Towards Very Large Scale Digital Library Building in Greenstone using Parallel Processing

John Thompson, David Bainbridge, and Hussein Suleman

Department of Computer Science, University of Waikato, Hamilton, New Zealand
jmt12@students.waikato.ac.nz, davidb@cs.waikato.ac.nz
http://cs.waikato.ac.nz/
Department of Computer Science, University of Cape Town, Cape Town, South Africa
hussein@cs.uct.ac.za
http://www.cs.uct.ac.za/

Abstract.
As very large digital library collections become more commonplace, software tools must adapt appropriately. This paper reports on an evolution of the Greenstone Digital Library software to support parallel processing during the collection building phase. A series of experiments were conducted to first establish a basic speed-up factor, and then deconstruct the parallelisation process to understand the execution profile of the application. Several bottlenecks were identified and resolved to further improve the performance. The adaptation of Greenstone confirms that the build phase is indeed a suitable candidate for parallelisation; and suggests that parallelisation of processing is a new avenue for exploration in emerging digital library architectures.

Keywords: Greenstone, VLDL, Parallel Processing, Open MPI

1 Introduction

The last few decades have seen an explosion in the amount of digital information being collected, and the issue at hand is matching this with the development of systems for presenting and making this information accessible. Key amongst these systems is digital library software, which provides a structured, browseable, 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 (www.hathitrust.org) has, at the time of writing, full text indices for approximately eight million documents.

The creation and practical use of VLDLs has also raised many new issues, such as those discussed during the international VLDL workshop series (e.g., Mangi et al. [5]), particularly related to size (in aspects such as disk space, number of documents, and number of unique terms or words), sustainability, and interoperability. But technology has also advanced and, with multi-processor/hyper-threading now the norm in consumer computers, explicitly developing techniques for digital library software that exploits parallel processing

capabilities looks promising as a solution to some of these scale concerns. For example, Stanfill [7] proposed a number of ways to leverage the power of parallel processing applicable to the information retrieval component of a digital library system, and acting on indices in the order of terabytes.

Meanwhile, Greenstone [8] is a well established, open-source, digital library tool, but has traditionally had a reputation of only being useful for small, lightweight, or demonstration collections. However recent scalability testing results [2] and large scale digital collections such as the National Library of New Zealand's PapersPast digital library [1] show that Greenstone is capable of serving collections numbering well into the millions of documents.

It is perhaps no surprise that the initial ingest of such large scale collections (using the software without modification) is time-consuming. Real-world VLDLs may take in excess of several days of processing to create. In our scalability testing results we found that Greenstone's importing time is roughly linear with the number of documents. This paper details research in an attempt to improve this performance by applying parallel processing.

The structure of the paper is as follows. We start by outlining the scope of the work reported here, which concentrates on the ingest phase of collection building, and then briefly mention related digital library projects. We then present details of the software changes we made to enable Greenstone to automatically process documents in parallel. Following this, we present and discuss the results of a series of tests that helps to establish where time-consuming steps occur within this process (leading to further adjustments in the software implementation), before concluding with a discussion of the implications of these findings on building large scale collections and future work.

2 Scope

In the work reported here, we focus on the collection indexing phase (*build-time*) of Greenstone—where documents are transformed into a homogeneous format during an import phase and then indexed during a build phase. We leave for future research the opportunity to apply parallel processing to the *run-time* of Greenstone—where the user interacts with the digital library, via a Web browser or other presentation tool, in potentially processor-heavy fashions (for instance when interacting with video or other multimedia).

Of the two build-time tasks mentioned, this paper will specifically explore our work on the process of importing disparate documents into a standardized XML format. We started with importing because:

- it offers clear opportunities for parallelism (given that by-and-large the transform of each document is atomic);
- there is a greater possibility of significant savings in processing time as importing is (anecdotally) slower than indexing (especially for processor intensive documents like media and PDFs);

