SAND2010-0457C

ParaText: Scalable Text Modeling and Analysis

Daniel M. Dunlavy
Comp. Sci. & Informatics
Sandia National Laboratories
P.O. Box 5800, M/S 1318
Albuquerque, NM 87185-1318
dmdunla@sandia.gov

Timothy M. Shead
Data Analysis & Visualization
Sandia National Laboratories
P.O. Box 5800, M/S 1323
Albuquerque, NM 87185-1323
tshead@sandia.gov

Eric T. Stanton
Data Analysis & Visualization
Sandia National Laboratories
P.O. Box 5800, M/S 1323
Albuquerque, NM 87185-1323
etstant@sandia.gov

ABSTRACT

Automated analysis of unstructured text documents (e.g., web pages, newswire articles, research publications, business reports) is a key capability for solving important problems in areas including decision making, risk assessment, social network analysis, intelligence analysis, scholarly research and others. However, as data sizes continue to grow in these areas, scalable processing, modeling, and semantic analysis of text collections becomes essential. In this paper, we present the ParaText Text Analysis Engine, a distributed software framework for processing, modeling, and analyzing collections of unstructured text documents. Results on several document collections using hundreds of processors are presented to illustrate the flexibility, extensibility, and scalability of the ParaText system.

Categories and Subject Descriptors

I.2.7 [**Computing Methodologies**]: Natural Language Processing—*text analysis*; H.3.3 [**Information Systems**]: Information Search and Retrieval—*retrieval models*

General Terms

Algorithms, design, performance, text analysis

1. INTRODUCTION

Automated processing, modeling, and analysis of unstructured text (news documents, web content, journal articles, etc.) is a key task in many data analysis and decision making applications. In many cases, documents are modeled as term or feature vectors and latent semantic analysis (LSA) [3] is used to model latent, or hidden, relationships between documents and terms appearing in those documents. LSA supplies conceptual organization and analysis of document collections by modeling high-dimension feature vectors in many fewer dimensions.

Most of the past work on scaling LSA modeling has focused on the computation of a truncated singular value decomposition (SVD) of the term-by-document matrix modeling the data. In this paper, we concentrate on the scalability of the full pipeline—from ingesting text and modeling the data, to analysis tasks such as information retrieval and document similarity. Moreover, we have implemented several alternate methods for parts of the pipeline that require significant inter-processor communication and present results of strong scaling studies for these methods.

2. RELATED WORK

As noted above, scalability studies associated with LSA have focused mainly on the computation of the SVD (e.g., [5, 8]). Much of that work uses scaling studies on general algorithms for singular value or eigenvalue decompositions (see e.g. [1] and references within). Other approaches for increasing computation performance of LSA include alternatives to the SVD for dimensionality reduction (e.g., [9]) and feature selection to reduce the size of the term-document matrices (e.g., [11]). The goal of our work is to investigate the best use of distributed memory architectures for the entire process of text analysis from raw data ingestion to semantic modeling to application analysis using the LSA models. In ParaText, we have created a modular system for prototyping new algorithms and studying the scalability of the entire process of text modeling and analysis.

3. THE PARATEXT SYSTEM

The ParaText system is comprised of a collection of text analysis components designed to work within the Titan data processing pipeline [10], where data sources, filters, and sinks can be combined in arbitrary ways (Figure 1). The ParaText components can be used either as a C++ programming library, or via a web service that implements a RESTful API [4] atop an Apache httpd server. Thus, the ParaText capabilities outlined in this report can be accessed using a variety of programming languages and environments.

3.1 Text Extraction

The first part of the pipeline consists of filters for extracting and transforming text. With the exception of determining which files should be processed on which processors, the filters described in this section all parallelize extremely well.

Document Ingestion. At the beginning of the analysis pipeline, the DOCUMENT INGESTION filter is responsible for partitioning a set of documents and loading them into memory as a table where each row corresponds to a document. Currently, this filter is parameterized using a list of documents that is distributed to each processor. Each instance

