. In *Advances in Data Mining. Applications and Theoretical Aspects*, pp. 572–583. Berlin: Springer-Verlag.

[50] Cox, M., Ellsworth, D. (1997). Managing Big Data for Scientific Visualization. In *ACM Siggraph*, vol. 97, pp. 146–162. MRJ/NASA Ames Research Center.

[51] Crockford, D., Bray, T. (2014). The JavaScript Object Notation (JSON) Data Interchange Format. RFC 7159, Internet Engineering Task Force.

[52] Cundiff, M. V. (2004). An Introduction to the Metadata Encoding and Transmission Standard (METS). *Library Hi Tech*, *22*(1), 52–64.

[53] Darema, F. (2001). The SPMD model: Past, Present and Future. In Cotronis, Y., Dongarra, J., et al. (Eds.), *Recent Advances in Parallel Virtual Machine and Message Passing Interface*, vol. 2131 of *Lecture Notes in Computer Science*. Santorini/Thera, Greece: Springer-Verlag.

[54] Dean, J., Ghemawat, S. (2004). MapReduce: Simplified Data Processing on Large Clusters. In *OSDI'04: Sixth Symposium on Operating System Design and Implementation*, pp. 137–150. OSDI.

[55] DeBergalis, M. S. (2000). *A Parallel File I/O API for Cilk*. Ph.D. thesis, Massachusetts Institute of Technology.

[56] Donohue, T. (2012). *DSpace Release 1.8.2 Notes*. Duraspace. [Last accessed: November 2015].
`https://wiki.duraspace.org/display/DSPACE/DSpace+Release+1.8.2+Notes`

[57] Donohue, T., Luyten, B. (2013). *DSpace 4.x Documentation: Mediafilters for Transforming DSpace Content*. Duraspace. [Last accessed: November 2015].
`https://wiki.duraspace.org/display/DSDOC4x/Mediafilters+for+Transforming+DSpace+Content`

[58] El-Abbadi, M. (1992). *The Life and Fate of the Ancient Library of Alexandria*. Unesco/UNDP.

[59] Evans, E. (2009). *Nosql: What's in a Name*. Eric Evans's Weblog. [Last accessed: November 2015].
`http://blog.sym-link.com/2009/10/30/nosql_whats_in_a_name.html`

[60] Fagg, G. E., Dongarra, J. J. (2000). FT-MPI: Fault Tolerant MPI, Supporting Dynamic Applications in a Dynamic World. In *Recent Advances in Parallel Virtual Machine and Message Passing Interface*, pp. 346–353. Berlin: Springer-Verlag.

[61] Farber, P. (2009). *Large-Scale Full-Text Indexing with Solr*. HathiTrust Digital Library. [Last accessed: November 2015].
`http://www.hathitrust.org/blogs/large-scale-search/large-scale-full-text-indexing-solr`

[62] Farrar, D. E., Glauber, R. R. (1967). Multicollinearity in Regression Analysis: The Problem Revisited. *The Review of Economic and Statistics*, pp. 92–107.

[63] Fisher, R. A. (1934). *Statistical Methods for Research Workers*. Edinburgh.

[64] Fisher, R. A., et al. (1920). A Mathematical Examination of the Methods of Determining the Accuracy of an Observation by the Mean Error, and by the Mean Square Error. *Monthly Notices of the Royal Astronomical Society*, *80*, 758–770.

[65] Fleischman, E. (1998). Wave and avi codec registries. RFC 2361, Internet Engineering Task Force.

[66] Flynn, M. (1972). Some Computer Organizations and Their Effectiveness. *Computers, IEEE Transactions on*, *C-21*(9), 948–960.

[67] Frew, J., Freeston, M., Kemp, R. B., *et al.* (1996). The Alexandria Digital Library Testbed. *D-Lib Magazine*, *2*.

[68] Frigo, M., Johnson, S. G. (1998). FFTW: An Adaptive Software Architecture for the FFT. In *Proceedings of the IEEE International Conference on Acoustics, Speech and Signal Processing*, vol. 3, pp. 1381–1384. IEEE.

[69] Gabriel, E., Fagg, G. E., Bosilca, G., *et al.* (2004). Open MPI: Goals, Concept, and Design of a Next Generation MPI Implementation. In *Proceedings of the 11th European PVM/MPI Users' Group Meeting*, pp. 97–104. Budapest, Hungary.

[70] Gabriel, E., Resch, M., Beisel, T., *et al.* (1998). Distributed computing in a heterogeneous computing environment. In Alexandrov, V., Dongarra, J. (Eds.), *Recent Advances in Parallel Virtual Machine and Message Passing Interface*, vol. 1497 of *Lecture Notes in Computer Science*, pp. 180–187. Berlin: Springer-Verlag. ISBN 978-3-540-65041-6.

[71] Gagliardi, F. (2005). The EGEE European grid infrastructure project. In *High Performance Computing for Computational Science-VECPAR 2004*, pp. 194–203. Berlin: Springer-Verlag.

[72] Gantt, H. (1910). Work, Wages and Profit. *The Engineering Magazine*.

[73] Gantz, J. F., Reinsel, D. (2012). The Digital Universe in 2020: Big Data, Bigger Digital Shadows, and Biggest Growth in the Far East. Tech. rep., IDC.

[74] Giles, C. L., Bollacker, K. D., Lawrence, S. (1998). CiteSeer: An Automatic Citation Indexing System. In *Proceedings of the Third ACM Conference on Digital Libraries*, pp. 89–98. ACM.

[75] Giles, C. L., Bollacker, K. D., Lawrence, S. (1998). CiteSeer: An automatic citation indexing system. In *Proceedings of the third ACM conference on Digital libraries*, pp. 89–98. ACM.

[76] Giles, J. (2005). Science in the Web Age: Start Your Engines. *Nature*, *438*, 554–555.

[77] Goetz, B. (2002). Java Theory and Practice: Thread Pools and Work Queues. Tech. Rep. j-jtp0730, IBM, New York, United States.

[78] Graham, M. H. (2003). Confronting Multicollinearity in Ecological Multiple Regression. *Ecology*, *84*(11), 2809–2815.

[79] Grotke, A. M. (2004). Creating Access Points to Thematic Web Collections. In *Archiving Conference*, 1, pp. 18–21. Society for Imaging Science and Technology.

[80] Guo, Y., Rao, J., Zhou, X. (2013). iShuffle: Improving Hadoop Performance with Shuffle-on-Write. In *Proceedings, 10th International Conference on Autonomic Computing*, pp. 107–117. USENIX Association.

[81] Gustafson, J. L. (1988). Reevaluating Amdahl's Law. *Communications of the ACM*, *31*, 532–533.

[82] Hatcher, E., Gospodnetic, O., McCandless, M. (2004). *Lucene in action*. Manning Publications Greenwich, CT.

[83] Hawking, D., Craswell, N. (2004). Overview of the TREC-2004 Web Track. In *TREC*. NIST.

[84] Hawking, D., Voorhees, E., Craswell, N., *et al.* (1999). Overview of the TREC-8 Web Track. In *TREC*. NIST.

[85] Heesch, D., Pickering, M., Rüger, S., *et al.* (2003). Video Retrieval Using Search and Browsing with Key Frames. In *Proceedings of TRECVID 2003, NIST*.

[86] Hibbard, B., Paul, B. (1994). Case Study #4: Examining Data Sets in Real Time, VIS-5D and WIS-AD for Visualizing Earth and Space Science Computations. In *Course #27, Visualizing and Examining Large Scientific Data Sets: A Focus on the Physical and Natural Sciences*, SIGGRAPH'94.

[87] Higley, G., Mission, N. (2006). The National Digital Newspaper Program (NDNP)-An NEH/LC Collaborative Program. *IFLA Publications*, *118*, 113.

[88] Hildebrand, D., Honeyman, P. (2005). Exporting Storage Systems in a Scalable Manner with PNFS. In *Proceedings of 22nd IEEE/13th NASA Goddard Conference on Mass Storage Systems and Technologies*, pp. 18–27. IEEE.

[89] Hirabayashi, H., Hirabayashi, M. (2007). *Benchmark Test of DBM Brothers*. Tokyo, Japan: FAL Labs. [Last accessed: November 2015]. `http://tokyocabinet.sourceforge.net/benchmark.pdf`

[90] Hirabayashi, H., Hirabayashi, M. (2011). *Kyoto Cabinet: A Straightforward Implementation of DBM*. FAL Labs, Tokyo, Japan. [Last accessed: November 2015]. `http://fallabs.com/kyotocabinet/`

[91] Holmes, G., Donkin, A., Witten, I. H. (1994). Weka: A machine learning workbench. In *Proceedings of the 1994 Second Australian and New Zealand Conference on Intelligent Information Systems*, pp. 357–361. IEEE.

[92] Holte, R. C. (1993). Very Simple Classification Rules Perform Well on Most Commonly Used Datasets. *Machine Learning*, *11*(1), 63–90.

[93] Huang, M., Du, W. (2011). A Service Chain for Digital Library Based on Cloud Computing. In *Knowledge Engineering and Management*, pp. 261–266. Berlin: Springer-Verlag.

[94] Huang, X., Alleva, F., Hon, H.-W., *et al.* (1993). The SPHINX-II Speech Recognition System: An Overview. *Computer Speech & Language*, *7*(2), 137–148.

[95] Huckman, R. S., Pisano, G. P., Kind, L. (2008). Amazon Web Services. *Harvard Business School Case*, *2008*(609-048).

[96] Hyndman, R. J., Athanasopoulos, G. (2014). *Forecasting: Principles and Practice*. OTexts.

[97] Jones, K. S. (1972). A Statistical Interpretation of Term Specificity and its Application in Retrieval. *Journal of Documentation*, *28*(1), 11–21.

[98] Kahle, B. (1997). Preserving the internet. *Scientific American*, *276*(3), 82–83.

[99] Kahle, B. (2004). *Large Scale Data Repository: Petabox*. Internet Archive. [Last accessed: November 2015].
`https://archive.org/web/petabox.php`

[100] Kincaid, J. P., Fishburne Jr, R. P., Rogers, R. L., *et al.* (1975). Derivation of New Readability Formulas (Automated Readability Index, Fog Count and Flesch Reading Ease Formula) for Navy Enlisted Personnel. Tech. rep., DTIC Document.

