Technical Report

Department of Computer Science
and Engineering
University of Minnesota
4-192 Keller Hall
200 Union Street SE
Minneapolis, MN 55455-0159 USA

TR 11-023

Mapping Multi-Layer Baysian LDA to Massively Parallel Supercomputers

Kuo-wei Hsu, Ching-yung Lin, and Jaideep Srivastava

October 10, 2011

Mapping Multi-Layer Baysian LDA to Massively Parallel Supercomputers

Kuo-Wei Hsu[1]
University of Minnesota
Minneapolis, MN USA
kuowei@cs.umn.edu

Ching-Yung Lin
IBM T. J. Watson Research Center
Hawthorne, NY USA
chingyung@us.ibm.com

Jaideep Srivastava
University of Minnesota
Minneapolis, MN USA
srivasta@cs.umn.edu

***Abstract*—LDA, short for Latent Dirichlet Allocation, is a hierarchical Bayesian model for content analysis. LDA has seen a wide variety of applications, but it also presents computational challenges because the iterative computation of approximate inference is required. Recently an approach based on Gibbs Sampling and MPI is proposed to address these challenges, while this report presents the work that maps it to a massively parallel supercomputer, Blue Gene. The work enhances the runtime performance by utilizing special hardware architecture of Blue Gene such as dual floating-point unit and by using general programming/compiling techniques such as loop unfolding. Results from the empirical evaluation using a real-world large-scale data set indicate the following findings: First, the use of dual floating-point unit contributes to a significant performance gain, and thus it should be considered in the design of processors for computationally intensive machine learning applications. Second, although it is a simple technique and most compilers support it, loop unfolding improves the performance gain even further. Since loop unfolding is general enough to be applied to other platforms, this report suggests that compilers should perform loop unfolding in a more intelligent manner.**

I. INTRODUCTION

Latent Dirichlet Allocation, or LDA, is a state-of-the-art machine learning algorithms for content analysis. LDA is a multi-level hierarchical Bayesian model proposed by Blei, Ng, and Jordan in [2], and it describes a model that generates a corpus of documents based on a bag of words and latent topics. It has attracted increasing attention and applied to various applications. Here are some examples: Shen et al. in [10] utilize LDA to find friends in blog data; Bhattacharya and Getoor propose in [1] the use of LDA for entity resolution; Maskeri, Sarkar, and Heafield in [7] apply LDA to source code mining; Dredze et al. use in [3] the use of LDA to generate email keywords; Henderson and Eliassi-Rad in [5] apply LDA to graph mining.

As social network analysis emerges as a rampant research topic, content analysis for social media becomes significant. LDA is widely used on social network analysis research but presents computational challenges. Mainly concerning a computationally efficient implementation of LDA, this report presents challenges and solutions of mapping LDA based on Gibbs Sampling and MPI (which is called PLDA, available on http://code.google.com/p/plda/, and presented by Wang et al. in [11]) to massively parallel supercomputers. Below are challenges:

- Algorithm-level: What intermediate data could be distributed and what could not?
- Source code-level: What types of modification should be done to what pieces of source code in order to guide compilers to utilize the special hardware architecture?
- Instruction code-level: How processing units and, for example, pipelines, could be utilized?

As shown later in this report, the runtime performance is enhanced because of the use of the special hardware architecture of Blue Gene, such as dual floating-point unit and also the use of general techniques such as loop unfolding. Contributions of this report are summarized below:

- For the machine learning community, this report presents a solution to exploit the architecture of massively parallel supercomputer, the

[1] This report presents the work Hsu made during his visit to IBM T. J. Watson Research Center in July-August, 2010.

architecture as well as programming/compiling techniques for computationally intensive tasks.

- For the computer architecture community, this report presents another application that benefits from massively parallel supercomputers, and it also provides reference points for the design of processors and/or compilers for computationally intensive machine learning applications.

The rest of this report is organized as follows. Sec. II provides background; Sec. III discusses the employed techniques; Sec. IV gives the empirical evaluation; Sec. V concludes this report.

II. BACKGROUND

LDA. Blei, Ng, and Jordan in [2] describe LDA using a graphical model, as shown in Figure 1. The description is as follows: Given a corpus of *M* documents and *N* words (i.e. the size of the bag of words is *N*), a document is viewed as a mixture of *K* topics while the distribution of topics follows a Dirichlet distribution. The latent topics are unobservable, so they are usually the focus on the analysis. Moreover, α is the prior on the topic distribution; β is the prior on the word distribution; θ is the joint distribution of a topic mixture; *w* is a word in a document; *z* is the topic of *w*.