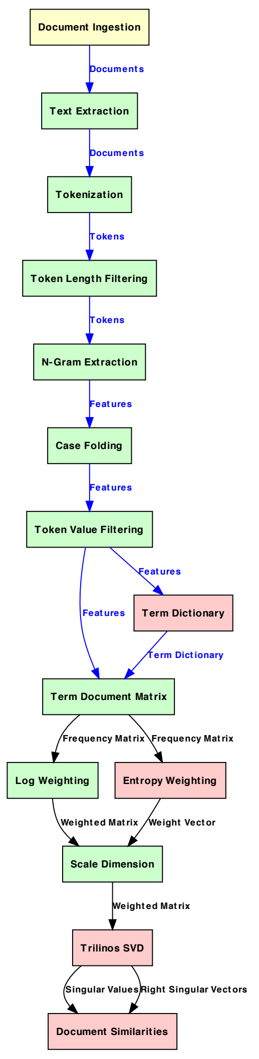


Figure 1: Data flow pipeline of the ParaText System. Filters in yellow boxes require minor interprocessor communication, red boxes indicate significant amounts of communication.

of the filter on each processor then reads its own subset of files from the list. We have implemented several partitioning strategies that control how processors determine which files to load locally. The *Document* partitioning strategy does a simple round-robin distribution where each process loads $1/p$ documents from the list. This strategy is simple to implement and requires no communication, but can lead to imbalanced loading as some processors may accumulate documents that are smaller- or larger-than-average. The *Bytes* partitioning strategy tries to balance loading by assigning files to processors so that each processor receives roughly the same number of input bytes. This process is a variation on bin packing, which is a combinatorial NP hard problem, so we follow a heuristic approach of maintaining a “bucket” for each processor, then inserting each file, in descending order of file size, into whichever bucket contains the fewest number of file bytes at the moment. Early versions of this approach (which we call *Thrash*) did not require communication, but performed poorly due to filesystem contention as every processor simultaneously tried to retrieve the size of every file in the list. Subsequent versions use a single processor to retrieve file sizes and distribute them to the remaining processes before beginning the bucketing process.

Text Extraction. Once the local table of documents to be loaded has been created, we use MIME type information to extract text, using the TEXT EXTRACTION filter. This filter contains a collection of strategy objects, each of which is responsible for extracting text from documents of a given MIME type. Note that the text extraction strategies can perform arbitrarily-complex operations to extract text from a document, including extracting text from binary file formats such as PDF or word-processing documents, extracting metadata from images, or even performing optical character recognition on the contents of an image. For the experiments presented here, we relied on a default extraction strategy that handles all `text/*` MIME types. The text of each document is stored as a Unicode string using UTF-8 encoding, so the system is capable of working with mixed-language text.

Tokenization. Following text extraction, the TOKENIZATION filter converts document text into a table of tokens. Tokenization is performed by splitting the document text into tokens using delimiters specified as half-open ranges of Unicode codepoints. Delimiters are divided into two categories: “dropped” and “kept” — “dropped” delimiters are discarded from the output, while “kept” delimiters are retained in the output as individual tokens. This makes it easy to tokenize logosyllabic scripts such as Chinese, Korean, and Japanese by specifying an entire range of logograms as “kept” delimiters, so that individual glyphs become tokens.

Token Length Filtering. We use two instances of the TOKEN LENGTH filter to discard tokens that are either too short or too long. This improves the downstream analysis by reducing noise in the data models.

N-Gram Extraction. The N-GRAM EXTRACTION filter converts individual tokens into n-grams and is parameterized for selection of arbitrary values of n. We used unigrams ($n = 1$) for all experiments in this paper.

Case Folding. We use the CASE FOLDING filter to transform the resulting tokens to a form where they can be used in case-insensitive comparisons. This transformation is carried-out using the rules provided by Unicode for case-folding, so the results are valid for all languages supported by Unicode.

Token Value Filtering. To provide filtering of stop-words, we use TOKEN VALUE filter that is parameterized by a list of tokens to be discarded. For these experiments, we used the standard stop word list from the SMART project [7].

3.2 Term Dictionary Creation

Once each processor determines the list of terms in its local data (i.e., documents), the TERM DICTIONARY filter creates a global “term dictionary” where each term is listed exactly once. Because this process necessitates communication of large numbers of strings between processors, we created several different implementations for testing: in *N-to-1*, every processor sends its local terms to processor 0, which creates the global dictionary and broadcasts the results back to every processor. For *N-to-N*, each processor broadcasts its local terms to every other processor, which then creates its own copy of the global dictionary. In the *Binary Tree* approach, each processor sends its local terms to a "neighbor", which consolidates them with its own local terms, sending the results to a "super neighbor", and-so-on until the complete global dictionary has been created on one process that broadcasts the results to the others. The *Round Robin* approach involves processor k sending its local terms to processor $(k + 1) \mod p$, where they are consolidated with the local terms. This process runs p times, so that every term eventually reaches every processor. Finally, we have a *MapReduce* approach that uses the MapReduce-MPI library [6] to consolidate and distribute terms.

