Published in: *Journal of Intelligence Community Research and Development* (2006).

Scalable Parallel Topic Models

David Newman[1], Padhraic Smyth[1], Mark Steyvers[2]
[1]Department of Computer Science, University of California Irvine
[2]Department of Cognitive Sciences, University of California Irvine

Abstract

(U) The topic model is a popular probabilistic model for text and document modeling. It can be used for topic indexing, document classification, corpus summarization and information retrieval. In the past, topic models have been applied to corpora containing thousands to hundreds of thousands of documents. Now there is an increasing need to model collections with millions to billions of documents. We present a parallel algorithm for the topic model that has linear speedup and high parallel efficiency for shared-memory symmetric multiprocessors (SMPs). Using this parallel algorithm, topic model computations on an 8-processor system took 1/7 the time of the same computation on a single processor.

1 Introduction

(U) The exponential growth in the creation and traffic of text objects continues. Nowhere is this more apparent than in the intelligence community, which processes high volumes of text information on a daily basis. Often these pieces of text information are untagged, uncategorized and unstructured. One can access and query these text objects with keyword search, but this doesn't help, say, an analyst organize, categorize and browse huge numbers of text objects. The topic model is an unsupervised probabilistic language model that automatically learns the topics or subjects contained in a collection of text documents, and through these topics, the topic model can automatically organize, categorize and make browsable huge numbers of documents. The topic model is extremely robust and versatile: there is no need for domain knowledge, labeled training examples, *a priori* category definitions, or even knowledge of the language. Furthermore, the topic model has been shown to work well on a wide variety of text collections, from emails, to news articles, to research literature abstracts.

(U) While the time and space complexity of the topic model scales linearly with the number of documents in a collection (a scaling that is already favorable), computations are only practical for modest-sized collections of up to hundreds of thousands of documents. With today's need to handle and process collections that are orders of magnitude larger, scalable and parallel topic model algorithms are needed.

(U) There has been no published work, to our knowledge, on parallel implementations of the topic model, which uses a statistical Markov chain Monte Carlo algorithm to learn the topics. For a more general reference on parallel methods and techniques in computational statistics see [Kontoghiorghes, 2006], which devotes a chapter to Markov chain Monte Carlo methods in general. Practical aspects of Markov chain Monte Carlo methods are discussed in [Gilks, 1996].

2 Overview of Topic Model Algorithm

(U) The topic model is a statistical language model that relates words and documents through topics. It is based on the idea that documents are made up of a mixture of topics, where topics are distributions over words. Specifically, the topic model is based on the latent Dirichlet allocation (LDA) model, which has become a popular model for discrete data, such as collections of text documents [Blei et al. 2003]. While Blei et al. proposed approximate variational Bayesian methods to solve LDA, Griffiths and Steyvers proposed Gibbs sampling for inference in LDA [Griffiths & Steyvers, 2004]. Griffiths and Steyvers reported that Gibbs sampling led to equivalent accuracy faster than alternative methods. Gibbs sampling, a Markov chain Monte Carlo method, is highly attractive because it is simple, fast, and has very few adjustable parameters.

(U) A key feature of the topic model is that it is an unsupervised learning technique, which means that the often human-intensive task of finding labeled examples is completely eliminated. *Unsupervised* also means that one can topic model a collection without being a domain expert – in fact one can even topic model collections in other languages, without needing to know much about the language. The topic model probabilistically figures out groups of words that tend to co-occur, and identifies these groups as semantic topics. These learned topics are useful for a variety of tasks, from automatically summarizing a document collection, to topically indexing individual documents, to information retrieval. For a review of topic modeling theory and applications, see [Newman et al., 2006] and [Steyvers & Griffiths, 2006].

(U) Because of the broad versatility and usefulness of the Gibbs sampled topic model, and the need to model larger collections, we explore in this paper scalable parallel versions of this algorithm. We first take the reader through the analysis of the sequential algorithm and an example topic model computation before proposing scalable parallel versions of the algorithm in Section 3.

2.1 Analysis of Sequential Algorithm

(U) In this section we review the Gibbs sampler for the topic model. Table 1 introduces key dimensions or size parameters that describe a corpus (*D*, *W*, *N* and *L*) and a topic model run (*T* and *ITER*), while Table 2 shows the arrays used in the topic model, and their sizes.

param.	Description
D	Number of documents in corpus
W	Number of words in vocabulary
N	Total number of words in corpus
L	Average length of document in words ($L = N/D$)
T	Number of topics
ITER	Number of iterations of Gibbs sampler

Table 1. Dimensions used in topic model.

array	Description
wid(N)	Word ID of nth word
did(N)	Document ID of nth word
z(N)	Topic assignment to nth word
Cwt(W,T)	Count of word *w* in topic *t*
Ctd(T,D)	Count of topic *t* in document *d*
Ct(T)	Count of topic *t*

Table 2. Arrays used in topic model.