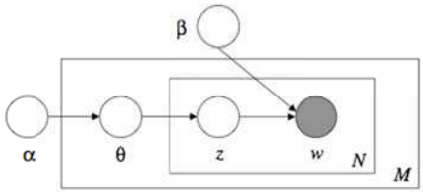

Figure 1. LDA represented as a graphical model [2]: A circle is a random variable, a plate is a replicate, and an arc indicates dependence of casuality.

Blei, Ng, and Jordan in [2] describe how LDA generates a document. It could be summarized as follows: The number of words, *N*, is drawn from a Poisson distribution first and then θ is drawn from a Dirichlet distribution with the parameter α. Next, for each word, a topic is drawn from a Multinomial distribution with the parameter θ first and then a word is drawn with the probability conditioned on the topic and the parameter β.

Blei, Ng and Jordan propose in [2] using an algorithm based on Expectation Maximization, or simply EM, to obtain a maximum-likelihood estimate of parameters. It is an iterative algorithm that has two steps in each iteration. The first step, E-step, is to infer the distributions over topics for each document. The second step, M-step, is to use the inference result to calculate the maximum likelihood and to update model parameters.

To perform the inference in the E-step, Blei, Ng and Jordan in [2] propose using variational Bayes. Minka and Lafferty in [8] propose using an algorithm called Expectation Propagation or EP. Griffiths and Steyvers in [4] propose using Gibbs Sampling that is basically a Markov-chain Monte Carlo method. Furthermore, various methods are proposed to implement LDA in a distributed environment (in order to speed up the learning process of LDA). For example, Newman et al. in [9] propose two methods, AD-LDA and HD-LDA, to perform Gibbs Sampling in a distributed environment. Wang et al. in [11] propose using MPI to implement AD-LDA. In the learning process given in [11], a topic is assigned as follows:

$$p\left(z_{d,i}=k \middle| w_{d,i}=v, W_{\backslash(d,i)}, Z_{\backslash(d,i)}\right) \propto \left(TD_{d,k,\backslash(d,i)}+\alpha\right)\cdot\frac{WT_{v,k,\backslash(d,i)}+\beta}{\sum_{v'} WT_{v',k,\backslash(d,i)}+V\cdot\beta} \quad (1)$$

where $1 \le k \le K =$ the specified number of topics; the left-hand side term is a conditional probability that the topic $z_{d,i}$ assigned to the word $w_{d,i}$, which is the *i*-th word in the document *d*; $W_{\backslash(d,i)}$ is a set of words obtained from removing $w_{d,i}$ from the corpus; $Z_{\backslash(d,i)}$ is the set of topic assignments for $W_{\backslash(d,i)}$; $TD_{d,k,\backslash(d,i)}$ (where *TD* is short for topic-document) is the co-occurrence that the topic *k* in the document *d* when $w_{d,i}$ and $z_{d,i}$ are removed; $1 \le v \le V =$ the vocabulary size; $WT_{v,k,\backslash(d,i)}$ (where *WT* is short for word-topic) is the co-occurrence that the word *v* is assigned to the topic *k* when $w_{d,i}$ and $z_{d,i}$ are removed. When a topic assignment of a word is updated, so are *TD* and *WT*. The estimates of the parameters converge after a number of iterations of assigning-and-updating.

Blue Gene. It is an architecture on which a family of massively parallel supercomputers relies. Blue Gene/L is the target platform of the work presented in this report. Two main features of Blue Gene/L make it attractive for computationally intensive tasks such as large-scale machine learning,

content analysis, or social network analysis. First, Blue Gene/L contains a large number of nodes, including compute and I/O nodes. Second, Blue Gene/L supports fast internal communications, e.g. peer-to-peer communication via a 3-dimensional network and global collective communication via a collective network.

Compute nodes are dedicated to computation since each runs a lightweight operating system, while I/O nodes are responsible for file operations and external communication. The use of a lightweight operating system reduces the overhead and hence allows compute nodes to provide full computing power. However, also because of this, only one process per processor may be running at a time. Memory space sharing is restricted as well.