[101] Kofler, M. (2001). What Is MySQL? In *MySQL*, pp. 3–19. Apress. ISBN 978-1-893115-57-6.

[102] Kovalenko, V. N., Koryagin, D. A. (2009). The grid: Analysis of basic principles and ways of application. *Programming and Computer Software*, *35*(1), 18–34.

[103] Kuć, R. (2013). *Apache Solr 4 Cookbook*. Packt Publishing Ltd.

[104] de Kunder, M. (2014). *The Size of the World Wide Web (The Internet).* WorldWideWebSize. [Last accessed: November 2015]. http://www.worldwidewebsize.com/

[105] Kunze, J., Arvidson, A., Mohr, G., *et al.* (2006). The WARC File Format (Version 0.9). *IIPC Framework Working Group.*

[106] Lakshman, A., Malik, P. (2010). Cassandra: a Decentralized Structured Storage System. *ACM SIGOPS Operating Systems Review*, *44*(2), 35–40.

[107] Lamolinara, G., Lohr, C., Mills, Q. (1998). *ALEXA Internet Donates Archive of the World Wide Web To Library of Congress.* The Library of Congress.

[108] Laney, D. (2001). 3D Data Management: Controlling Data Volumes, Velocity, and Variety. *Application Delivery Strategies.*

[109] Layton, J. B. (2010). *I/O Profiling of Applications: Strace_Analyzer.* Linux Magazine. [Last accessed: November 2015]. http://www.linux-mag.com/id/7730/

[110] Lester, N., Moffat, A., Zobel, J. (2005). Fast On–Line Index Construction by Geometric Partitioning. In *Proceedings of the 14th ACM Conference on Information and Knowledge Management*, pp. 776–783.

[111] Li, H., Councill, I., Lee, W.-C., *et al.* (2006). CiteSeerX: An Architecture and Web Service Design for an Academic Document Search Engine. In *Proceedings of the 15th International Conference on World Wide Web*, pp. 883–884. ACM.

[112] Li, X., Furht, B. (1999). Design and Implementation of Digital Libraries. In *Proceedings of IASTED International Conference on Internet, Multimedia Systems and Applications*, pp. 213–217. IASTED/ACTA Press.

[113] Li, Y. (2006). *Analyze and Optimize Cloud Cluster Performance.* IBM developerWorks. [Last accessed: November 2015]. http://www.ibm.com/developerworks/cloud/library/cl-cloudclusterperformance/

[114] Lichti, C., Faloutsos, C., Wactlar, H., *et al.* (1998). Informedia: Lessons from a Terabyte+ Operational, Digital Video Database System. In *Proceedings of the 1998 Conference on Very Large Databases*, pp. 24–17. VLDB Foundation.

[115] Littman, J. (2006). A Technical Approach and Distributed Model for Validation of Digital Objects. *D-Lib Magazine*, *12*(5), 3.

[116] Littman, J. (2007). Actualized Preservation Threats: Practical Lessons from Chronicling America. *D-Lib Magazine*, *13*(7), 5.

[117] Litzkow, M., Livny, M. (1990). Experience with the Condor distributed batch system. In *Experimental Distributed Systems, 1990. Proceedings., IEEE Workshop on*, pp. 97–101. IEEE.

[118] Loney, K. (2004). *Oracle Database 10g: The Complete Reference*. McGraw-Hill/Osborne.

[119] Lowe, D. G. (1999). Object Recognition from Local Scale-Invariant Features. In *The Proceedings of the Seventh IEEE International Conference on Computer Vision*, vol. 2, pp. 1150–1157. IEEE.

[120] Lyman, P. (2002). Archiving the World Wide Web. *Building a National Strategy for Digital Preservation: Issues in Digital Media Archiving*, pp. 38–51.

[121] Lynch, C. A. (1991). The Z39.50 Information Retrieval Protocol: An Overview and Status Report. *SIGCOMM Comput. Commun. Rev.*, *21*(1), 58–70.

[122] Maindonald, J., Braun, J. (2006). *Data analysis and graphics using R: an example-based approach*, vol. 10. Cambridge University Press.

[123] Manghi, P., Pagano, P., Ioannidis, Y. (2010). Second Workshop on Very Large Digital Libraries: in Conjunction with the European Conference on Digital Libraries. *SIGMOD Rec.*, *38*, 46–48.

[124] Maron, M. E. (1961). Automatic Indexing: an Experimental Inquiry. *Journal of the ACM (JACM)*, *8*(3), 404–417.

[125] Masár, I. (2014). *DSpace: Solr*. Duraspace. [Last accessed: November 2015].
`https://wiki.duraspace.org/display/DSPACE/Solr`

[126] Mattos, N., Deßloch, S., DeMichiel, L., *et al.* (1996). Object-Relational DB2. *IBM White Paper*.

[127] McCallum, S. H. (2004). An Introduction to the Metadata Object Description Schema (MODS). *Library Hi Tech*, *22*(1), 82–88.

[128] McCreadie, R., Macdonald, C., Ounis, I. (2012). MapReduce Indexing Strategies: Studying Scalability and Efficiency. *Information Processing & Management*, *48*(5), 873–888. Large-Scale and Distributed Systems for Information Retrieval.

[129] McCreadie, R. M. C., Macdonald, C., Ounis, I. (2009). On Single-pass Indexing with MapReduce. In *Proceedings of the 32nd International ACM SIGIR Conference on Research and Development in Information Retrieval*, SIGIR '09, pp. 742–743. New York, NY, USA: ACM.

[130] McHenry, O. (2008). EuropeanaLocal - Its Role in Improving Access to Europe's Cultural Heritage Through the European Digital Library. In *11th Annual International Conference (EVA Moscow)*.

[131] McHugh, A., Ruusalepp, R., Hofman, H., *et al.* (2007). *Digital Repository Audit Method Based on Risk Assessment (DRAMBORA)*. DRAMBORA Consortium. [Last accessed: November 2015]. http://www.repositoryaudit.eu/

[132] McLaughlin, M. P. (1993). *"...the very game..." - A Tutorial on Mathematical Modeling*. causaScientia. [Last accessed: November 2015]. http://www.causascientia.org/math_stat/Tutorial.pdf

[133] McMillan, G. (2013). *Huzzah! Library of Congress' Useless Twitter Archive is Almost Complete... But You can't Read it Yet*. DigitalTrends. [Last accessed: November 2015]. http://www.digitaltrends.com/social-media/library-of-congress-useless-twitter-archive-is-almost-complete/

[134] Mearian, L. (2009). *Internet Archive to unveil massive Wayback Machine data center*. Computerworld. [Last accessed: November 2015]. http://www.computerworld.com/article/2531864/data-center/internet-archive-to-unveil-massive-wayback-machine-data-center.html

[135] Miller, R. (2011). *Report: Google Uses About 900,000 Servers*. Data Center Knowledge. [Last accessed: November 2015]. http://www.datacenterknowledge.com/archives/2011/08/01/report-google-uses-about-900000-servers/

[136] Misra, D., Seamans, J., Thoma, G. R. (2008). Testing the Scalability of a DSpace-based archive. In *Proceedings of the Society for Imaging Sciences and Technology Archiving*, IS&T-A'08, pp. 36–40. National Library of Medicine, United States.

[137] Mohr, G., Stack, M., Rnitovic, I., *et al.* (2004). Introduction to Heritrix, an archival quality web crawler. In *4th International Web Archiving Workshop*.

[138] Moore, G. E. (1965). Cramming More Components onto Integrated Circuits. *Electronics (Magazine)*, *38*.

[139] Moroo, J., Yamada, M., Kato, T. (2012). Operating System for the K computer. *Fujitsu Sci. Tech. J*, *48*(3), 295–301.

[140] Nakashole, N., Suleman, H. (2009). A Hybrid Distributed Architecture for Indexing. In *Proceedings of the 13th European Conference on Research and Advanced Technology for Digital Libraries*, ECDL'09, pp. 250–260. Berlin: Springer-Verlag.

[141] Nelson, M. L., Sompel, H. V. d., Warner, S. (2002). Advanced Overview of Version 2.0 of the Open Archives Initiative Protocol for Metadata Harvesting. In *Proceedings of the 2nd ACM/IEEE-CS Joint Conference on Digital Libraries*, pp. 418–418. ACM.

[142] Osterberg, G., Allen, E. (2013). *Update on the Twitter Archive At the Library of Congress*. The Library of Congress. [Last accessed: November 2015].
`http://www.loc.gov/today/pr/2013/files/twitter_report_2013jan.pdf`

[143] Ounis, I., Amati, G., Plachouras, V., *et al.* (2006). Terrier: A High Performance and Scalable Information Retrieval Platform. In *Proceedings of ACM SIGIR '06 Workshop on Open Source Information Retrieval*, pp. 18–25. OSIR.

[144] Papadopoulos, P., Bruno, G., Katz, M. (2007). Beyond Beowulf clusters. *Queue*, *5*(3), 36–43.

[145] Payette, S., Lagoze, C. (1998). Flexible and Extensible Digital Object and Repository Architecture (FEDORA). In *Research and Advanced Technology for Digital Libraries*, pp. 41–59. Berlin: Springer-Verlag.

[146] Plimpton, S. J., Devine, K. D. (2011). MapReduce in MPI for Large-Scale Graph Algorithms. *Parallel Computing*, *37*(9), 610–632.

[147] Preston, J., Moses, S.-A. (2008). eJamaica.org, a DSpace driven digital repository to promote the sharing of information. In *Third International Conference on Open Repositories 2008*.
`http://pubs.or08.ecs.soton.ac.uk/121/`

[148] Quinlan, J. R., et al. (1992). Learning with continuous classes. In *Proceedings of the 5th Australian joint Conference on Artificial Intelligence*, vol. 92, pp. 343–348. Singapore.

[149] R Development Core Team (2011). *R: A Language and Environment for Statistical Computing*. R Foundation for Statistical Computing, Vienna, Austria. ISBN 3-900051-07-0.

