Parallelization and Characterization of Probabilistic Latent Semantic Analysis

Chuntao Hong, Wenguang Chen, Weimin Zheng
Tsinghua University, China, 100084
hct05@mails.thu.edu.cn, cwg@thu.edu.cn, zwm-tse@thu.edu.cn

Jiulong Shan[*], Yurong Chen, Yimin Zhang
Intel China Research Center
{jiulong.Shan, yurong.chen, yimin.zhang}@intel.com

Abstract

Probabilistic Latent Semantic Analysis (PLSA) is one of the most popular statistical techniques for the analysis of two-model and co-occurrence data. It has applications in information retrieval and filtering, nature language processing, machine learning from text, and other related areas. However, PLSA is rarely applied to large datasets due to its high computational complexity.

This paper presents an optimized and parallelized implementation of PLSA which is capable of processing datasets with 10000 documents in seconds. Compared to the baseline program, our parallelized program can achieve speedup of more than six on an eight-processor machine. The characterization of the parallel program is also presented. The performance analysis of the parallel program indicates that this program is memory intensive and the limited memory bandwidth is the bottleneck for better speedup.

Keywords: parallelization, characterization, PLSA, tempered EM, multi-core

1. Introduction

Learning from text and natural language is an important part of machine learning, which has been drawing more and more attention in the computer science community. However, learning from text and nature language still remains one of the most challenging fields in computer science. The main challenge such a machine learning system has to deal with when learning roots in the problem of polysemys and synonymys, i.e., a word having several meanings and several words having the same meaning.

Latent Semantic Analysis (LSA)[2] is a well-known technique developed to solve the problem of polysemys and synonymys. Traditionally, documents are viewed as vectors of words, which then make up a large document-word co-occurrence table. LSA, on the other hand, tries to map the vectors into a lower-dimension space, called latent semantic space. By doing this, LSA avoids the problem of polysemys and synonymys. Due to its generality, LSA has been applied to a wide range of fields.[2][6][7][8] Despite this, the theoretical foundation of LSA remains unsatisfactory and incomplete.[1]

Probabilistic Latent Semantic Analysis (PLSA)[1] is proposed as an alternative of LSA. In contrast to LSA, which performs Singular Value Decomposition (SVD) on the co-occurrence tables, PLSA is based on mixture decomposition derived from a latent class model, which results in a more principle approach that has a solid statistical foundation. However, PLSA is rarely applied to large datasets because of its high computational complexity.

This paper presents an optimized and parallelized implementation of PLSA, which is able to achieve speedup of more than six on an eight-processor shared-memory machine. The baseline program is an optimized Tempered Expectation-Maximization (TEM) implementation of PLSA, which is stemmed from the implementation included in Lemur[5]. The original implementation in Lemur is not well optimized, but we performed several common optimizing techniques, such as loop interchange and unrolling, to the program, and made it possible to run the program on datasets with more than 10000 documents.

Considering the multi-core trend in future CPUs, we implemented our parallel program using the OpenMP[3] programming model, which is the most widely used programming model for shared-memory

[*] Jiulong is now working in Google China.

0190-3918/08 $25.00 © 2008 IEEE
DOI 10.1109/ICPP.2008.8

machines. During the parallelization, we found that our datasets are sparse matrices whose nonzero elements are not evenly distributed, which incurred load imbalance among the processors. We propose a block dividing algorithm and a block scheduling algorithm to achieve better load balance. According to our experiments, the two algorithms are both effective and of low cost.

Experiments were also carried out to identify the characteristics of the program. The performance analysis of the parallel implementation indicates that this program is memory intensive and the limited memory bandwidth is the bottleneck for better speedup.

The main contributions of this paper include:

1. An efficient parallel TEM implementation of PLSA based on the implementation provided in Lemur[5], which uses the same memory size, but is much faster than Lemur implementation, and scales well on our multi-core machines.

2. An adaptive block dividing algorithm and a block scheduling algorithm to achieve load balance for the parallel implementation of PLSA, which prove to be both effective and of low cost.