III. MAPPING LDA TO BLUE GENE

In Blue Gene/L, a task that needs a large amount of memory has to allocate many nodes. Figure 2 gives an overview of parallelization in the algorithm-level. In Figure 2, a corpus is distributed to available compute nodes. Each compute node has a subset of the corpus, while it has its own copy of the vocabulary or the set of words. In each iteration, all compute nodes use *MPI_Allreduce* to aggregate as well as broadcast distributions of words. Each compute works on local distributions of documents over words but global distributions of words.

That is, a single copy of *TD* in (1) is distributed over nodes and the intermediate data for *TD* in each node is used in calculation and updated locally; multiple copies of *WT* in (1) are distributed over nodes, while they are used in calculation, aggregated, and updated globally.

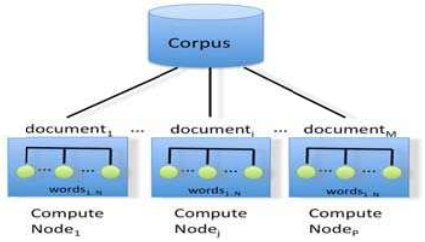

Figure 2. Parallelization of LDA in the algorithm-level.

So far the discussion is about how to manage the data, i.e. usually large data matrices, in (1). This is related to algorithm-level optimization. Next, the discussion is about how to speed up the (local) calculation in each node. This is related to source code-level and instruction code-level optimization. As shown in (1), the (local) calculation in each node involves referring each entry in the subset of *TD* stored locally, referring each entry in the local copy of *WT*, and arithmetic operations such as addition (including summation), multiplication, division. Thus, how to speed up these operations becomes the key for the performance enhancement.

Since compilers are responsible for most of instruction code-level optimization, now the focus becomes source code-level optimization or how to modify source code such that source code can help compilers generate more efficient instruction code. In Blue Gene/L, each processor has a special dual floating-point unit. A processor can issue a load/store operation and a floating-point operation per clock cycle. Running (some) floating-pointing operations in parallel has the potential to enhance the runtime performance of some computationally intensive tasks. Utilizing pipeline could also enhance the runtime performance, and it is to keep the processing units as busy as possible [6]. Figure 3 gives an example.

```
for (int k = 0; k <K; k++) {
        topic_doc[k] =(doc_topic_occ[k] + alpha)/(doc_topic_occ_sum + alpha * K);
}
```

```
double tmp =  doc_topic_occ_sum  + alpha * K;
Int x = K - ( K % 5 );
for (k = 0; k < x; k += 5) {
        topic_doc[k] = (doc_topic_occ[k] + alpha) / tmp;
        topic_doc[k+1] = (doc_topic_occ[k+1] + alpha) / tmp;
        topic_doc[k+2] = (doc_topic_occ[k+2] + alpha) / tmp;
        topic_doc[k+3] = (doc_topic_occ[k+3] + alpha) / tmp;
        topic_doc[k+4] = (doc_topic_occ[k+4] + alpha) / tmp;
}
for (k = x; k < K; k++) {
        topic_doc[k] = (doc_topic_occ[k] + alpha_) / tmp;
}
```

Figure 3. An cxample of loop unfolding.

The basic idea illustrated in Figure 3 is to run lines of code in batch. The piece of code shown in Figure 3 is to update *TD* in (1). It transforms or unfolds a loop incremented by 1 into two loops. The first one is a new loop incremented by 5. The second is like the original one and is simply to handle the remaining part of the original loop. For example, the main line of code in the original loop has 5 loads, 2 additions, 1 multiplication, and 1 division, 1 store. A line in the unfolded loop has 3 loads, 1 addition, 1 division, and 1 store. The difference in numbers of operations is significant especially when *K* is large. Moreover, because fives line of code in the unfolded loop share the same

reference to the variable *tmp*, compilers now have more information to overlap instructions and exploit pipeline. Unfolding a loop where there are simple arithmetic operations would increase the utilization of pipeline stages in a clock cycle and it would further keep arithmetic units busy. Compilers can analyze some loops, but effort from developers is required for others. Loop unfolding is general enough to be applied to other platforms.

IV. EMPIRICAL EVALUATION

Experiments are conducted on Blue Gene/L. The number of nodes is 512, and the amount of memory on a compute node is 512 MB. A real-world corpus presenting document-term-frequency is used in experiments. It contains 4,321,278 documents and 56,710 words. As for LDA parameters, α is 0.1 and β is 0.01. The numbers of latent topics (K) and iterations are 200 and 150, respectively.