(U) Since our purpose is to analyze the sampling equation and code, we state, without derivation, the Gibbs sampling equation for the topic model (see [Griffiths & Steyvers, 2004] for further details). The topic model expresses the idea that words and documents are connected through topics. Equation (1) shows that the likelihood of a particular word in a particular document depends on the likelihood of that word in all the topics weighted by the likelihood of all the topics in that document. Following this idea, one can write down a generative process, and formally derive the Gibbs sampling equation (2). This expression gives the probability that the nth word in the corpus belongs to topic *t*. In this equation we explicitly see the various count arrays, including *Cwt*, *Ct* and *Ctd*. The '$-n$' superscript is standard sampling notation to indicate that the current word (word n) has been removed from these counts. Here, α and β are Dirichlet priors, and can be informally interpreted as pseudo-counts or smoothing parameters.

$$P(w \mid d) = \sum_{t=1}^{T} P(w \mid t) P(t \mid d) \qquad (1)$$

$$P(z_n = t \mid z_{-n}) \propto \frac{C_{wt}^{-n} + \beta}{C_t^{-n} + W\beta} \left(C_{td}^{-n} + \alpha \right) \qquad (2)$$

```
1   for iter = 1:ITER
2     for n = 1:N
3       t = z(n)                                // get current topic assignment
4       Ctd(t,did(n))--, Cwt(wid(n),t)--, Ct(t)--                 // decrement
5       for t = 1:T
6         P(t) = (Cwt(wid(n),t)+β)(Ctd(t,did(n))+α)/(Ct(t)+Wβ)   // equation(2)
7       end
8       sample t from P(t)
9       z(i) = t                                     // set new topic assignment
10      Ctd(t,did(n))++, Cwt(wid(n),t)++, Ct(t)++                 // increment
11    end
12  end
```

Table 3. Pseudo-code for topic model Gibbs sampler.

(U) The algorithm for the topic model Gibbs sampler, shown in Table 3, embeds equation (2) in a loop over *T* topics, *N* words in the corpus, and *ITER* sweeps of the Gibbs sampler. After a sufficient number of iterations, the count arrays *Cwt* and *Ctd* hold the data to estimate the distributions over words in each topic, *P(w|t)*, and over topics in each document, *P(t|d)*. By inspection of the pseudo-code, we can write down time and space complexity of the topic model Gibbs sampler.

$$\text{Time} \sim O(\,ITER \cdot N \cdot T\,) \qquad (3)$$

$$\text{Space} \sim O(\,3\,N + (D + W)T\,) \qquad (4a)$$

For large corpora, D >> W, and N = D · L, so

$$\text{Space} \sim O(\,(3\,L + T)D\,) \qquad (4b)$$

(U) To appreciate this time and space complexity, consider a million-document corpus with the following size parameters: D=10^6, W=10^4, and L=10^3 (so $N = DL = 10^9$). For this corpus, it would be reasonable to run with T=10^3 topics and $ITER$=10^3 iterations. Assuming a single 3 GHz processor, and storing counts as 4-Byte integers, we can estimate that running the topic model would take 10^{15} evaluations of equation (2), line 6 in Table 3, or 30 days[1]. Furthermore, using equation (4b), the required memory would be $(3 \cdot 10^3 + 10^3)\, 10^6\, 4$ = 16 GByte. This memory requirement is beyond most desktop computers, and the length of time makes this topic model computation impractical for many purposes. We will see in Section 3 how to do this computation in less time or with less memory.

2.2 Example Topic Model Computation

(U) To make concrete the concept of topic modeling, in this section we step through an example using the CiteSeer collection of scientific literature abstracts (citeseer.ist.psu.edu). We harvested from CiteSeer 530,000 abstracts spanning more than a decade. After preprocessing, stopword removal, and ignoring any words that occurred in fewer than ten abstracts, we produced a vocabulary of W = 50,000 words. Using this vocabulary, we produced a document-by-word matrix with a total of N = 35,000,000 words. For the topic-model run in this example, we set the number of topics T = 100, and the number of Gibbs sampler iterations $ITER$ = 200. Table 4 shows three selected topics out of the 100-topic run. These topics show the 17 most likely words in the topic, and their probability *P(w|t)*. The title above each box is a human-assigned label (automatically assigning labels to topics is a current

[1] Assuming 5 floating-point operations for equation (2), and 2 Gflop processing capacity.

research challenge). Note how each list of likely words captures a theme that is easily interpreted as a particular topic or subject area. Also note how words in the vocabulary can belong to multiple topics, for example the topics of Probabilistic Models and Numerical Solution to PDEs both have the important concepts around the words 'method' and 'error'. We re-iterate that these topics are *learned* by the topic model – there is no need for any domain knowledge or *a priori* definitions of topics. These topics could be used, for example, to find the most likely documents relating to Parallel Computing, Probabilistic Models or Numerical Solution to PDEs, since during the learning algorithm described in the previous sections, all words in all documents are assigned to topics.