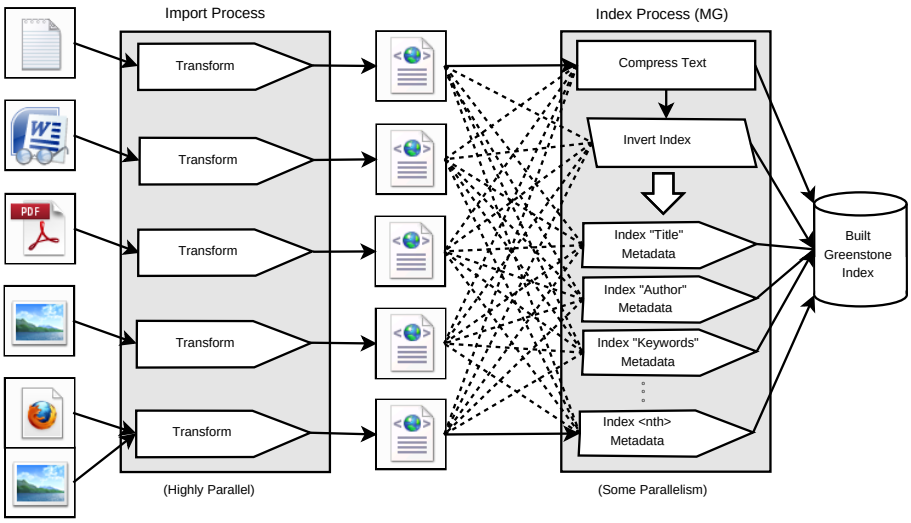

Fig. 1. Potential for parallelization in Greenstone's import and build processes.

– it offers a wide window for experimentation (such as altering the number and processing complexity in each group of documents being ingested by each process).

We also believe that importing offers the more novel problem. The accompanying issue of large scale indexing is already being explored (for instance [3]). Furthermore we will need to ascertain whether Greenstone's indexing algorithms and architecture will actually benefit from using multiple processors as suggested by existing literature [6].

3 Related Work

As an example of another large scale digital library project, the HathiTrust Digital Library estimated that an unoptimized re-import of their seven million plus volumes might take ten days.[1] However, by utilizing parallel architecture and optimizing the underlying indexing system's configuration (Solr[2]) in terms of IO interaction they were able to reduce this to around ten days.

It is worth mentioning that, unlike many of its contemporary digital respository projects (such as DSpace and Islandora) that use a relational database for their datastore (and hence only have full-text indexing if that database supports it), Greenstone makes use of a flat database for storing metadata with a separate system for indexing (typically MG, MGPP or Lucene). While this structure

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

[2] `http://lucene.apache.org/solr/`

proved an early advantage—before relational databases began supporting full text indexes—it now presents unique challenges to large scale collection building.

4 Method

Figure 1 gives a schematic overview of Greenstone's ingest process—which is divided into two completely separate phases, importing and indexing—and helps to illustrate where the potential lies in the system for supporting parallelism. During importing, each document is handled independently, lending itself well to parallelism. The only real complication to deal with here is metadata that has been defined higher up in the file-system structure and set to inherit to child folders and files—for instance all documents in the given folder, and sub-folders gets assigned the Dublin Core Subject metadata value "Art History." During indexing, the main opportunity for parallelism is the multiple passes that the indexer makes to form the indexing files, as some of these passes—depending on how a collection has been configures—are independent of one another.

More specifically, in order to explore parallel processing opportunities, the decision was made to integrate an existing Message Passing Interface (MPI) library called Open MPI [5] into Greenstone. This mature open source project has been built with high performance in mind and supports a plethora of operating systems, as well as cluster and multi-core configurations. While MPI is arguably a low-level interface, it was adequate for this project and the MPI primitives can easily be translated to other libraries. The Greenstone import process was extended to use Open MPI's architecture to allow for a single controller import process to manage a number of worker import processes, each of which would ingest one batch of files using the serial Greenstone import process. The parallel import process allowed configuration in terms of number of worker threads (parallel jobs) and batch size for files to be processed. Greenstone already had mechanisms for importing the collection's documents in batches (controlled by XML manifest files). It should be noted that a typical, non-manifest, Greenstone import actually has to do more processing; for example, it performs a non-trivial pre-scan of all documents to be imported to locate any applicable metadata files. To avoid these algormithm differences we passed manifest files both to the parallel import processes and the 'control tests' of non-parallel imports processes.

While the majority of the Greenstone importing process (such as the "plugins" used to read the disparate document formats and the "plugouts" used to write the standardized XML files) was found to work fine when applied to multiple processes, there were other aspects that needed customization to support parallel processing. Key amongst these was the interaction between Greenstone and the GDBM database[3] it used to store metadata and processing information. GDBM only allows a single process to write to it. This had not previously been an issue with normal serial processing, but now required the addition of locking/synchronization calls around Greenstone's critical sections that made calls

[3] `http://www.gnu.org/software/gdbm/`

to GDBM in order to prevent errors in the database and abnormal program terminations. Once this work was completed, Greenstone's parallel importing was brought to a point where we could begin performance tests.