3.3 Term Document Matrix Creation

Given the list of local terms and the global term dictionary computed in the TERM DICTIONARY filter, each processor uses the FREQUENCY MATRIX filter to create its part of a sparse, distributed term-document frequency matrix (no inter-processor communication is required). For each term from the local term list, the document ID is mapped to a column of the frequency matrix, and the global term dictionary is used to lookup the frequency matrix row, so that the appropriate frequency value can be incremented. Two methods are implemented in the FREQUENCY MATRIX filter for term dictionary lookup: *Global* lookup is a naive approach where each term is simply looked-up in the global term dictionary with $O(m \log m)$ performance; *Global+Local* lookup is a more sophisticated two-stage approach where local lookup results are "cached" in a smaller lookup table to improve performance when tokens are repeated throughout and across documents (which is often the case in practice).

3.4 Term Weighting

Once the term-document frequency matrix is generated, it must be weighted before computing the SVD in order to model the importance of terms both within each document as well as across the entire document collection. In this paper, we focus on the standard *log-entropy* weighting scheme [3] employed in many LSA studies; this will help illustrate the challenges associated with term weighting on distributed memory architectures. This weighting scheme involves the product of local quantities (frequencies of terms within each document) and global quantities (entropies of terms across the entire document collection). In ParaText, the local and global computations are separated into different filters; for this weighting scheme, these are the LOG WEIGHTING and ENTROPY WEIGHTING filters, respectively.

The entropy of term i across the collection is computed as

$$g_i = -\frac{1}{n}\sum_{j=1}^{n}\frac{tf_{ij}}{gf_i}\log\frac{tf_{ij}}{gf_i}\ , \qquad (1)$$

where tf_{ij} is the frequency of term i in document j and gf_i is the global frequency of term i across the collection. Thus, interprocessor communication is required both in computing gf_i for each term and computing the sum in g_i for each term. We have implemented several methods to study the impact of these inter-processor communication requirements. In the *N-to-1* method, every processor computes its local values of gf_i and sends those to processor 0, which sums the values and broadcasts the results back to every processor. The sums for g_i are then computed in a similar fashion. In the *N-to-N* method, gf_i and g_i are first computed locally and then results are broadcast to all other processors for computing the global values. In both methods, there is the option to broadcast the locally computed values using either dense or sparse vectors. Once the local and global term weights are computed, the SCALE DIMENSION filter then applies these weights to the matrix appropriately.

3.5 Singular Value Decomposition

To compute the SVD of the weighted term-document matrix, A, ParaText wraps the distributed block Krylov Schur method from the Anasazi package of the Trilinos solver library [1]. Using shallow copies of data into the sparse matrix class in Trilinos, we avoid data replication. The rank k truncated SVD of A is computed as $A_k \approx U_k \Sigma_k V_k^{\mathsf{T}}$, where U_k, Σ_k, and V_k are matrices containing the left singular vectors, singular values, and right singular vectors, respectively.

3.6 Corpus Analysis

Document Similarities. An important application of text modeling in general and LSA in particular [2] is determination of the topical or conceptual relationships between documents in a large collection. To model these relationships, pariwise document similarity or distance measures are often computed. In ParaText, document similarities are computed as the cosine values between scaled LSA document vectors (in $V_k\Sigma_k$). Thus, the similarity between documents i and j are computed as $\langle \hat{V}_i, \hat{V}_j \rangle / (\|\hat{V}_i\|\ \|\hat{V}_j\|)$, where $\langle \cdot, \cdot \rangle$ is the standard inner product and $\hat{V}_i$ is the ith row of $V_k\Sigma_k$.

4. RESULTS

Computing Environment. The system we used for testing is comprised of 256 compute nodes, each with a Dual 3.6 GHz Intel EM64T processor and 6 GB RAM. The system's high-speed message passing fabric is Infiniband, and the file system is Lustre v1.4.11.1 running on 47 servers and 186 object storage targets, yielding a bandwidth of 15 GB/second.

Data. The data used in the experiments presented here consist of a subset of HTML documents in the 2007 Spock Challenge test set[1]. For experiments involving 64 processors, 2458 documents with 669940 unique terms (0.12% matrix density) were used; and for experiments using 512 processors, 45945 documents with 4,440,327 unique terms (0.017% matrix density) were used. Note that this decrease in frequency matrix density for problems with more documents is typical in text analysis.