Parallel Computing		**Probabilistic Models**		**Numerical Solution to PDEs**	
parallel	0.067	model	0.047	equation	0.072
processor	0.026	estimation	0.033	solution	0.046
communication	0.024	parameter	0.024	method	0.037
performance	0.024	estimator	0.018	differential_equation	0.013
application	0.015	distribution	0.018	numerical	0.013
implementation	0.013	data	0.017	finite_element	0.011
computation	0.012	regression	0.016	nonlinear	0.009
parallelism	0.012	method	0.015	diffusion	0.009
cluster	0.011	bayesian	0.015	partial	0.009
high_performance	0.011	estimate	0.014	discretization	0.008
workstation	0.011	statistical	0.013	boundary	0.008
multiprocessor	0.010	error	0.013	order	0.008
shared_memory	0.010	estimates	0.012	scheme	0.008
system	0.010	sample	0.010	approximation	0.007
machine	0.010	variance	0.009	convergence	0.007
distributed_memory	0.009	prior	0.009	linear	0.007
message_passing	0.009	density	0.008	error	0.007

Table 4. Selected topics from T=100 topic model run of CiteSeer data set.

(U) Even though Gibbs sampling is an iterative procedure, there are no universally agreed-upon formal convergence criteria. This problem, however, tends to be more theoretical than practical, and there are a variety of measures one can use as a guide for terminating the Gibbs sampler. A straightforward measure is the in-sample perplexity, equation (5), which has the advantage of not requiring an out-of-sample or test data set. The perplexity, a standard measure in language modeling, is the inverse of the geometric mean per-word likelihood. Perplexity varies from 1 to W; lower perplexity is better, and the maximum perplexity of W is reached when all words in the vocabulary are equally likely.

$$pplex = \exp\left(-\frac{1}{N}\sum_{n=1}^{N}\log P(w_n \mid d_n)\right), \qquad (5)$$

where $P(w_n|d_n)$ is the probability assigned by the model to a word w_n, after summing over topics as in equation (1).

(U) We plot in Figure 1 in-sample perplexity versus iteration for the *T*=100 topic run on the CiteSeer data set. At iter=0, the perplexity is 4550, which is the perplexity under the unigram model (where the words in every document are drawn independently from a single multinomial distribution). Over the next 80 iterations of the Gibbs sampler, we see the perplexity decrease rapidly and settle around a value of approximately *pplex*=1315. During this initial "burn-in" period our parameter estimates are converging to the true or target distribution. While it does not occur in this figure, there is no guarantee that perplexity monotonically decreases. In practice, however, any samples collected after the burn-in period can be used to compute the word-topic and topic-document distributions.

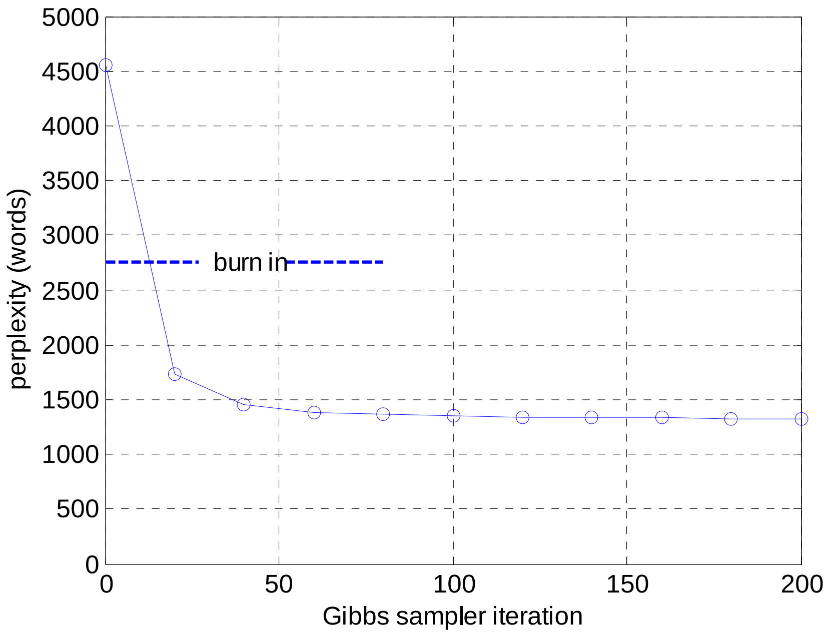

Figure 1. Perplexity versus iteration for T=100 topic run of CiteSeer data set, illustrating the concept of burn-in.

2.3 Evaluation Metrics

(U) Before moving on to parallel and scalable implementations of the topic model Gibbs sampler, we define evaluation metrics that we will use to establish correctness of the parallel implementation. Establishing correctness of code for Gibbs samplers can be difficult because runs that are started with different seeds (and different initial conditions) produce different results. In particular, the topic IDs (that range from 1 to T) are arbitrarily labels, i.e. they are exchangeable (this property is referred to as lack of identifiability). For example, the Parallel Computing topic might be topic-31 in one run, and topic-82 in another. When computing quantities that sum over topics, e.g. *P(w|d)*, one can ignore the topic IDs and, beyond that, average over multiple runs or samples to improve accuracy and reduce variance. However, if the topics themselves are of direct interest, we need a way to match the set of topics from one run to the set of topics from another run. A straightforward way of doing this is to choose the minimum bipartite matching that minimizes the average KL divergence between topics in run1 and topics in run2. We write this function as:

$$B(run1, run2) = \frac{1}{T}\sum_{t=1}^{T} KL^{*}\left[P_1(w\,|\,t)\middle\|P_2(w\,|\,perm(t))\right] \qquad (6a)$$

$$KL^{*}\left(P\middle\|Q\right) = \frac{1}{2}\sum_{w=1}^{W}\left(P_w \log_2\frac{P_w}{Q_w} + Q_w \log_2\frac{Q_w}{P_w}\right) \qquad (6b)$$

where KL^{*} is the symmetrized KL divergence (6b), and perm(t) is the permutation that minimizes B (6a). Bipartite minimum matching is carried out using the Kuhn-Munkres algorithm, also known as the Hungarian method. In the remainder of this paper, we will use two metrics to assess correctness of code. The first metric is *pplex*, in-sample perplexity (described in the previous section) and the second metric is *B*, the best-matched average KL divergence.

(U) To demonstrate the variation between topic model runs started with different seeds, we ran the topic model multiple times with different seeds on the CiteSeer data set, with T=100 topics. We'll call two of these runs run1 and run2. We computed the permutation matrix (mapping topic IDs in run1 to topic IDs in run2) that minimized the average of the 100 KL divergences. While the average KL divergence between topics in run1 and topics in run2 was 2.8 bits, individual KL divergences varied from less than one bit to more than 10 bits. Table 5 shows selected matched topics. The first row shows a closely matched topic relating to parallel computing, differing only by 0.33 bits. The second row shows an average matched topic (at 2.53 bits). We see an obvious topical correspondence between run1 and run2, with run1 having a game/player spin on this topic relating to market/risk/return. The third row shows a more distant matching (at 7.31 bits). In this case we see two different topics related to visualization. Not shown here are the sizes of each topic; we generally observe that the larger topics (e.g. the topic of parallel computing) tend to have a smaller KL divergence when matched between runs.

(U) It is important to point out that, in this example, the topics from run1 and run2 are equally valid, assuming that the in-sample perplexities of run1 and run2 are reasonably similar.

Most likely words in run1 topic	run1 topic id	KL (bits)	run2 topic id	Most likely words in run2 topic
parallel processor communication performance application implementation computation parallelism cluster high_performance workstation multiprocessor shared_memory system machine distributed_memory message_passing	[t93]	0.33	[T77]	parallel communication performance processor application implementation parallelism cluster computation high_performance workstation shared_memory machine multiprocessor distributed_memory machines
market game risk games price model player option return stock auction equilibrium pricing prices volatility asset financial trading exchange economic	[t58]	2.53	[t65]	market risk price financial option firm return stock bank auction cost economic pricing rate model prices exchange policy asset volatility
tool user system visualization interactive environment application graphical interface visual user_interface graphic window support display software computer animation toolkit	[t42]	7.31	[t46]	visualization rendering animation graphic volume interactive display method techniques light ray object algorithm technique view computer_graphic scene image visual tracing

Table 5. Sample matched topics from two Gibbs sampler runs started with different initial random seeds. Average KL distance between matched topics is 2.7 bits, so the table shows examples of a closely matched, average matched, and more distant matched pair of topics.

(U) Extending the above example, we compare topic distributions from four T=100 runs with random seeds = 1, 3, 5 and 7. Histograms of the matched KL divergences for the six run-pairs are shown in Figure 2. The six histograms show similar distributions: every one of them has more than 50 matched topics diverging by less than 2 bits, and the means and standard deviations of the KL divergence are reasonably consistent. Furthermore, the perplexities of the four runs are very close, varying at most by 2.5 words or 0.2%.

(U) Now that we can measure the variability between runs (using the sequential code), we have a framework for assessing whether a run produced by a different (e.g. parallelized) code is within this variability, and therefore a valid implementation of the topic model Gibbs sampling algorithm.