The initial results from early testing were somewhat disappointing (as detailed in Results) with processing time only being halved despite utilizing multiple cores. At this point we used profiling tools[4] to analyze the import process to determine what factors were limiting the performance of multiple processors. This quickly revealed a number of bottlenecks, many of which were subsequently mitigated by careful selection of built-in Greenstone configuration options and the tests rerun. But 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 utilization, occurs at P or $P + 1$ (where P is number of computer processors) [4]. Under the assumption that importing might be exhibiting IO bounds rather than processing ones, further tests were designed to determine if IO was the remaining limiting factor in parallel importing.

To test each of the configurations listed below we ran a series of import processes over a collection of 10,000 documents. Each import in the series varied the numbers of parallel worker threads and batch sizes in order to measure the influence of those factors. The documents—plain TXT files—contained carefully generated lorem ipsum[5] text designed to approximate real-world content. The testing was carried out on a modern PC (Intel Core i7) with eight cores (hyper-threaded from four physical cores), 4GB RAM and running CentOS 5.5. The result for each configuration, and for each combination of number of threads and batch size therein, captured the real, user and system time of importing averaged over multiple executions.

The output materials from applicable configurations (explained in the next section) were compared in order to ensure that neither order of processing nor synchronous processing altered or damaged the output (in respect to a standard Greenstone import process). Aside from acceptable differences in timestamps the outputs were the same. The outputs (if any) for the configurations that altered IO were not compared as we already knew their content was compromised by the test.

4.1 Configurations

The following is a list of each of the limiting factors located by profiling and the modifications made to mitigate or remove them:

Standard. The first series of tests were run on an unaltered Greenstone configuration and made use of standard input plugins and plugouts.

Set ID. During import Greenstone uses a hashing process to generate unique identifiers for documents. This is not just processor intensive, but also IO

[4] `http://code.google.com/p/perl-devel-nytprof/`

[5] `http://www.lipsum.com/`

intensive as it would re-read the source file contents (even for large files like videos) to hash upon. To avoid this cost the configuration was changed to assign the filename as the unique identifier instead. This setup was used for the second series of tests.

Set encoding. In order to convert all documents to a standard encoding (UTF8), Greenstone must first determine the input encoding of each document. Again this was found to be IO dependent as the encoding analysis involves reading in a block of characters of the file and determining the codepage of each one. While not always possible in real world situations, in this experiment we were able to manually assign the encoding of the input documents as we had generated them as ASCII. A configuration with both assigned identifiers and assigned encodings was used for the third series of tests.

No input IO. With the configuration optimized with the above points, we started to investigate whether the act of reading and writing the documents was the limiting factor. We started by replacing the standard input plugin for TXT files with one that returned a hardcoded lorem ipsum document. This document was representative of an average document in the collection and the plugin allowed us to decrease input IO activity during the fourth series of tests. It should be noted that while acceptable for an experiment on importing, replicating the same content multiple times would have had serious consequences if indexing were to be applied.

No output IO. For the fifth series of tests we decided to minimize both input and output IO, and so the output 'plugouts' were replaced with versions that did no file writing (neither the homogeneous XML files nor the GDBM database used for processing information) at all. This decreases the IO activity to nearly nothing, but obviously does not reflect real-world or any practical usage and is instead meant to provide a baseline indicating the fastest possible processing times.

TDB. Finally, during the "no IO" test it was noticed that the interaction with the GDBM database was expensive compared to the output of the XML files. We include below one more test result that shows the 'bleeding-edge' results of an attempt to replace the GDBM database with TDB[6] which supports parallel readers and writers. This sixth series of tests was carried out with both plugins and plugouts restored to default but retained the assigned identifier and encoding configuration.

5 Results

Figure 2 shows a comparison of the average import process times (importing 10,000 documents) of the first five series of tests as the number of worker threads increase. To recap, the series "standard" represents a default Greenstone import configuration, "set ID" and "set encoding" series represent configurations with those metadata values specified (note that the latter has both id and encoding

[6] Short for Trivial Database, developed by the Samba project. See http://tdb.samba.org for more details.