Strong Scaling Studies. Figure 2 presents the results of strong scaling studies using 64 processors for the filters mentioned in Section 3 having more than one implementation. The plots all show mean CPU times (with error bars denoting standard sample error) over three runs as a function of the number of processors. We see that there are significant differences in the document partitioning methods (Fig. 2a), where partitioning by *Bocuments* appears the most scalable. Also the *N-to-N* methods appear to perform slightly (but not statistically significantly) better than the *N-to-1* methods (Figs. 2b and 2d, even though the former methods require more overall communication. In terms of sizes of packets being communicated, using dense over sparse arrays in the ENTROPY WEIGHTING filter appears better for fewer processors (Fig. 2d). However, as the number of processors increases (and thus the local term dictionaries become

[1]http://challenge.spock.com/

more sparse due to fewer documents on each processor), the sparse data passing has potential for improved performance (as demonstrated by the trajectory of improvement in the figure). Since *MapReduce* for term dictionary creation seems promising as the number of processors increases (Fig. 2b), we plan to explore additional use of MapReduce in future versions of ParaText where appropriate. Finally, caching of locally computed information reduces the overall computational costs associated with distributed term dictionary creation (Fig. 2c), and we will be investigating more pervasive use of this idea throughout the ParaText pipeline.

Figure 3 presents the results of strong scaling studies for all pipeline filters. Figure 3a illustrates that for the larger problem using 512 processors, most of the filters requiring little or no inter-processor communication achieve strong scalability as expected, although improvement is still possible for the document ingestion and term document matrix creation. For the filters requiring significant inter-processor communication, we see that more work is needed to achieve useful speedups as we increase the number of processors (Figure 3b). We leave this as future work.

5. CONCLUSIONS

We presented the ParaText Text Analysis Engine, a set of scalable methods for analyzing large document collections. We described several alternate approaches for methods requiring significant inter-processor communication and presented the results of strong scaling studies illustrating the performance of those approaches on HTML documents. We have also identified several directions for potential improvements in terms of term dictionary creation (e.g., using distributed merge sort) and matrix computations (e.g., tuned matrix vector products using graph partitioning and load balancing). Previous LSA scalability research has focused on the SVD computation. In this paper, we illustrate the scalability of the entire process from raw data to analysis for LSA models, illustrating that there are significant challenges associated with LSA scalability beyond matrix computations.

6. ACKNOWLEDGMENTS

This work was funded by the Laboratory Directed Research & Development (LDRD) program at Sandia National Laboratories, a multiprogram laboratory operated by Sandia Corporation, a Lockheed Martin Company, for the United States Department of Energy's National Nuclear Security Administration under Contract DE-AC04-94AL85000.

7. REFERENCES

[1] C. G. Baker, U. L. Hetmaniuk, R. B. Lehoucq, and H. K. Thornquist. Anasazi software for the numerical solution of large-scale eigenvalue problems. *ACM TOMS*, 36(3):13:1–13:23, 2009.

[2] P. Crossno, D. Dunlavy, and T. Shead. LSAView: A tool for visual exploration of latent semantic modeling. In *Proc. IEEE VAST*, 2009.

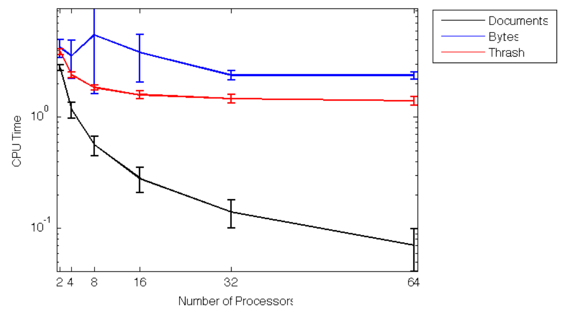
[3] S. T. Dumais. Improving the retrieval of information from external sources. *Behavior Research Methods, Instruments, & Computers*, 23(2):229–236, 1991.

[4] R. T. Fielding and R. N. Taylor. Principled design of the modern web architecture. *ACM TOIT*, 2(2):115–150, 2002.