[150] Rasmussen, E. M. (1991). Introduction: Parallel Processing and Information Retrieval. *Inf. Process. Manage.*, *27*, 255–263.

[151] Raymond, M. (2010). *How Tweet It Is!: Library Acquires Entire Twitter Archive*. The Library of Congress. [Last accessed: November 2015]. `http://blogs.loc.gov/loc/2010/04/how-tweet-it-is-library-acquires-entire-twitter-archive/`

[152] Reschke, C., Sterling, T., Ridge, D., *et al.* (1996). A Design Study of Alternative Network Topologies for the Beowulf Parallel Workstation. In *Proceedings of the Fifth IEEE International Symposium on High Performance Distributed Computing*, pp. 626–635.

[153] Robertson, S., Walker, S., Jones, S. (1995). Okapi at TREC-3. In *The Third Text Retrieval Conference (TREC-3)*, pp. 109–126.

[154] Ronstrom, M., Thalmann, L. (2004). MySQL Cluster Architecture Overview. *MySQL Technical White Paper*.

[155] Roüast, M., Bainbridge, D. (2012). Live Television in a Digital Library. In *Proceedings of the 12th ACM/IEEE-CS Joint Conference on Digital Libraries*, JCDL'12, pp. 81–90. ACM.

[156] Sammer, E. (2012). *Hadoop Operations*. O'Reilly Media, Inc.

[157] Sanderson, R., Larson, R. R. (2006). Indexing and Searching Tera-scale Grid-Based Digital Libraries. In *Proceedings of the 1st International Conference on Scalable Information Systems*, InfoScale '06. New York, NY, USA: ACM. ISBN 1-59593-428-6.

[158] Schneider, S. M., Foot, K., Kimpton, M., *et al.* (2003). Building Thematic Web Collections: Challenges and Experiences from the September 11 Web Archive and the Election 2002 Web Archive. In *3rd Workshop of Web Archives in Conjunction with the 7th European Conference on Digital Libraries*, Digital Libraries, ECDL, pp. 77–94.

[159] Schwan, P. (2003). Lustre: Building a File System for 1000-Node Clusters. In *Proceedings of the Linux Symposium*, vol. 2003.

[160] Seeley, Y. (2006). Apache Solr. *ApacheCon Europe 2006*.

[161] Seltzer, M. I., Yigit, O. (1991). A New Hashing Package for UNIX. In *USENIX Winter*, pp. 173–184.

[162] Shannon, C. E. (1948). A Mathematical Theory of Communication. In *Bell System Technical Journal*, vol. 27, pp. 379–423, 623–656.

[163] Slee, M., Agarwal, A., Kwiatkowski, M. (2007). Thrift: Scalable Cross-Language Services Implementation. *Facebook White Paper*, *5*(8).

[164] Smiley, D., Pugh, E. (2009). *Solr 1.4 Enterprise Search Server*. Packt Publishing Ltd.

[165] Smith, M., Barton, M., Bass, M., *et al.* (2003). *DSpace: An Open Source Dynamic Digital Repository*. Corporation for National Research Initiatives.

[166] Squyres, J. (2008). *Open MPI:* 10^5 *Flops Can't Be Wrong*. CISCO Systems. [Last accessed: November 2015].
`http://www.open-mpi.org/papers/sc-2008/jsquyres-cisco-booth-talk-1up.pdf`

[167] Stanfill, C. (1990). Partitioned Posting Files: A Parallel Inverted File Structure for Information Retrieval. In *Proceedings of the 13th Annual International ACM SIGIR Conference on Research and Development in Information Retrieval*, SIGIR '90, pp. 413–428. New York, NY, USA: ACM. ISBN 0-89791-408-2.

[168] Staples, T., Wayland, R., Payette, S. (2003). The Fedora Project: An Open-Source Digital Object Repository Management System. *D-Lib Magazine*, *9*(4).

[169] Stone, L., Hollister, V. (2010). *DSpace Development Areas: Bitstream Relationships*. Duraspace. [Last accessed: November 2015].
`https://wiki.duraspace.org/display/DSPACE/BitstreamRelationships`

[170] Stonebraker, M., Rowe, L. A. (1986). The Design of POSTGRES. In *Proceedings of the 1986 ACM SIGMOD International Conference on Management of Data*, pp. 340–355. New York, NY, USA: ACM.

[171] Targett, C. (2013). *Apache Solr 4.7 Reference Guide: Near Real-Time Searching*. Apache.

[172] Thompson, J., Bainbridge, D., Roüast, M. (2012). Parallel Processing Videos in Very Large Digital Libraries. In *The Outreach of Digital Libraries: A Globalized Resource Network*, pp. 219–228. Berlin: Springer-Verlag.

[173] Thompson, J., Bainbridge, D., Suleman, H. (2011). Towards Very Large Scale Digital Library Building in Greenstone Using Parallel Processing. In *Proceedings, 13th International Conference on Asia-Pacific Digital Libraries*, vol. 7008 of *Lecture Notes in Computer Science*, pp. 331–340. Berlin: Springer-Verlag.

[174] Thompson, J., Bainbridge, D., Suleman, H. (2011). Using TDB in Greenstone to Support Scalable Digital Libraries. In *4th Workshop on Very Large Digital Libraries (VLDL)*. Berlin, Germany: Springer-Verlag.

[175] Treinish, L. A. (1995). Solution Techniques for Data Management, the Visual Display and the Examination of Large Scientific Data Sets. In *Supercomputing '95 Course, Visualizing and Examining Large Scientific Data Sets: A Focus on the Physical and Natural Sciences*.

[176] Tridgell, A. (2009). *Trivial Database (TDB)*. The Samba Project. [Last accessed: November 2015].
`http://tdb.samba.org/`

[177] Tridgell, A., Sahlberg, R., Adam, M., *et al.* (2013). *Samba CTDB*. The Samba Project. [Last accessed: November 2015].
`https://ctdb.samba.org/`

[178] Unattributed (1983). DBC/1012 Data Base Computer Concepts and Facilities. Tech. rep., Teradata Corporation, Los Angeles. No. C02-001-00.

[179] Unattributed (1999). *Personal Video Recorder,"The Basics in 5 Short Tours"*. TiVo Central UK. [Last accessed: November 2015].
`www.tivocentral.co.uk/tivo/Chapter2.pdf`

[180] Unattributed (2006). *Sun HPC ClusterTools 6*. Oracle. [Last accessed: November 2015].
`http://docs.oracle.com/cd/E19061-01/hpc.cluster6/index.html`

[181] Unattributed (2007). *CiteSeerX Data*. The College of Information Sciences and Technology, The Pennsylvania State University. [Last accessed: November 2015].
`http://csxstatic.ist.psu.edu/about/data`

[182] Unattributed (2011). *HathiTrust Digital Library*. United States: University of Michigan. [Last accessed: November 2015].
http://www.hathitrust.org/

[183] Unattributed (2011). *MapR's Direct Access NFS vs. Hadoop FUSE*. MapR Technologies. [Last accessed: November 2015].
www.mapr.com/sites/default/files/mapr-nfs-techbrief.pdf

[184] Unattributed (2011). *Number of Words in the English Language:* 1,025,109.8. Global Language Monitor. [Last accessed: November 2015].
http://www.languagemonitor.com/new-words/number-of-words-in-the-english-language-1008879/

[185] Unattributed (2012). *EPrints: Flexible Repository Software*. University of Southampton, UK. [Last accessed: November 2015].
http://www.eprints.org/

[186] Unattributed (2012). *Petabox* 4. Internet Archive. [Last accessed: November 2015].
https://archive.org/web/petabox.php

[187] Unattributed (2014). *DSpace User Registry*. DuraSpace. [Last accessed: November 2015].
http://registry.duraspace.org/registry/dspace

[188] Unattributed (2014). *Frequently Asked Questions*. The Library of Congress. [Last accessed: November 2015].
http://www.loc.gov/webarchiving/faq.html

[189] Unattributed (2014). *Members*. International Internet Preservation Consortium. [Last accessed: November 2015].
http://netpreserve.org/about-us/members

[190] Unattributed (2014). *Social Media Analytics Installation and Configuration Guide 1.2.0*. IBM. [Last accessed: November 2015].
http://www-01.ibm.com/support/knowledgecenter/SSJHE9_1.2.0/com.ibm.swg.ba.cognos.sma.doc/welcome.html

[191] Unattributed (2015). *ITU-T H.222.0 ISO/IEC 13818-1: 2015 - Information technology – Generic coding of moving pictures and associated audio information – Part 1: Systems*. CH-1214 Vernier, Geneva, Switzerland: Internation Organization for Standardization / International Electrotechnical Commission.

[192] Venkatraman, S. S. (2004). Software as a Service. *Academy of Information and Management Sciences*, *8*(1), 97.

[193] Wactlar, H. D., Kanade, T., Smith, M. A., *et al.* (1996). Intelligent Access to Digital Video: Informedia Project. *Computer*, *29*(5), 46–52.

[194] Walker, D. (1992). Standards for Message Passing in a Distributed Memory Environment. Tech. rep., Technical Report TM-12147, Oak Ridge National Laboratory.

[195] Wang, J. Z., Li, J., Wiederholdy, G. (2000). SIMPLIcity: Semantics-Sensitive Integrated Matching for Picture libraries. In *Advances in Visual Information Systems*, pp. 360–371. Berlin: Springer-Verlag.

[196] Warschko, T. M., Blum, J. M., Tichy, W. F. (1998). ParaStation - Efficient Parallel Computing by Clustering Workstations Design and Evaluation. *Journal of Systems Architecture*, *44*(3), 241–260.

[197] Wenqing, W., Ling, C. (2010). Building the New-Generation China Academic Digital Library Information System (CADLIS): A Review and Prospectus. *D-Lib Magazine*, *16*.

[198] White, T. (2009). *Hadoop: The Definitive Guide: The Definitive Guide*. O'Reilly Media.

[199] Wilder, B. (2012). *Cloud Architecture Patterns: Using Microsoft Azure*. O'Reilly Media, Inc.