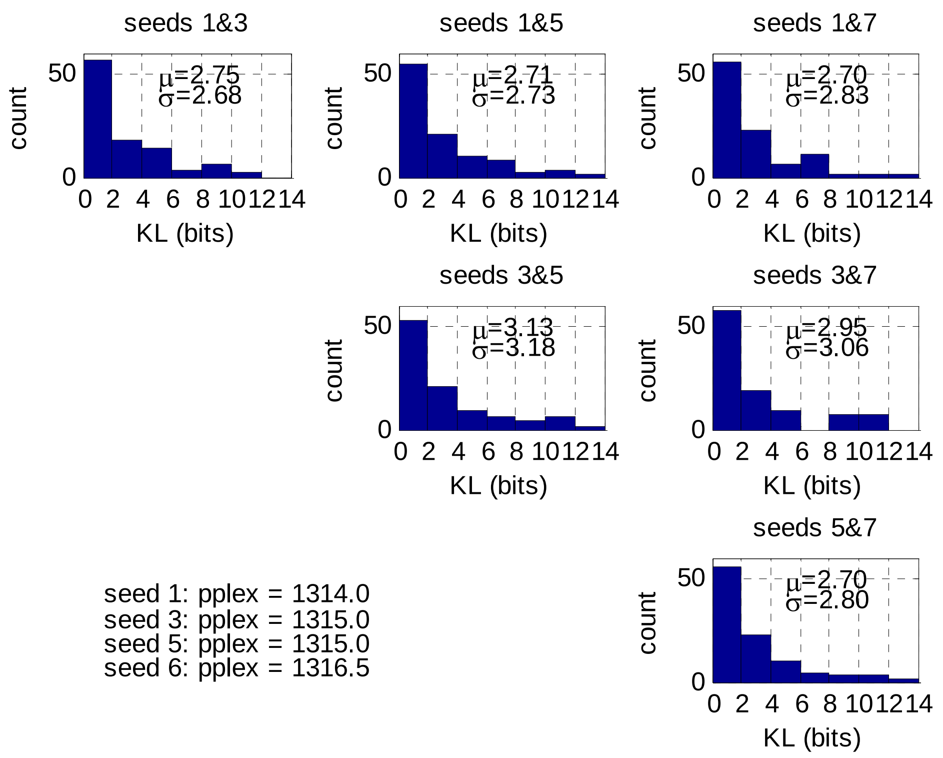

Figure 2. Histogram of matched KL divergences between runs with seed = 1, 3, 5 and 7.

2.4 Data Sets

(U) We used two data sets in this study, CiteSeer and New York Times. We used the majority of the entire CiteSeer Scientific Literature Digital Library, available at citeseer.ist.psu.edu. The New York Times dataset includes New York Times and other urban and regional news articles, and can be obtained from the Linguistic Data Consortium, ldc.upenn.edu. Size parameters for these two data sets are shown in Table 6.

	CiteSeer	New York Times
D	520,000	330,000
W	50,000	100,000
N	35,000,000	110,000,000
L	70	330

Table 6. Size parameters of data sets.

3 Parallel and Scalable Implementations of Topic Model Algorithm

(U) In this study we only considered parallel implementations of the topic model for shared-memory processors. We did this for several reasons. First, practically all modern supercomputers are made up of collections of n-way shared memory processors (also known as SMPs or Symmetric MultiProcessors), so any code parallelization on these larger machines builds on code parallelization for a single multiprocessor node[2]. Second, SMP processors with up to 32 processors (sometimes called '32-way') and 64 GB of shared memory are becoming commodity items, providing substantial processing power for large topic model computations. Third, the programming model for distributed memory machines is a fundamentally different paradigm. Distributed memory machines typically use

[2] See www.top500.org for a list of the world's 500 fastest supercomputers.

MPI, the Message Passing Interface. Message-passing is more difficult to program and doesn't support incremental parallelization of an existing sequential program.

(U) We parallelized our topic model code using OpenMP[3], the current standard for shared-memory parallel programming. One advantage of using OpenMP is that sequential code can be parallelized easily and incrementally simply by using compiler directives. When designing parallel code, one aims to maximize parallel efficiency $\varepsilon = t_1 / (P \cdot t_P)$, where t_1 is the execution time on one processor, and t_p is the execution time on P processors. This is usually achieved by identifying the largest possible chunk of calculations that can be performed in parallel, which amortizes overhead of thread creation and synchronization (and the overhead of communication in MPI programs). The basic OpenMP model is to create work-sharing constructs where multiple threads divide execution of the enclosed code region. When a thread reaches a parallel directive, it creates a team of threads. The code is duplicated and all threads execute the code. Parallelizing over for-loops is typically the best place to start given a sequential algorithm. Note that in this discussion, we will use the terms 'thread' and 'processor' synonymously.

3.1 Parallelization over T topics

(U) The simplest parallelization of the topic-model Gibbs sampler is computing P(t) in parallel (pseudo-code lines 5-7, Table 3), i.e. parallelizing the `for t=1:T` loop. While this calculation can clearly be done in parallel, even with a large number of topics, *T*, there will not be sufficient floating point operations to amortize the relatively time consuming task of creating, synchronizing and joining threads. While this fixed cost of creating/synchronizing/joining threads varies depending on architecture, operating system, and compiler, it can be larger than one million floating point operations. One needs to be doing orders or magnitude more than one million floating point operations before OMP parallelization is worthwhile. Given that ~1000 is a large number of topics for a topic model run, this strategy for parallelization will clearly not work, and in fact, parallelizing over this `for` loop will result in a significant slowdown.

3.2 Exact Parallelization over N words

(U) The next obvious opportunity for work-sharing is to parallelize over the words in the corpus, i.e. divide the `for n=1:N` loop (pseudo-code lines 2-11) into P `for` loops of size *N/P*. To analyze this approach, we first examine data layout and code execution with a single thread, shown in Figure 3a. Observe that the corpus is typically laid out document-by-document, thus *did(n)* monotonically increases from 1 to D. This is not true for *wid(n)*, because any document in the corpus can contain any word (e.g. in Figure 3a, *wid(6)=wid(n)*, i.e. the word 'two'). Now consider the case of P threads working in parallel, as shown in Figure 3b. Each thread works on a `for` loop of size *N/P*, illustrated by the double vertical line in the figure.

n	1	2	3	4	5	6									n									...	N
did	1	1	1	1	1	1	1	...			...	d	d	d	d	d	d	...				...	D	D	D
word(wid)		thing	one	and	thing	two							one	fish	two	fish	red								
z															t										

Figure 3a. Schematic of data layout for Gibbs sampling for single thread. Gray region illustrates current value of n in for loop.

n	1	2	3	4		n_0									n_1									...	N
did	1	1	1	1	1	d_0	1	...			...	d	d	d	d_1	d	d	...		...		...	D	D	D
word(wid)		thing	one	and	thing	w_0							one	fish	w_1	fish	red								
z						t_0									t_1										

Figure 3b. Schematic of data layout for Gibbs sampling for P threads. Each thread works on a portion of the data delimited by the double vertical lines. Gray region illustrates current value of n in for loop for threads 0 and 1.