3. Analysis on the parallel implementation which indicates that the limited memory bandwidth has become the bottleneck of this kind of memory intensive program.

The remaining sections of this article are organized as follows. Section 2 gives brief introduction to PLSA and the tempered EM algorithm. Section 3 demonstrates how we optimize the Lemur code to get our baseline program. Section 4, then, presents the parallelization of the program, in which we propose a data dividing and a job scheduling algorithm. Section 5 exhibits the experimental results, and gives the analysis, which leads to the conclusions part.

2. Probabilistic Latent Semantic Analysis and Tempered EM Algorithm

Suppose we have a collection of documents $D = \{d_1, d_2, ..., d_N\}$ with words from a vocabulary $W = \{w_1, w_2, ..., w_M\}$. By ignoring the sequence order in which words occur in a document, one can summarize the data in a $N * M$ document-word co-occurrence table A, where the element A_{ij} indicates the number of times a term w_j occurs in document d_i. This representation of document collections is called the vector-space representation. It is widely used by many algorithms such as text retrieval based on keywords. However, this representation suffers from the issue of polysemys and synonymys.

To address this problem, PLSA introduces the latent semantic variables $Z = \{z_1, z_2, ...z_K\}$. Figure 1 shows the statistical model PLSA is based on. The model, which is called aspect model, has two forms, namely the symmetric and asymmetric forms. We use the symmetric form in our paper, as [1] did.

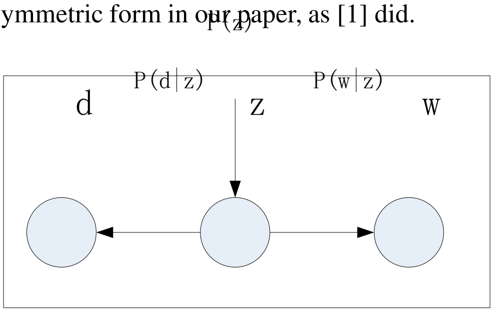

Figure 1. Graphical model representation of the aspect model in symmetric form. d indicates a document, w is a word, and z is a latent semantic variable.

Given this aspect model, the joint probability model over DxW is defined as:

$$P(d,w) = \sum_{z \in Z} P(z)P(w|z)P(d|z) \quad (1)$$

In this way, the relationship between two documents can be expressed by $P(d|z)$ and $P(z)$, instead of the original $P(d,w)$, thus the issue of polysemys and synonymys can be solved. The output of the PLSA algorithm is the best estimation of the parameters $P(d|z)$, $P(w|z)$, and $P(z)$. Following the likelihood principle, one determines the best estimation by maximization of the log-likelihood function:

$$L = \sum_{d \in D} \sum_{w \in W} n(d,w) \log P(d,w), \quad (2)$$

This algorithm can be implemented using the Expectation-Maximization (EM) algorithm[10]. First part of the text dataset is held out as test data to evaluate the quality of the estimation. Then EM iterations are performed on the rest of the dataset. The log likelihood L computed using an estimation on the held-out data increases after each EM iteration, until it reaches a maximum point, which is defined as the convergence point.

An EM iteration can be divided into two steps, namely **(i)** the expectation step (E-step), where posterior probabilities $P(z|d,w)$ are computed for the latent variables z, based on the current estimates of the parameters, and **(ii)** the maximization step (M-step), in which parameters $P(d|z)$, $P(w|z)$, and $P(z)$ are updated for given posterior probabilities computed in the previous E-step.

In other words, the E-step calculates the probability of

$$P(z|d,w) = \frac{P(z)[P(d|z)P(w|z)]}{\sum_{z'} P(z')[P(d|z')P(w|z')]}, \quad (3)$$

where $P(z|d,w)$ is the posterior probability of latent variable z, given the occurrence of a term w in the document d, and the parameters $P(d|z)$, $P(w|z)$, and $P(z)$ are the results of the last EM iteration.