[200] Williams, J. (2014). *Strace Analyzer NG*. ClusterBuffer. [Last accessed: November 2015].
`https://clusterbuffer.wordpress.com/strace-analyzer/`

[201] Witten, I. H., Bainbridge, D., Nichols, D. M. (2009). *How to Build a Digital Library*. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 2nd edn.

[202] Witten, I. H., Bainbridge, D., Tansley, R., *et al.* (2005). StoneD: A Bridge Between Greenstone and DSpace. *D-Lib Magazine*, *11*(9), 2–19.

[203] Witten, I. H., Boddie, S. J., Bainbridge, D., *et al.* (2000). Greenstone: A Comprehensive Open-Source Digital Library Software System. In *Proceedings of the Fifth ACM Conference on Digital Libraries*, pp. 113–121. ACM.

[204] Witten, I. H., Frank, E. (2005). *Data Mining: Practical Machine Learning Tools and Techniques*. Morgan Kaufmann.

[205] Witten, I. H., Moffat, A., Bell, T. C. (1999). *Managing Gigabytes : Compressing and Indexing Documents and Images*. San Francisco, CA: Morgan Kaufmann, 2. edn.

[206] Yadava, H., Olson, M. (2007). *The Berkeley DB Book*, vol. 79. Berlin: Springer-Verlag.

[207] York, J., Arbor, A. (2009). This Library Never Forgets: Preservation, Cooperation, and the Making of HathiTrust Digital Library. *Archiving Conference*, *2009*(1), 5–10.

[208] Yu, W., Vetter, J., Canon, R. S., *et al.* (2007). Exploiting Lustre File Joining for Effective Collective I/O. In *In CCGRID 2007: Proceeding of the Seventh IEEE International Symposium on Cluster Computing and the Grid*, pp. 267–274. IEEE.

[209] Zaharia, M., Konwinski, A., Joseph, A. D., *et al.* (2008). Improving MapReduce Performance in Heterogeneous Environments. In *Proceedings of the 8th USENIX Symposium on Operating Systems Design and Implementation*, vol. 8, pp. 29–42. USENIX Association.

[210] Zambrano-Bigiarini, M. (2012). *HydroGOF: Goodness-of-fit functions for comparison of simulated and observed hydrological time series*. RForge. [Last accessed: November 2015].
`http://www.rforge.net/hydroGOF/`

Appendices

Appendix A

Exemplar Digital Collection Processing Measurements

This appendix presents Table A.1, a selection of exemplar collection imports detailing the metadata extraction/processing configured and the resulting percentage of time spent doing file I/O and megabytes I/O throughput per second. Drawing upon existing Linux profiling approaches [109], this percentage was determined by recording timing information on the kernel level system calls during repeated imports (using the Linux tool *strace*) and then summing up all the time spent in known I/O functions. Care was taken to account interrupted/suspended calls and system signals. While recording strace logs incurs some extra I/O costs in and of itself, this cost was mitigated by the writing these logs into RAM rather than disk.

The kernel level system calls associated with I/O were: access, chmod, close, creat, fclose, fcntl, fgetpos, flock, fseek, fsetpos, fstat, fsync, ftell, getdents, ioctl, llseek, lockf, lseek, lseek64, mkdir, open, read, readdir, rename, rewind, rewinddir, scandir, stat, stat64, telldir, unlink, write.

Table A.1: Exemplar digital library processing measurements

Media	Processing load	I/O%	MBIOperS	Description of configured processing
Text	Minimal	0.4862	0.0226	Add filename, filesize, mime-type, and text content as metadata
	Typical	0.0989	0.0006	*plus* calculate text metrics (Flesh-Kincaid) and encrypt metadata (Blowfish)
	Intensive	0.0903	0.0002	*plus* use data mining to extract keywords and phrases
Image	Minimal	0.5651	0.5359	Copy images into place and add filename, file size, and mime-type as metadata
	Typical	0.1673	0.0133	*plus* generate thumbnail preview and copy into place
	Intensive	0.1145	0.0028	*plus* use machine learning image processing to generate metrics (SIFT)
Audio	Minimal	0.9197	89.0661	Copy track into place and extract simple metadata (ID3 tags)
	Typical	0.0658	0.0339	*plus* convert track into streamable format (FLV)
	Intensive	0.0135	0.0004	*plus* use data mining to extract music information retrieval features (FFTW)
Video	Minimal	0.5882	15.8376	Copy video into place and extract simple codec metadata
	Typical	0.0083	0.0048	*plus* convert video into streamable format (MP4)
	Intensive	0.0062	0.0010	*plus* use machine learning to capture high-value keyframes (Hive2)

Appendix B

Selected Parallel Processing Visualisations

This appendix provides a selection of the Gantt-like charts used to visualise digital library import processes utilising parallel processing. There are three example charts included in this appendix:

- Figures B.1 through B.3 chart a typical video document import, utilising Hadoop with the HDThriftFS interface to HDFS, configured to perform intensive metadata processing.
- Figures B.4 through B.6 build on the first example by showing how staggering the launching time of worker threads has an effect on the initial file transfer time (which would otherwise encounter higher contention).
- Finally, Figures B.7 through B.9 illustrate a problematic import of a video collection using the HDFS-NFS-Proxy driver to access files from the Hadoop Distributed File System. Premature truncations of video file processing are evident.

The formatting for these charts was explained in Section 3.4.1, but to recap:

- The top of each chart includes a detailed summary of important metrics relevant to the import,
- With the chart, each thread—be it the master or a worker, and running on either a processor core or compute node in a cluster—is represented with a summary task bar indicating its unique identifier, start time relative to master thread, and total elapsed time,

- Each processing task is labelled with the document/file being processed,
- The vertical cascade shows the ordering of tasks over time,
- Shading of the task indicates the time spent being processed by a CPU in grey, while a white background indicates file input/output,
- Specific to cluster hardware configurations, a solid task border indicates that the process was performed on the same compute node as the data resided, while a dotted line and the suffix "[NL]" indicates data had to be copied to the compute node, and finally
- In the case where a file was not completely processed, a suffix of the form "[Incomplete! $XXX\%$]" is appended stating the percentage that was processed.

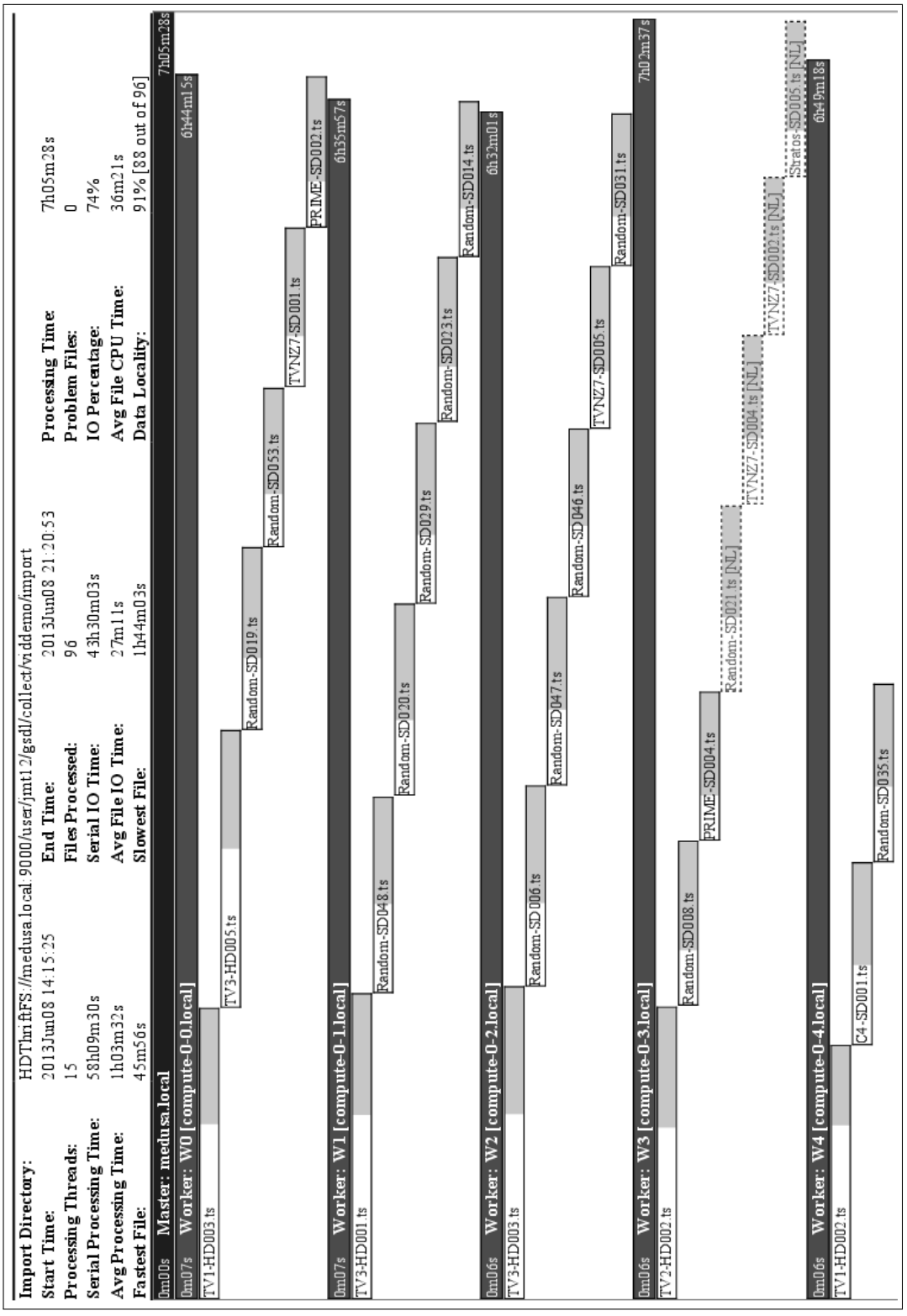


Figure B.1: Gantt chart of parallel import of video collection utilising Hadoop with HDThriftFS - Part 1 of 3