[3] See www.openmp.org "OpenMP is a specification for a set of compiler directives, library routines, and environment variables that can be used to specify shared memory parallelism in Fortran and C/C++ programs."

(U) While multiple threads can simultaneously *read* from a memory location, one should generally not let multiple threads simultaneously *write* to a memory location[4]. The only variables being written to are *z*, *Ctd*, *Cwt* and *Ct*. Taking these variables one at a time, we see that *z* is in effect private to each thread, and elements in the *Ctd* array will never be written to simultaneously. However, there is a small chance that elements in the *Cwt* array could be written to simultaneously (e.g. if $w_0=w_1$ and $t_0=t_1$). Furthermore, there is a large chance that elements in the one-dimensional array *Ct* will be frequently written to simultaneously (i.e. any time $t_0=t_1$). We make the *P(T)* variable (line 6 in the pseudo-code) private to each thread. We can resolve the indetermininacy of simultaneous writes by using OMP's atomic directive, which enforces atomic updates to a specific memory location. Armed with this atomic device, we can create an OpenMP version of the Gibbs sampler that exactly implements the Gibbs sampling equations, by first dividing the problem using OMP's *parallel* and *sections*, and second ensuring correct counting using *atomic*. The schematic of the parallelized pseudo-code and detail of the atomic directives is shown in Table 7. Note that this will produce an exact parallelization of the Gibbs sampled topic model (i.e. the semantics of the code will be unchanged), because of exchangeability of words under latent Dirichlet allocation, i.e. the Gibbs sampler doesn't care in which order topics are assigned to words.

```
for iter = 1:ITER

#pragma omp parallel sections shared(wid,did,z,Ctd,Cwt,Ct) private(n,t,Prob)
{
#pragma omp section
{
  p = omp_get_thread_num();
  for n = pN/P:(p+1)N/P
    ...
#pragma omp atomic
    Cwt(wid(n),t)--
#pragma omp atomic
    Ct(t)--
    ...
#pragma omp atomic
    Cwt(wid(n),t)++
#pragma omp atomic
    Ct(t)++
    ...
  end
}
... repeat above section block P times
}

end     //over iter
```

Table 7. Pseudo-code fragments for exact parallelization using atomic updates for thread p of P.

3.3 Nearly-Exact Parallelization over N words

(U) While it is satisfying to have exactly parallelized our Gibbs sampler, our primary goal is speedup, so we further examine the use of atomic updates. We anticipate that atomic updates, which serialize threads, may have an adverse effect on speedup, and may not scale well, meaning parallel efficiency will decrease with increasing number of threads. If we omit the atomic directives in Table 7, we may occasionally have counts that are dropped or added. If counts are dropped during a thread-simultaneous increment operation (++), or added during a thread-simultaneous decrement operation (--), then we will not be implementing the Gibbs sampler exactly. However, the result may be very close, if not indistinguishable, from the exact parallelization. Furthermore, we can periodically correct or settle any count discrepancies, which will eliminate the problem of error build-up. Periodic correction or settlement is straightforward: the topic assignment variable, *z*, along with *wid* and *did* can be use to reconstruct the count arrays *Cwt* and *Ct* (and incidentally *Ctd*, which never needs to be reconstructed).

[4] If two threads encounter a LOAD/ADD/STORE sequence of instructions for a specific memory location, the result is indeterminate. For example, if x=0, and x=x+1 is executed concurrently, the final result may be x=1 or 2.

(U) We can estimate within an order-of-magnitude the rate of write collisions to the *Ct* and *Cwt* arrays. The rate of collisions is analogous to the probability of two people in a large group sharing a birthday. With the *Cwt* array, a collision will potentially occur if two threads are writing to the same element of *Cwt*, i.e. both threads have the same wid and t (in Figure 3b, when w_0=w_1 and t_0=t_1). When topic modeling, the highest frequency words ('the', 'and') are removed in preprocessing as stopwords. This means that the highest frequency word in the remaining vocabulary may occur at about 1 in 500 words. With 8 threads, the probability that two threads are at the same word is 0.04. Multiplying this by the probability of being at this most likely word (1/500), and the probability that the topics are the same, results in a relatively small probability. With the *Ct* array, collisions will be far more frequent (any time t_0=t_1). However, the average value of elements in *Ct* is *N/T* (10^6 for large topic model runs), so even losing or adding tens of counts will have negligible effect. Finally, errors from the -- and ++ operations will tend to cancel, and we ultimately expect the net effect or sum of errors be zero over the longer term, i.e. there will be minimal drift. One could analyze this indeterminacy more completely by modeling dropped or added counts as a noise process on the Markov chain. Finally note that it is impossible to produce negative counts as a result of collisions. Again, all these problems are mitigated by the periodic correction or settlement of *Cwt* and *Ct* counts.