And the M-step updates the parameters by the following equations:

$$P(w|z) = \frac{\sum_d n(d,w)P(z|d,w)}{\sum_{d,w'} n(d,w')P(z|d,w')} \quad (4)$$

$$P(d|z) = \frac{\sum_w n(d,w)P(z|d,w)}{\sum_{d',w} n(d',w)P(z|d',w)} \quad (5)$$

$$P(z) = \frac{\sum_{d,w} n(d,w)P(z|d,w)}{R}, R = \sum_{d,w} n(d,w) \quad (6)$$

Tempered EM (TEM) is almost the same as standard EM, but it uses a power β, which should be less or equal to 1, to avoid overfitting. And it calculates $P(z|d,w)$ as

$$P_\beta(z|d,w) = \frac{P(Z)[P(d|z)P(w|z)]^\beta}{\sum_{z'} P(z')[P(d|z')P(w|z')]^\beta} \quad (7)$$

The TEM algorithm can be described as follows:[1]

1. Set $\beta = 1$ and perform EM until the log likelihood of the test data deteriorates
2. Decrease β by setting $\beta = \eta\beta$, where η is a parameter less than 1
3. As long as the log likelihood on test data improves, continue TEM iterations at this value of β
4. Stop on β if decreasing β does not yield further improvements, otherwise goto step 2

The PLSA algorithm basically starts from randomized Pz, $P(d|z)$ and $P(w|z)$, and performs TEM iterations until convergence.

3. The Baseline Program

We start from a simple reference implementation which is provided in the cluster package of Lemur 4.2[5]. In this implementation, the parameters $P(d|z)$, and $P(w|z)$, and $P(z)$ are stored in matrices and the common parts $\sum_{d,w} n(d,w)P(z|d,w)$ in equations 4, 5, 6 are stored in an array of size Z named *denominator*, while the posterior probabilities $P(z|d,w)$ are calculated when needed.

So the implementation has space complexity of

$$C_s = O(D*Z+W*Z+Z) = O(D*Z+W*Z),$$

and time complexity for each Tempered EM (TEM) iteration of

$$C_t = O(W*D*Z^2+N*Z^2) = O(W*D*Z^2).^1$$

For convenience, we use D, W and Z to denote the number of documents, words, and latent semantic variables. Due to the space limit of the paper, we have to leave out the deduction of the complexity.

As can be seen, the Lemur implementation is inefficient, mainly due to the in-need calculating of $P(z|d,w)$. By reorganizing the order of the operations, we managed to reduce the time complexity, while keeping the space complexity unchanged. The time complexity of a TEM iteration in the optimized program is:

$$C_t' = O(N*Z).$$

This is actually the number of $P(z|d,w)$ needed to be calculated in each iteration. So we are not doing any unnecessary calculations.

Apart from the reduction of computational complexity, we also applied other common optimization techniques to the program, such as loop interchange, and loop unrolling. As we are running the program on x86 machines, we also take advantage of SSE instructions to accelerate the program.

Dataset	cranmed	la12	new3
Lemur	136	915	2257
Baseline	0.195	0.537	1.54

Table 1. Time spent in an TEM iteration on HPC134 in seconds.

A comparison of the Lemur implementation and the baseline program is presented in Table 1. Experiments were carried out on HPC134 (refer to Table 3 for details). We used three different datasets, with different number of documents and words. As expected, our baseline program is orders of magnitude faster than Lemur, mainly due to reduced complexity.

[1] N is usually orders of magnitude smaller than W*D.