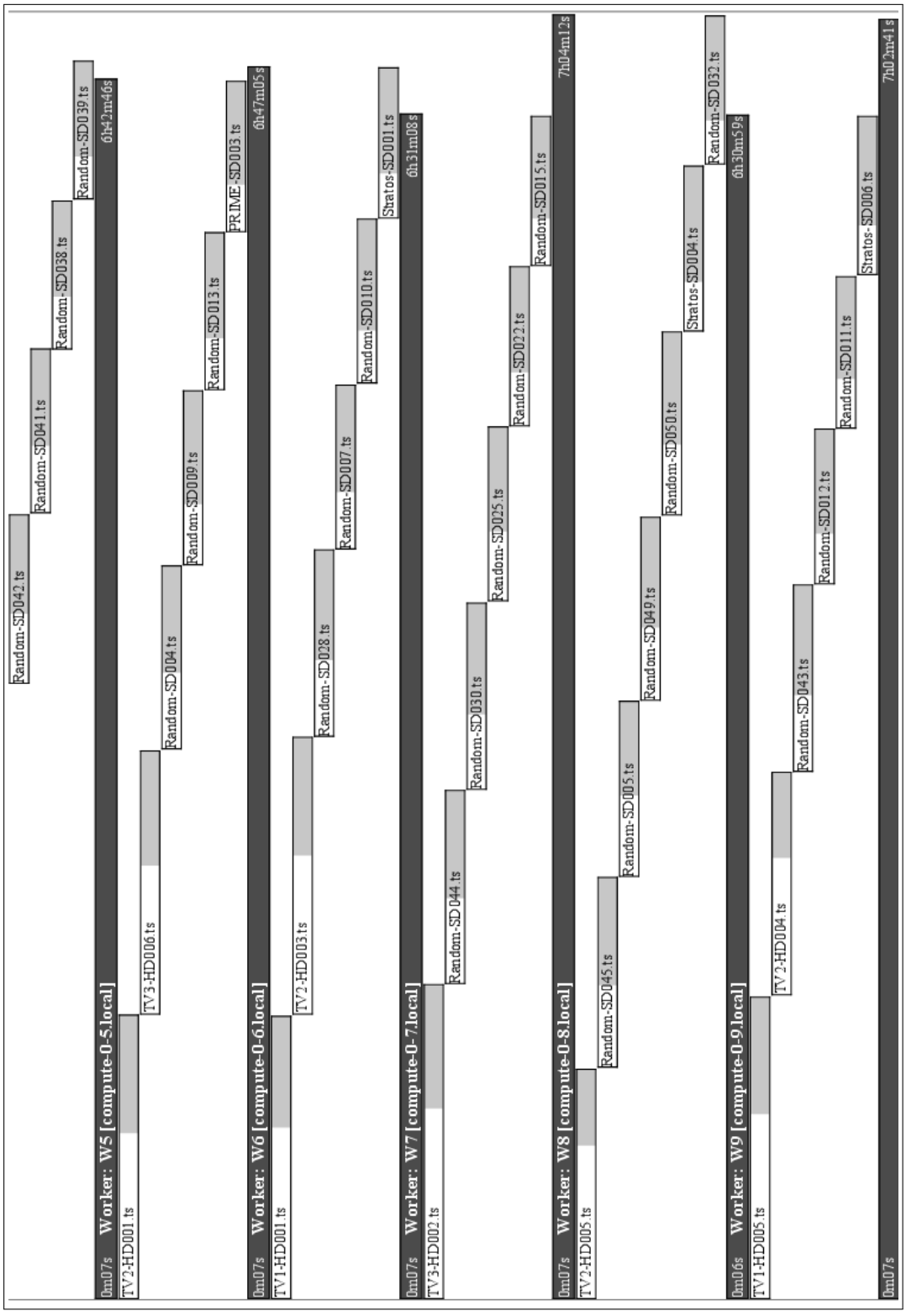


Figure B.2: Gantt chart of parallel import of video collection utilising Hadoop with HDThriftFS - Part 2 of 3

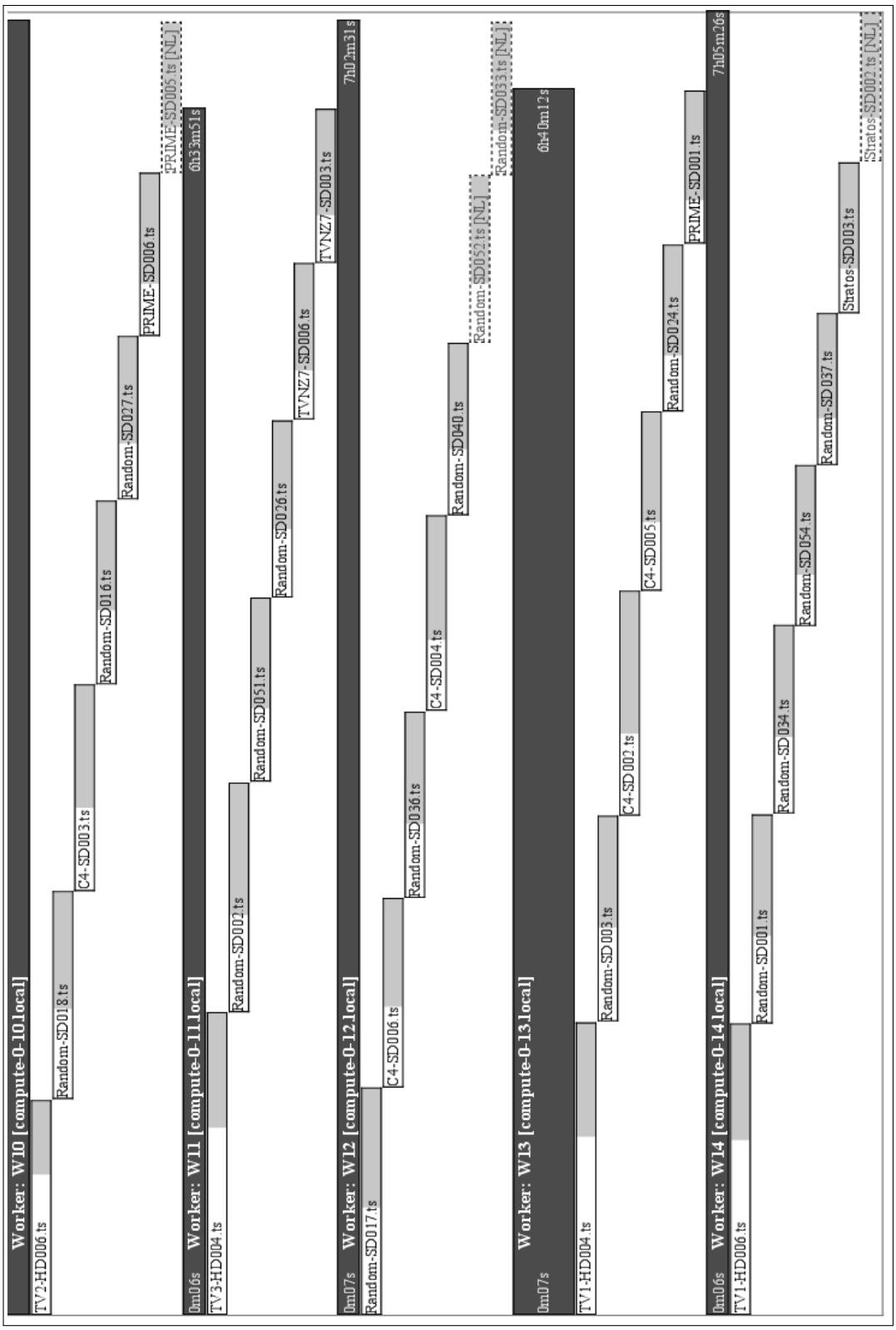


Figure B.3: Gantt chart of parallel import of video collection utilising Hadoop with HDThriftFS - Part 3 of 3