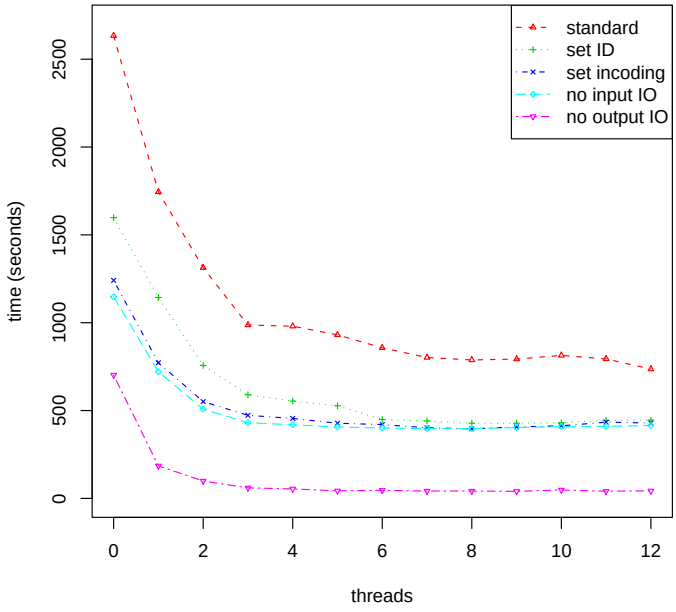

Fig. 2. Comparison of the import process time of five configurations (excluding TDB)

assigned), and "no input IO" and "no IO" represent configurations where IO have been reduced. When the number of worker threads is zero, a serial Greenstone import is used. When processing in parallel a fixed batch size of 100 was used.

These results show that, in general, increasing the number of threads running in parallel continues to decrease total processing time. Two results that surprised us were that the performance when number of threads equals one actually improved (we predicted performance would drop as a single thread should have all the overhead of parallel processing management with none of the benefit) and that the benefit of adding more threads quickly dropped after threads equal to three (we'd expected performance to keep improving - this is the result we earlier described as disappointing, and hinted that something other than processing capability was hindering progress). While setting simple configuration changes to avoid optional processing and IO causes a slight improvement in performance, the last series clearly highlights that some part of the output IO is dramatically limiting the effectiveness of parallel processing in Greenstone.

Figure 3 illustrates the effects of variation in batch sizes and number of threads on the performance of the import process. The test results shown here made use of the configuration where both unique identifier and encoding are assigned, but are representative, in terms of general shape, for all of the config-

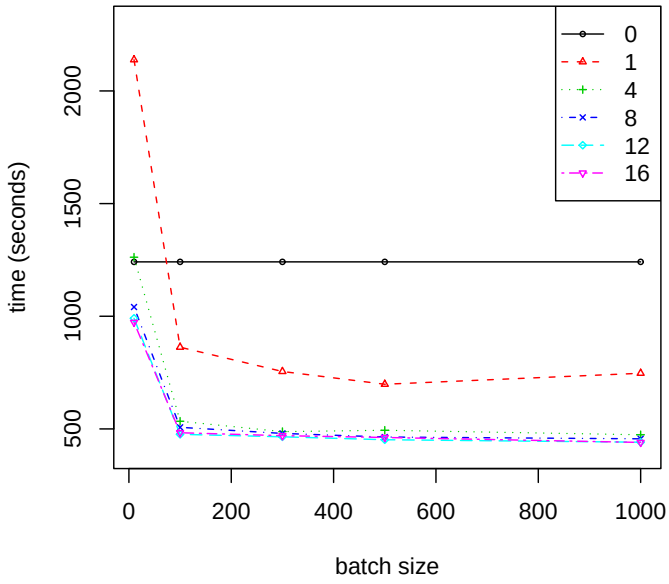

Fig. 3. Influence of number of threads and batch size on import time

urations (excluding the one with TDB support). Notice that when the number of worker threads is equal to zero, a serial Greenstone import is used. While a manifest is still provided to this process (as explained above) it always contains the full ten thousand documents and thus the batch size is effectively ignored and the processing time remains essentially constant.

The results show that increasing batch size decreases average time up to around a batch size of n/w (where n in the total number of documents and w the number of worker threads) but after that point it causes processing time to (very slightly) increase. With smaller size batches, there is more overhead of worker threads doing IO reading in the XML manifest files, while at larger batch sizes some worker threads are being starved of documents and sitting idle while other threads finish processing.