Various optimization steps are considered. An IBM XL compiler is responsible for some optimization steps, and so is modification source code such as loop unfolding. These steps are described below: Step 0 is the baseline, where no optimization is applied. Step 1 specifies 3rd level compiler optimization and PowerPC 440. Step 2 is from Step 1, but it additionally specifies dual floating-point unit. Step 3 is from Step 2, but it specifies the 4th level compiler optimization. Step 4 is from Step 2, but it specifies the 5th level compiler optimization. Step 5 is from Step 4, but it uses loop unfolding. Table I summarizes the runtime performance. It also implies that loop unfolding could further enhance the runtime performance.

TABLE I. THE RUNTIME PERFORMANCE W.R.T. OPTIMIZATION STEPS.

Steps	Time in minutes
baseline (no optimization)	101.66
O3 qarch=440	36.36
O3 qarch=440d	24.13
O4 qarch=440d	24.04
O5 qarch=440d	23.86
O5 qarch=440d & loop unfolding	20.7

Next, speedup is employed to measure the relative performance. It is defined below:

$$Speedup = \frac{T_0}{T_i} = \frac{\text{Time for Step 0 (baseline)}}{\text{Time for Step i}} \quad (2)$$

Figure 4 shows a comparison with respect to speedup between optimization steps. The following findings can be obtained from Figure 3. First, the use of the special dual floating-point unit significantly increases speedup. Second, different optimization levels done by the compiler provide similar speedup. Third, loop unfolding corresponds to the increase in speedup from 4.3 to 4.9.

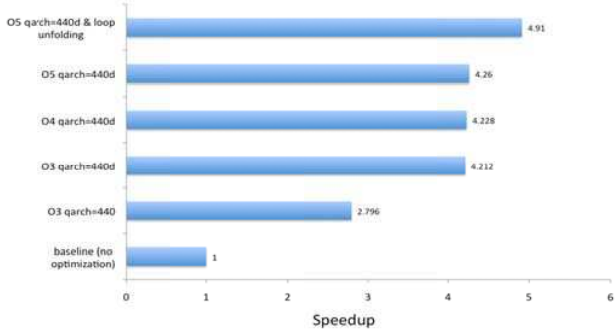

Figure 4. Speedup in different optimization steps against baseline.

V. CONCLUSIONS

This report presents the work that maps a recently proposed parallel implementation of LDA to Blue Gene. Experiments are conducted with a real-world large-scale data set. Results show that the special architecture of Blue Gene such as dual floating-point unit and the general techniques such as loop unfolding could further enhance the runtime performance. Future work includes extending the work to other algorithms and platforms.

REFERENCES

[1] I. Bhattacharya and L. Getoor, "A Latent Dirichlet Model for Unsupervised Entity Resolution," SDM, 2006.

[2] D. M. Blei, A.Y. Ng, and M. I. Jordan, "Latent Dirichlet allocation," JMLR, 2003.

[3] M. Dredze, H. M. Wallach, D. Puller, and F. Pereira, "Generating Summary Keywords for Emails Using Topics," IUI, 2008.

[4] T. L. Griffiths and M. Steyvers, "Finding scientific topics," Proc. of the National Academy of Sciences of U.S. 101, 2004.

[5] K. Henderson and T. Eliassi-Rad, "Applying Latent Dirichlet Allocation to Group Discovery in Large Graphs," SAC, 2009.

[6] IBM, "IBM System Blue Gene Solution: Application Development," 2007.

[7] G. Maskeri, S. Sarkar, and K. Heafield, "Mining Business Topics in Source Code using Latent Dirichlet Allocation," ISEC, 2008.

[8] T. Minka and J. Lafferty, "Expectation-propagation for the generative aspect model," UAI, 2002.

[9] D. Newman, A. Asuncion, P. Smyth, and M. Welling, "Distributed inference for latent Dirichlet allocation," NIPS, 2007.

[10] D. Shen, J.-T. Sun, Q. Yang, and Z. Chen, "Latent Friend Mining from Blog Data," ICDM, 2006.

[11] Y. Wang, H. Bai, M. Stanton, W.-Y. Chen, and E. Y. Chang, "LDA: Parallel Latent Dirichlet Allocation for Large-scale Applications," AAIM, 2009.