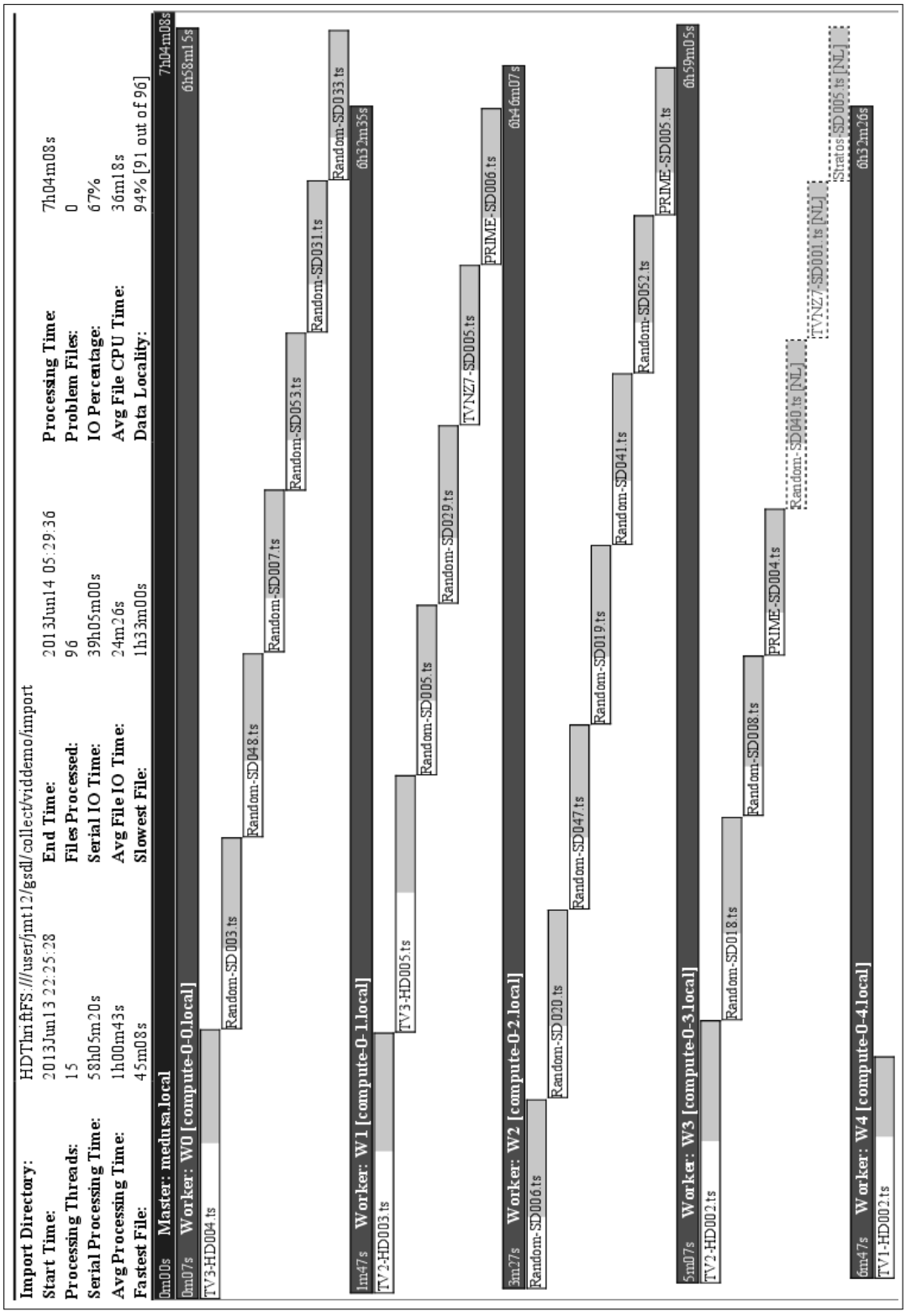


Figure B.4: Gantt chart of parallel import of video collection utilising Hadoop with HDThriftFS and staggered thread starting - Part 1 of 3

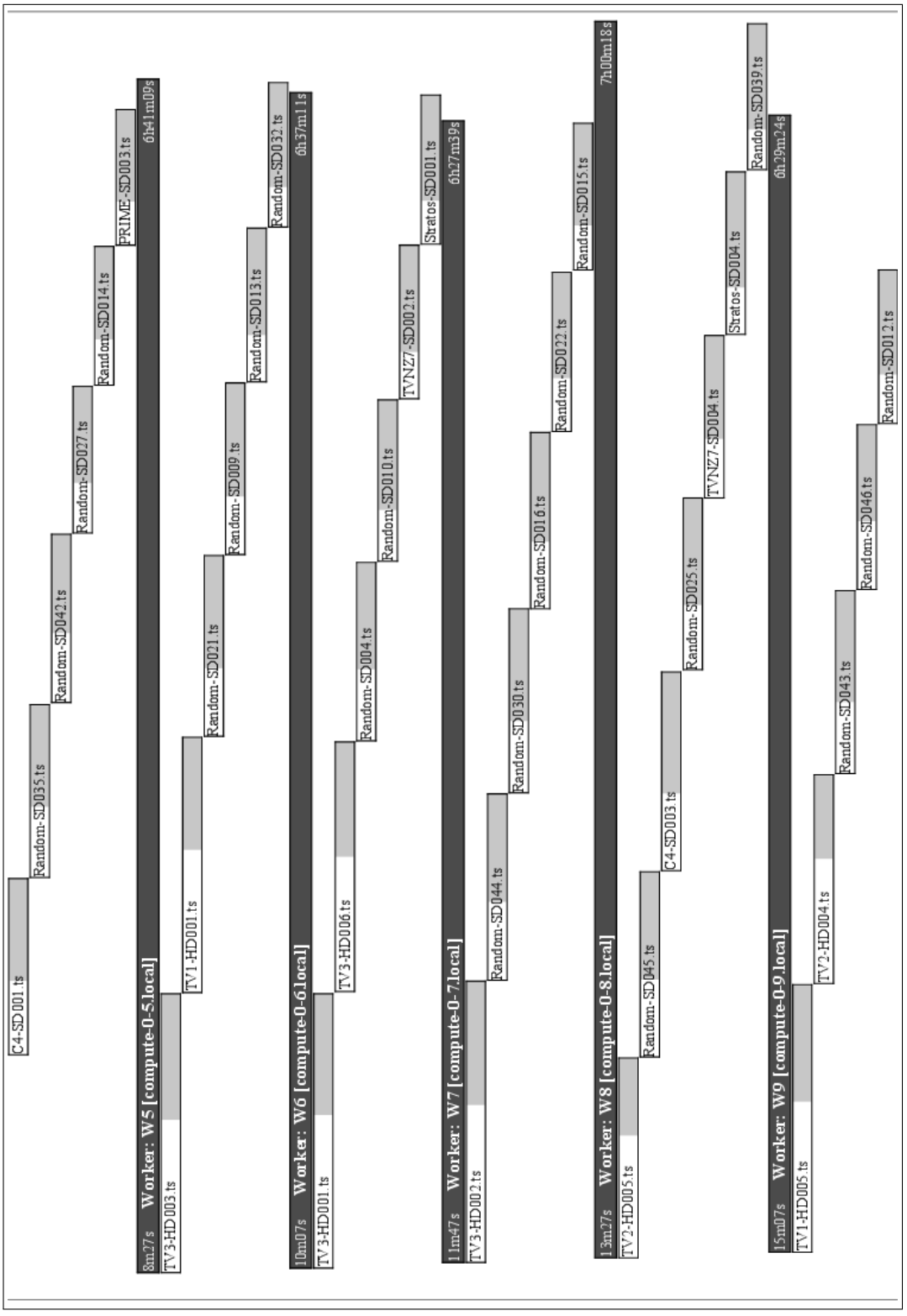


Figure B.5: Gantt chart of parallel import of video collection utilising Hadoop with HDThriftFS and staggered thread starting - Part 2 of 3

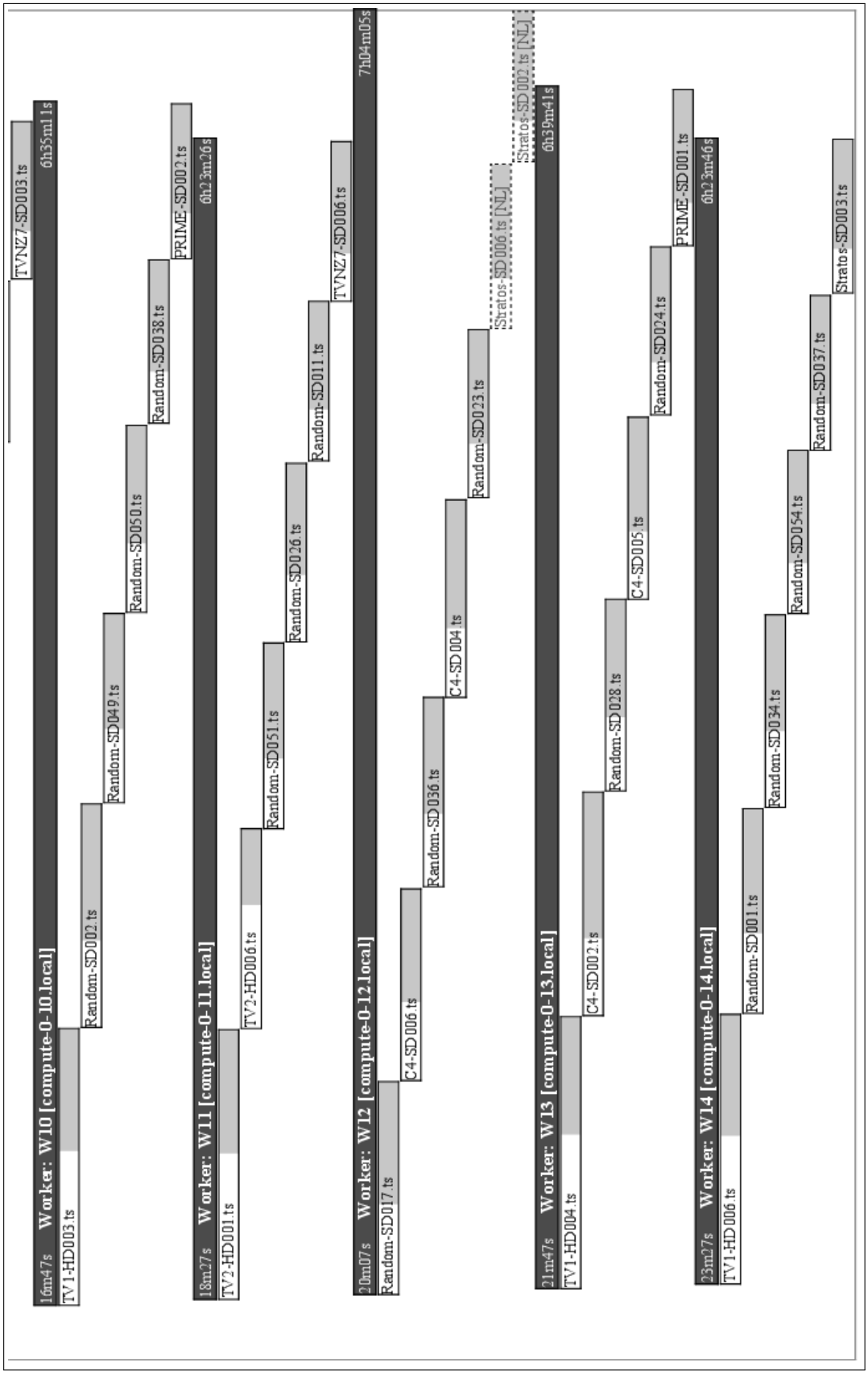


Figure B.6: Gantt chart of parallel import of video collection utilising Hadoop with HDThriftFS and staggered thread starting - Part 3 of 3