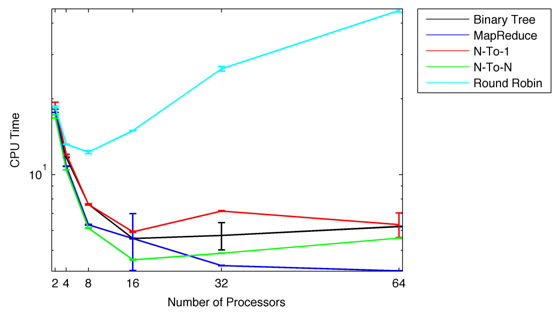
[5] D. Martin and M. Berry. Parallel svd for scalable information retrieval. In *Proc. Parallel Matrix Algorithms and Applications*, 2000.

[6] S. Plimpton and K. Devine. MapReduce-MPI Library. `http://www.sandia.gov/~sjplimp/mapreduce.html`.

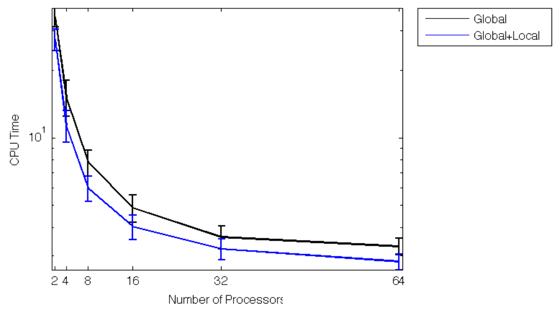
[7] G. Salton, editor. *The SMART Retrieval System: Experiments in Automatic Document Processing.*



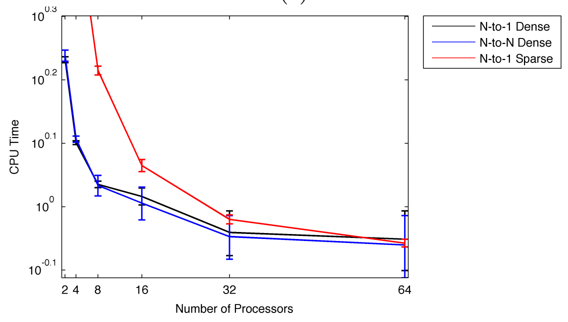
(a)



(b)



(c)



(d)

Figure 2: Strong scaling studies using 64 processors for methods associated with (a) document partitioning, (b) term dictionary creation, (c) frequency matrix creation and (d) entropy weighting.

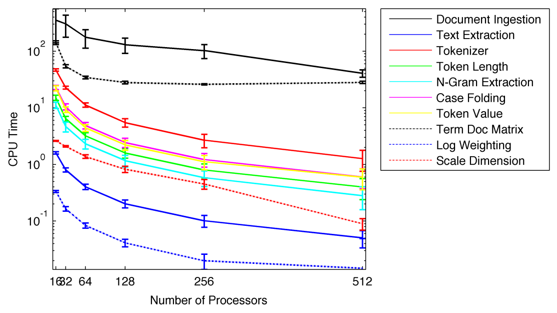
Prentice-Hall, 1971.

[8] S. Vigna. Distributed, large-scale latent semantic analysis by index interpolation. In *Proc. InfoScale*, pages 1–10, 2008.

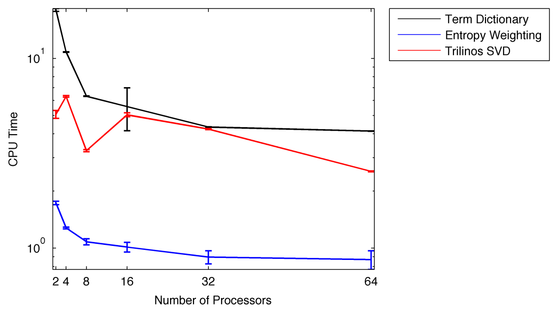
[9] D. Widdows and K. Ferraro. Semantic vectors: a scalable open source package and online technology management application. In *Proc. LREC*, 2008.

[10] B. Wylie and J. Baumes. A unified toolkit for information and scientific visualization. In *Proc. SPIE*, 2009.

[11] J. Yan, S. Yan, N. Liu, and Z. Chen. Straightforward feature selection for scalable latent semantic indexing. In *Proc. SDM*, pages 1159–1170, 2009.



(a)



(b)

Figure 3: Strong scaling studies for the ParaText pipeline illustrating performance of filters with (a) little or no inter-processor communication using 512 processors and (b) significant inter-processor communication using 64 processors.