(U) Given this possibility of *not* using the atomic directive, we propose an alternative parallelization to the Exact Parallelization described in Section 3.2, and call it the Nearly-Exact Parallelization. The code differs from that in Section 3.2 by the *omission* of the atomic directives, and the inclusion of a periodic call to reconstruct the *Cwt* and *Ct* count arrays. The pseudo-code for the Nearly-Exact Parallelization is given in Table 8.

```
for iter = 1:ITER

#pragma omp parallel sections shared(wid,did,z,Ctd,Cwt,Ct) private(n,t,Prob)
{
#pragma omp section
{
  p = omp_get_thread_num();
  for n = pN/P:(p+1)N/P
    t = z(n)
    Ctd(t,did(n))--, Cwt(wid(n),t)--, Ct(t)--
    for t = 1:T
      Prob(t) = (Cwt(wid(n),t)+β)(Ctd(t,did(n))+α)/(Ct(t)+Wβ)
    end
    sample t from P(t)
    z(i) = t
    Ctd(t,did(n))++, Cwt(wid(n),t)++, Ct(t)++
  end
}
... repeat above section block P times
}

if (iter mod 100 == 0) then recompute Cwt and Ct from z

end     //over iter
```

Table 8. Pseudo-code for the nearly-exact parallelization of the topic model Gibbs sampler running on P processors.

3.4 Speedup Results

(U) We ran a series of experiments to measure speedup, defined as execution time on 1 processor divided by execution time on P processors, t_1/t_P. Both the CiteSeer and New York Times data sets were used in the experiments, and multiple runs were conducted to determine variability of execution times. The computations were run on an IBM p655 8-way node (i.e. 8 processors) with 16 GB memory, AIX operating system and the OMP version of our topic model C-code compiled with IBM's xlC compiler. Figure 4 shows the speedup results running on 1, 2, 4 and 8 processors (threads). As suspected, the exact parallelization of the topic model code using atomic updates of *Cwt* and *Ct* greatly reduces parallel efficiency, with 4 processors only being 1.5 times faster than one processor. However the nearly-exact parallelization of the topic model code – with no atomic updates – performs very well, achieving linear speedup (over 2-8 processors), and a parallel efficiency of 72%. As is often the case, this parallel efficiency depends on problem size. When we ran a larger version of the problem by increasing number of topics *T* from 100 to 1000, the parallel efficiency increased from 72% to 88%. Variability of execution times was negligible for repeated runs.

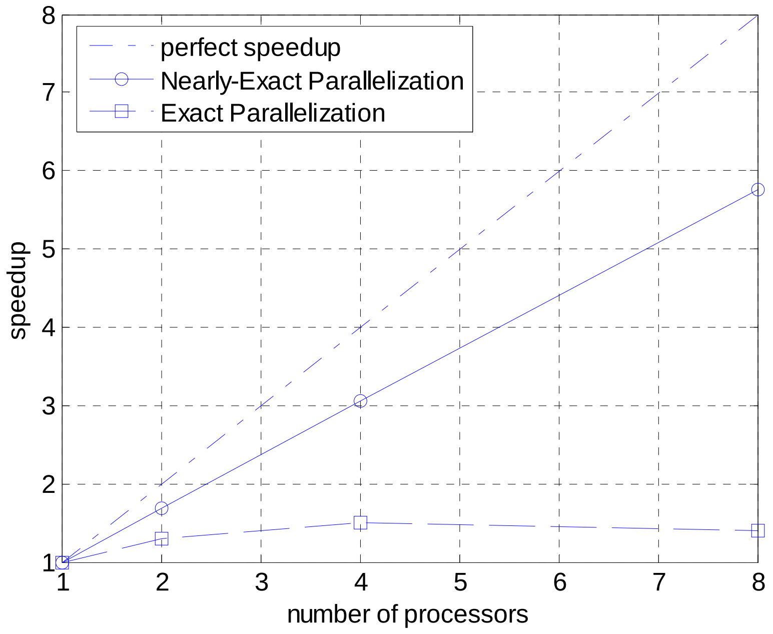

Figure 4. Speedup of topic model Gibbs sampler from 1 to 8 processors.