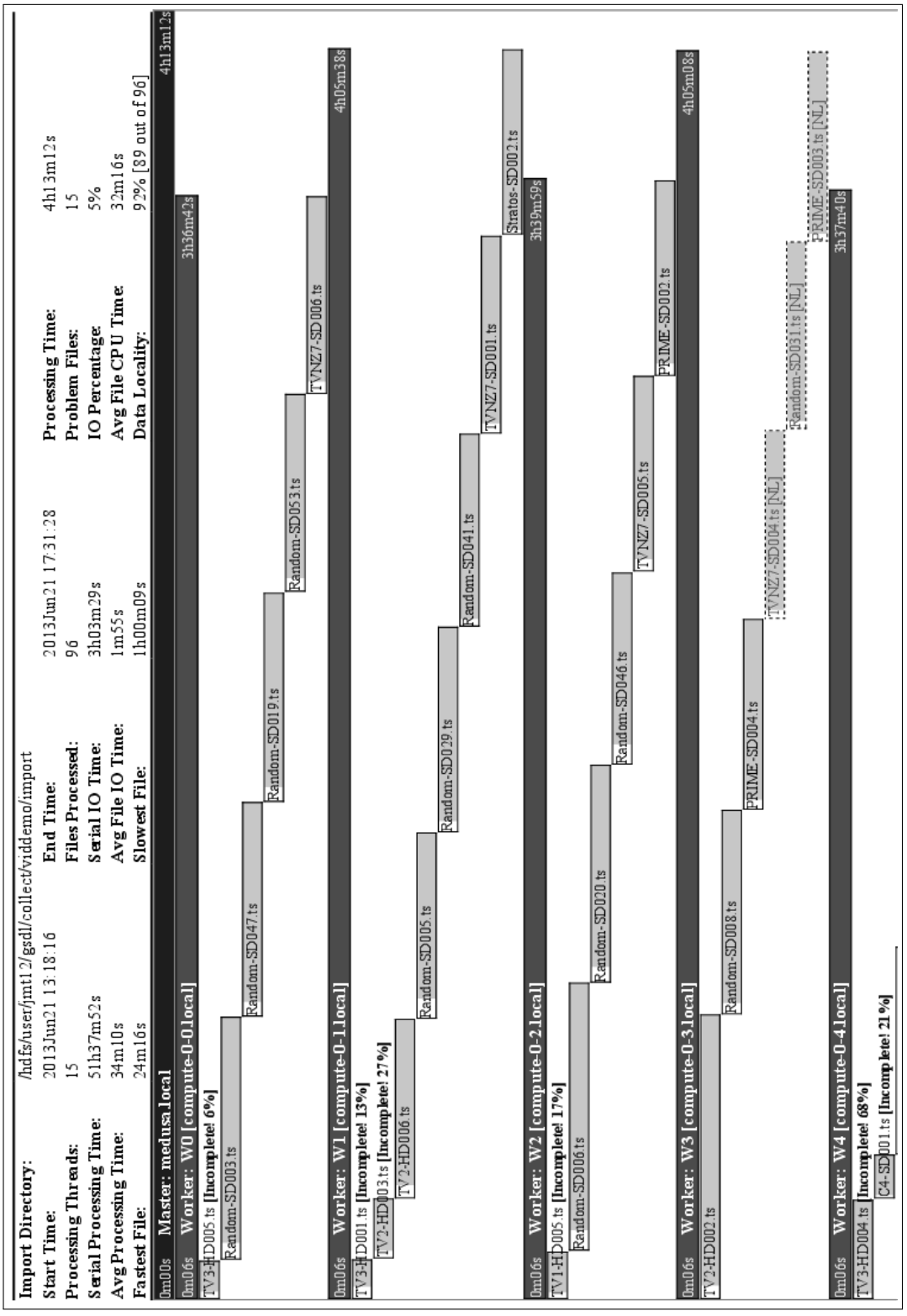


Figure B.7: Gantt chart of problematic parallel import of video collection utilising Hadoop with HDFS-NFS-Proxy - Part 1 of 3

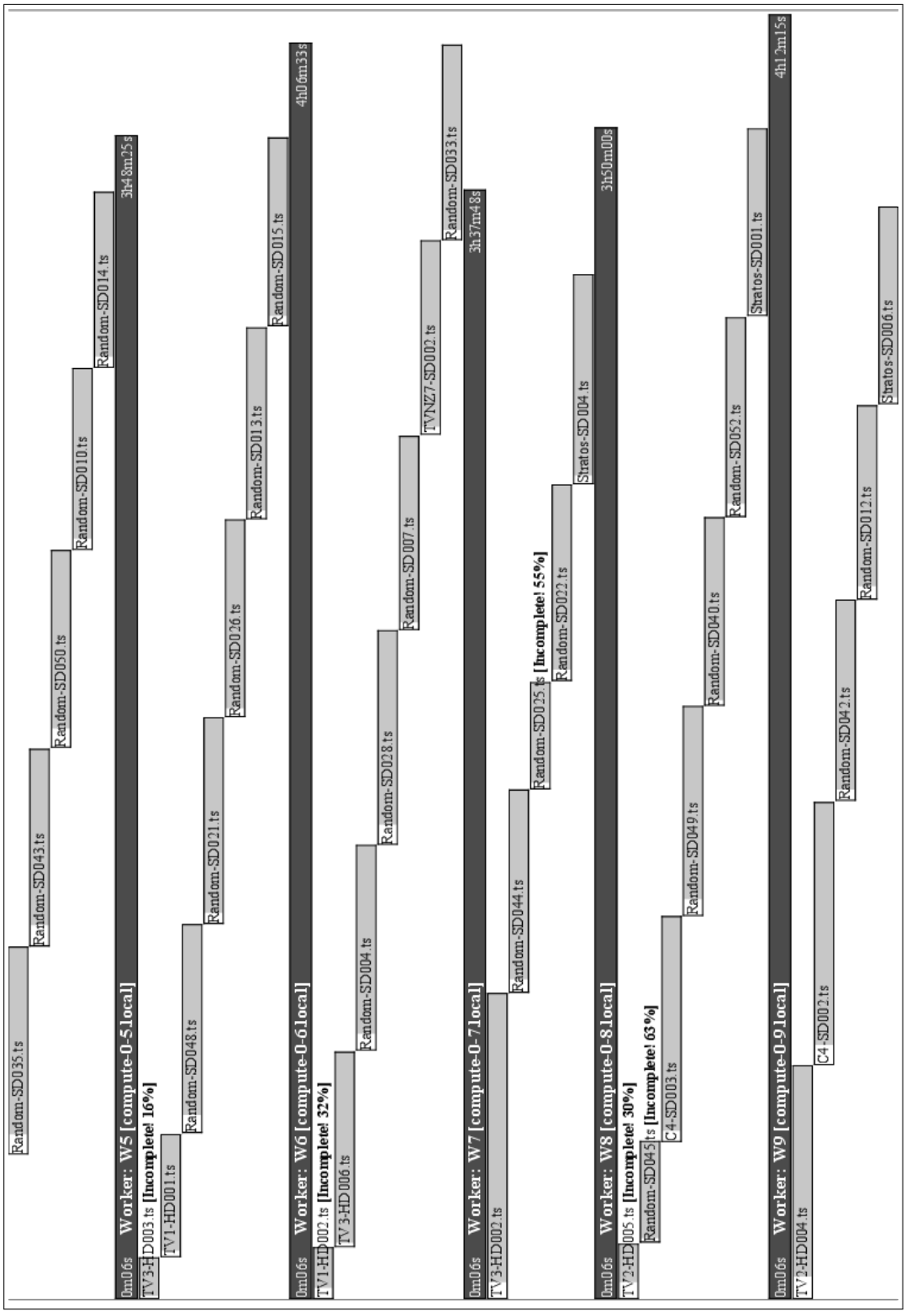


Figure B.8: Gantt chart of problematic parallel import of video collection utilising Hadoop with HDFS-NFS-Proxy - Part 2 of 3

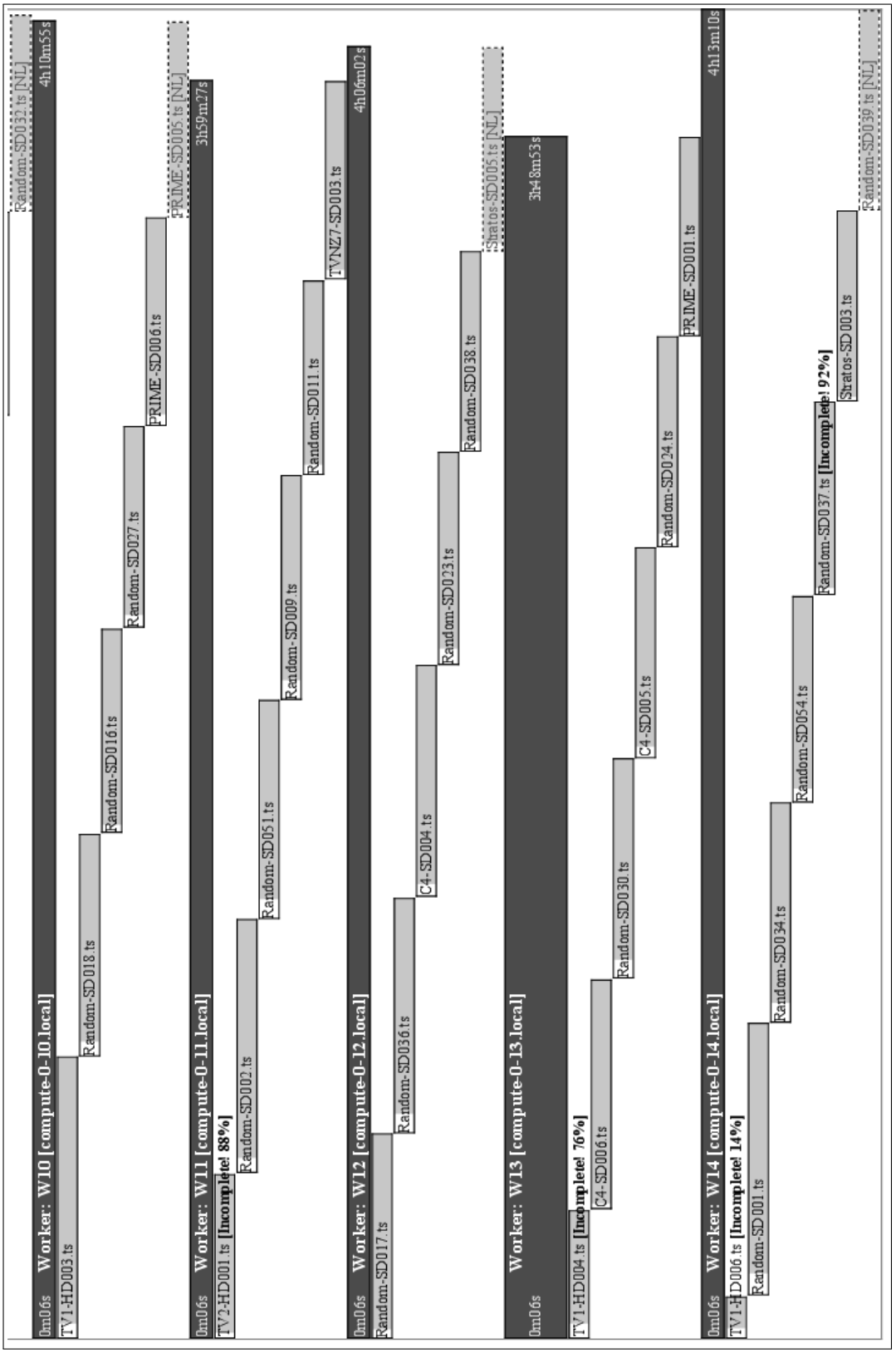


Figure B.9: Gantt chart of problematic parallel import of video collection utilising Hadoop with HDFS-NFS-Proxy - Part 3 of 3