However it is interesting to notice that in both Figure 2 and Figure 3 the import process achieves a minimum processing time of around six minutes, after which processing time actually increases marginally as more threads are added. While we initially suspected that this was just the overhead of managing more threads, it was these results that led us to take a closer look at what was actually using the time. By reviewing profiling tool output and measuring the system time used (as compared to real time), it was discovered that a significant amount

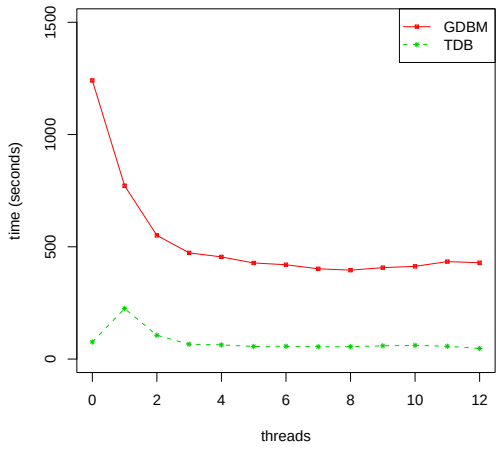

Fig. 4. Timing with comparing GDBM with TDB

of time was spent waiting for opening and closing connections to the GDBM database.

Figure 4 shows the result of replacing the GDBM database used by default by Greenstone with TDB, which supports parallel readers and writers and thus is better suited for parallel processing. Once again the test used the configuration where both unique identifier and encoding are assigned. From these results it is clear that by using a database that supports parallel access we can avoid the point of diminishing returns created by the basic file locking we implemented around GDBM. The speed-up in changing to this database system is striking—processing times a magnitude smaller than traditional Greenstone. Also interesting is that TDB exhibits the decrease in performance expected (but not seen in the earlier graph) when threads equals one (and thus the process has all of the overheads of parallel processing but none of the benefits). However there were also some unexpected results; one being that serial Greenstone using TDB performed almost as well as parallel Greenstone using TDB with eight or more threads (with the latter only being a second or two faster) and the other being that TDB's performance seems far more variable than GDBMs (some TDB runs took four times longer than others).

6 Conclusions and Future Work

The results shown above have exposed several questions that we must answer before proceeding with parallel processing support to the global Greenstone community. We applied profiling tools to locate several of the IO or other bottlenecks limiting the performance of parallel processing. While we were able to remove

several of these by altering the configuration (for instance, specifying the encoding of the import documents manually rather than leaving Greenstone to automatically determine this), there are still issues with IO which we plan to investigate by reviewing file IO in Greenstone, simulating different drive configurations, and altering filesystem and disk cache settings.

However, most exciting has been the performance gained by moving to TDB as the database. These results are still preliminary and need to be carefully explored. In particular we need to determine if the results for standard, serial Greenstone importing is accurate, since on the surface it seems to marginalize any work on supporting parallel processing (at least for eight cores, although more populous CPUs are just around the corner).

The current results have already demonstrated that there is some benefit in parallelising the import process in Greenstone and that the development of parallelisation support leads to optimisations that benefit the general architecture as well. Given the increasing proliferation of high-core CPUs (with 12-cores available at the time of writing), multi-processor-capable systems and commodity publicly-available high performance computing in the form of pay-per-use virtual utility computing, it is essential that software like Greenstone and its peer systems plan for parallelisation as their architectures evolve.

References

1. Adams, D.: Papers past: browse access for online New Zealand newspapers. Microform & Imaging Review 34, 22–27 (2005)
2. Bainbridge, D., Witten, I.H., Boddie, S., Thompson, J.: Stress-testing general purpose digital library software. In: Proceedings of the 13th European conference on Research and advanced technology for digital libraries. pp. 203–214. ECDL'09, Springer-Verlag, Berlin, Heidelberg (2009), http://portal.acm.org/citation.cfm?id=1812799.1812828
3. Barroso, L.A., Dean, J., Hölzle, U.: Web search for a planet: The Google cluster architecture. IEEE Micro 23, 22–28 (March 2003), http://portal.acm.org/citation.cfm?id=776692.776716
4. Goetz, B.: Java theory and practice: Thread pools and work queues. Tech. Rep. j-jtp0730, IBM, New York, United States (2002), http://www.ibm.com/developerworks/library/j-jtp0730/index.html
5. Manghi, P., Pagano, P., Ioannidis, Y.: Second Workshop on Very Large Digital Libraries: in conjunction with the European Conference on Digital Libraries. SIGMOD Rec. 38, 46–48 (June 2010), http://doi.acm.org/10.1145/1815948.1815959
6. Rasmussen, E.M.: Introduction: parallel processing and information retrieval. Inf. Process. Manage. 27, 255–263 (June 1991), http://portal.acm.org/citation.cfm?id=117658.117659
7. Stanfill, C.: 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. pp. 413–428. SIGIR '90, ACM, New York, NY, USA (1990), http://doi.acm.org/10.1145/96749.98247
8. Witten, I.H., Bainbridge, D., Nichols, D.M.: How to Build a Digital Library, Second Edition. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2nd edn. (2009)