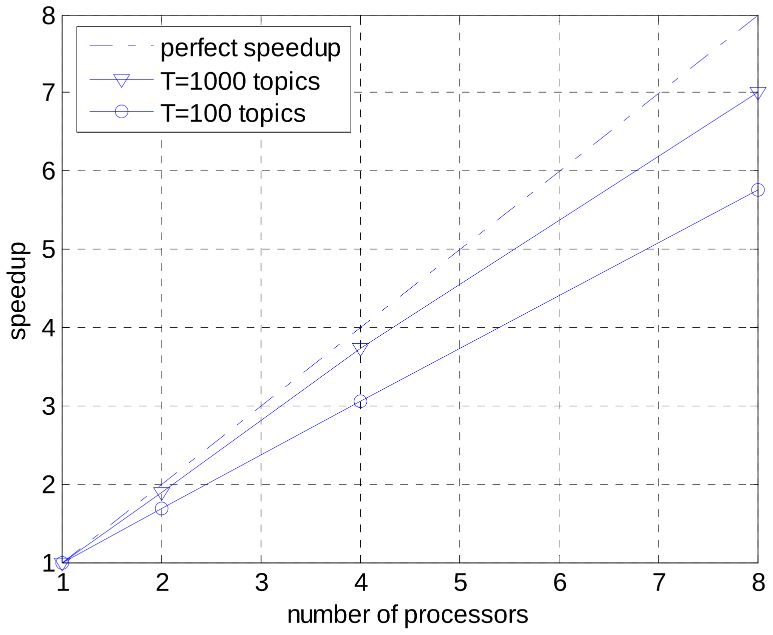

Figure 5. Speedup curve showing parallel efficiency increasing from 72% to 88% with increasing problem size.

(U) Recall that the exact parallelization (with atomic updates) exactly implements the Gibbs sampling algorithm, but the nearly-exact parallelization (without atomic updates) does not. The exact parallelization is not useful in practice because of its poor speedup, but the nearly-exact parallelization has very good speedup. While our analysis of the lack of atomicity indicates that the error should be small, we verify that this code is producing the same statistics by measuring perplexities, *pplex*, and average run-run KL divergences, *B*. Table 9 shows the mean and standard deviation of perplexity for 10 runs of the sequential code, and 10 runs of the nearly-exact parallelization. These statistics indicate that the parallel code is no worse, perplexity wise, than the sequential code, and therefore that the nearly-exact parallelization is valid. Recall that, for runs computed with the sequential code, average run-run KL distances varied from *B* = 2.7 to 3.1 bits.

pplex (sequential) = (μ = 1316.3, σ = 2.2) words
pplex (parallel) = (μ =1315.1, σ=2.8) words
B(sequential,parallel) = (μ =2.99, σ=2.92) bits

Table 9. Statistics show that parallel code produces same results as sequential code.

3.5 Scalability of Memory

(U) Scalability implies scalability of computation time *and* scalability of required memory. Recall from Section 2.1, Equation (4a), that required memory scales as $3N+(D+W)T$. This required memory can be reduced by using sparse storage for the count arrays, *Cwt* and *Ctd*, which typically contain only 5-10% nonzero entries, i.e. they are 90-95% sparse. The memory for a topic model run with a huge number of documents and a large number of topics will be dominated by the *Ctd* count array, so using sparse storage will greatly reduce required memory. But this reduction in required memory comes at the cost of increased computation time. If, on average, a hash table requires x operations to set or get (key,value) pairs, overall computation time will increase by a factor of x. As is often the case, there is a tradeoff between memory and speed. For this study we were mostly interested in speedup, because we were not currently limited by memory.

4 Conclusion

(U) In this paper we addressed scalability of the topic model Gibbs sampler. After outlining the sequential algorithm, we show how the time and space complexity is linear in the number of documents in the collection. Scalability in time can be achieved through parallelization. We present an OpenMP parallelization of the algorithm that exhibits linear speedup with parallel efficiencies from 72% to 88%. For example, topic model computations on an 8-processor system took 1/7 the time of the same computation on a single processor. For sufficiently large problems, we could, for example, complete a 4-week topic model computation in 1 day. Scalability in memory can be achieved through hashed associative arrays by taking advantage of the sparseness of the count arrays.

Acknowledgements

(U) This work was supported by the National Science Foundation under award number IIS-0083489 (as part of the Knowledge Discovery and Dissemination Program) and under award number ITR-0331707. Computations were performed on UC Irvine's ESMF computer.

5 References

[Blei et al., 2003] D. Blei, A. Ng, M. Jordan. Latent Dirichlet Allocation. Journal of Machine Learning Research 2 993-1022. (2003).

[Griffiths & Steyvers, 2004] T. Griffiths and M. Steyvers. Finding Scientific Topics. Proceedings of the National Academy of Sciences 101:5228-5235. (2004).

[Newman et al., 2006] D. Newman, C. Chemudugunta, P. Smyth, M. Steyvers. Analyzing Entities and Topics in News Articles using Statistical Topic Models. LNCS 3975, Intelligence and Security Informatics. Springer. (2006).

[Steyvers & Griffiths, 2006] M. Steyvers and T. Griffiths. Probabilistic Topic Models. In Latent Semantic Analysis: A Road to Meaning. Laurence Erlbaum. (2006).

[Kontonghiorghes, 2006] Erricos John Kontonghiorghes (Ed). Handbook of Parallel Computing and Statistics. Chapman & Hall/CRC. (2006).

[Gilks et al., 1996] W. Gilks, S. Richardson, D. Spiegelhalter (Eds). Markov Chain Monte Carlo in Practice. Chapman & Hall/CRC. (1996).