Appendix C

Software

This appendix provides a listing of important software used or investigated during the course of this research. We include the URL for accessing downloads as available at October 2014. Descriptions are retrieved from Wikipedia.

DSpace

`http://www.dspace.org/`

DSpace is an open source repository software package typically used for creating open access repositories for scholarly and/or published digital content. While DSpace shares some feature overlap with content management systems and document management systems, the DSpace repository software serves a specific need as a digital archives system, focused on the long-term storage, access and preservation of digital content.

FFmpeg

`http://ffmpeg.org/`

FFmpeg is a free software project that produces libraries and programs for handling multimedia data. FFmpeg includes libavcodec, an audio/video codec library used by several other projects, libavformat, an audio/video container mux and demux library, and the ffmpeg command line program for transcoding multimedia files. FFmpeg is published under the GNU Lesser General Public License 2.1+ or GNU General Public License 2+ (depending on which options are enabled).

GDBM

`http://www.gnu.org.ua/software/gdbm/`

A free/libre version of DBM written by Philip A. Nelson for the GNU project. It added support for arbitrary-length data in the database

whereas previously all data had a fixed maximum length. The last version was released on 25 Dec 2013.

Greenstone

`http://www.greenstone.org/download`

Greenstone is a suite of software tools for building and distributing digital library collections on the Internet or CD-ROM. It is open-source, multilingual software, issued under the terms of the GNU General Public License. Greenstone is produced by the New Zealand Digital Library Project at the University of Waikato, and has been developed and distributed in cooperation with UNESCO and the Human Info NGO in Belgium.

Greenstone Extension: Parallel Building

`http://trac.greenstone.org/browser/gs2-extensions/parallel-building/trunk/src`

An extension providing parallel processing support to Greenstone. Requires a framework such as Open MPI or Hadoop be already installed. This extension is under active development.

Greenstone Extension: TDB Edit

`http://trac.greenstone.org/browser/gs2-extensions/tdb-edit/trunk/src`

An extension allowing for the use of Trivial DB as Greenstone's backend database. Provides a executable that offers the same functionality as GDBM, while providing additional features such as multiple readers/writers (necessary for parallel processing).

Greenstone Extension: Video and Audio

`http://trac.greenstone.org/browser/gs2-extensions/video-and-audio`

An extension for Greenstone providing advanced support for advanced audio and video collections. Options include version conversion filters when importing documents, and the extraction of audio and video features for use in non-textbased information retrieval interfaces.

Hadoop

`http://hadoop.apache.org/`

Apache Hadoop is an open-source software framework for storage and large-scale processing of data-sets on clusters of commodity hardware. Hadoop is an Apache top-level project being built and used by a global community of contributors and users. It is licensed under the Apache License 2.0.

HandBrake

`http://handbrake.fr/`

HandBrake is a free and open-source multithreaded transcoding app, originally developed by Eric "titer" Petit in 2003 to make ripping a film from a DVD to a data storage device easier. Since then, it has undergone many changes and revisions. Handbrake is available for Windows, OS X and Ubuntu from its official website, although it is possible to compile it for Debian, Linux Mint, Fedora, CentOS or RHEL. HandBrake uses third-party libraries such as x264, Libav, and FAAC, the latter of which is slated for removal due to licensing issues.

HDFS-NFS-Proxy

`https://github.com/cloudera/hdfs-nfs-proxy/wiki/`

Distributed as part of the Cloudera project, the NFSv3 proxy allows a client to mount HDFS as part of the client's local file system. The gateway machine can be any host in the cluster, including the NameNode, a DataNode, or any HDFS client. The client can be any NFSv3-client-compatible machine.

Hive: Video shot boundary detection

`-not available online-`

A prototype software library developed at the Imperial College, London, at part of an application that applies CBIR techniques to video media [85]. Specifically, the library uses machine learning techniques to locate the boundaries between shots and then ranks the next frame in the video by information value. A user can then ask for the top n frames to use as key frames indicating the content of the video, or in interaction mechanisms like scene selection.

Lucene

`http://lucene.apache.org/`

Apache Lucene is a free open source information retrieval software library, originally written in Java by Doug Cutting. It is supported by the Apache Software Foundation and is released under the Apache Software License.

MapR

`http://www.mapr.com/`

MapR is a San Jose, California-based enterprise software company that develops and sells Apache Hadoop-derived software. The company contributes to Apache Hadoop projects like HBase, Pig (programming language), Apache Hive, and Apache ZooKeeper. MapR's Apache Hadoop

distribution claims to provide full data protection, no single points of failure, improved performance, and dramatic ease of use advantages. MapR provides three versions of their product known as M3, M5 and M7. M3 is a free version of the M5 product with degraded availability features. M7 is like M5, but adds a purpose built rewrite of HBase that implements the HBase API directly in the file-system layer.

MediaInfo

`http://mediaarea.net/mediainfo/`

MediaInfo is a free and open-source program that displays technical information about media files, as well as tag information for many audio and video files. It is used in many programs such as XMedia Recode, MediaCoder, eMule, and K-Lite Codec Pack. It can be easily integrated into any program using a supplied MediaInfo.dll. MediaInfo supports popular video formats (e.g. AVI, WMV, QuickTime, Real, DivX, XviD) as well as lesser known or emerging formats such as MKV including WebM. In 2012 MediaInfo 0.7.57 was also distributed in the PortableApps format.

NYTProf

`http://search.cpan.org/~timb/Devel-NYTProf-5.06/lib/Devel/NYTProf.pm`

The purpose of this tool is to allow developers to easily profile Perl code line-by-line with minimal computational overhead and highly visual output. With only one additional command, developers can generate robust color-coded HTML reports that include some useful statistics about their Perl program.

Open MPI

`http://www.open-mpi.org/`

Open MPI is a Message Passing Interface (MPI) library project combining technologies and resources from several other projects (FT-MPI, LA-MPI, LAM/MPI, and PACX-MPI). It is used by many TOP500 supercomputers including Roadrunner, which was the world's fastest supercomputer from June 2008 to November 2009, and the K computer, the fastest supercomputer from June 2011 to June 2012.

R

`http://www.r-project.org/`

R is a free software programming language and software environment for statistical computing and graphics. The R language is widely used among statisticians and data miners for developing statistical software and data analysis. Polls and surveys of data miners are showing R's

popularity has increased substantially in recent years.

ReplayMe!

`-not available online-`

An extensively customised collection for Greenstone, providing in a personalised television/movie library experience similar to the commercial product TiVo. Implements importing plugins for raw TS streams, and integrates several novel user interfaces to improve usability.

strace

`http://sourceforge.net/projects/strace/`

strace is a diagnostic, debugging and instructional userspace utility for Linux. strace is used to monitor interactions between processes and the Linux kernel, which include system calls, signal deliveries, and changes of process state. The operation of strace is made possible by the kernel feature known as *ptrace*.

taskset

`http://freecode.com/projects/util-linux/`

taskset is used to set or retrieve the CPU affinity of a running process given its PID or to launch a new COMMAND with a given CPU affinity. CPU affinity is a scheduler property that "bonds" a process to a given set of CPUs on the system.

Thrift

`http://thrift.apache.org/`

The Apache Thrift software framework, for scalable cross-language services development, combines a software stack with a code generation engine to build services that work efficiently and seamlessly between C++, Java, Python, PHP, Ruby, Erlang, Perl, Haskell, C#, Cocoa, JavaScript, Node.js, Smalltalk, OCaml and Delphi and other languages.

Trivial DB

`http://tdb.samba.org/`

This is a simple database API. It was inspired by the realisation that in Samba we have several ad hoc bits of code that essentially implement small databases for sharing structures between components. The interface is based on GDBM but extends it to allow multiple simultaneous writers to a database.

Terrier IR Platform

`http://www.terrier.org/`

Terrier is a modular open source software for the rapid development of

large-scale Information Retrieval applications. Terrier was developed by members of the Information Retrieval Research Group, Department of Computing Science, at the University of Glasgow. Written in Java, a core version of Terrier is available as open source software under the Mozilla Public License (MPL), with the aim to facilitate experimentation and research in the wider information retrieval community.

Veridian

`http://www.veridiansoftware.com/`

Large-scale digital library software developed commercially by DL Consulting Ltd. See Section 2.2.6 for details. Early development in this software for the PapersPast project inspired the research presented in this thesis.

Weka

`http://www.cs.waikato.ac.nz/~ml/weka/`

Weka (Waikato Environment for Knowledge Analysis) is a popular suite of machine learning software written in Java, developed at the University of Waikato, New Zealand. The Weka (pronounced Weh-Kuh) workbench contains a collection of visualization tools and algorithms for data analysis and predictive modeling, together with graphical user interfaces for easy access to this functionality. Weka is free software available under the GNU General Public License.