4. Parallelization of the Algorithm

4.1. Data Race and the Block-dispatching Algorithm

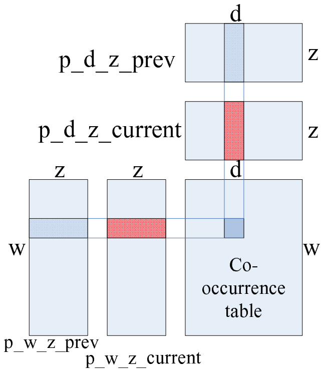

Figure 2. A demonstration of how the matrices are accessed when calculating the values of $P(z|d,w)$ for a specific (d,w) pair. A value $n(d,w)$ from the co-occurrence table, and a row in the previous matrices *p_d_z_prev* and *p_w_z_prev* are read in to produce these values of $P(z|d,w)$. These values are then accumulated to the corresponding row in *p_d_z_current* and *p_w_z_current*. The *p_d_z* matrices are rotated in the figure for the convenience of drawing. They are actually stored as D rows and Z columns.

Figure 2 demonstrates how the matrices are accessed in our program. We introduce several new symbols in this figure. The matrices *p_d_z* and *p_w_z* store the values of $P(d|z)$ and $P(w|z)$, and the co-occurrence table is named $n(d,w)$. *p_d_z_prev* and *p_w_z_prev* store the values produced in the previous TEM iteration, while *p_d_z_current* and *p_w_z_current* store the current ones. The program basically reads a value in $n(d,w)$, and a row in *p_d_z_prev* and *p_w_z_current*, uses them to calculate the corresponding $P(z|d,w)$, then adds the values to *p_d_z_current* and *p_w_z_current*. If two processors get two elements of the co-occurrence table in the same row or column, they may write into the same row of *p_d_z_current* or *p_w_z_current* at the same time, thus causing data race.

To avoid data race, we divided the co-occurrence table into n columns and n rows, and dispatch the blocks to the processors using an algorithm that makes sure no blocks in the same column or row would be processed at the same time. A straightforward approach is to search through all blocks for an available block every time. But this simple approach is too costly in multiprocessing environments. Searching through all blocks is an operation that should be put in a critical section. Considering the complexity of this operation, putting it in a critical section would incur great lock overhead, and thus should be avoided.

Instead of this straightforward algorithm, we implemented an algorithm named TwoForward (TF) which is described below:

1. at the begining of the algorithm, put all the blocks in the diagonal into a queue Q

2. when a processor P is free, it tests if Q is empty, if not, it pops a block $B_{i,j}$ from Q, otherwise the algorithm stops

3. P push the block $B_{(i-1) \bmod n,j}$ and $B_{i,(j+1) \bmod n}$ into Q if this won't raise a conflict

4. P process the block

5. goto 2

This algorithm is lightweight, hence suitable for multi processing environment. Although it cannot exploit all possible chances of parallel processing, it would be sufficiently good if the blocks are well divided. According to our experiments, TF works well with our blocking algorithm described below, and achieves good load balance at relatively low cost.

4.2. Load Balance and the Blocking Strategy

The co-occurrence tables of our datasets are sparse matrices, meaning that two columns may have different number of nonzero elements. Moreover, two of the datasets we use, *new*3 and *cranmed* as described in the experimental section, are sorted such that most of the nonzero elements are placed on the upper-right corner of the matrix. If we divide the blocks so that they have the same number of rows and columns, the number of nonzero elements they have will differ greatly. For the *new*3 and *cranmed* datasets, the amount may even differ by orders of magnitude.

In order to address this issue, we developed an adaptive blocking strategy called FairDividing (FD). Assuming that the dataset should be divided into $K*K$ blocks, and $B_{i,j}$ is an arbitrary resulting block whose elements ranges from A_{ab} to A_{cd} where $c \geq a$ and $d \geq b$, the FD algorithm divides the blocks so that:

$$\begin{aligned} & sum_{i=a}^{c} R_i \approx N/K, \\ and \quad & sum_{j=b}^{d} C_j \approx N/K, \end{aligned}$$

where R_i is the number of nonzero elements in row i of the co-occurrence table, ,C_j is the number of nonzero elements in column j, and N is the total number of nonzero elements.

This heuristic algorithm tend to produce smaller blocks where the nonzero elements are densely distributed, while produce larger blocks where the nonzero elements are scarce. Though the algorithm does not guarantee that all the blocks contain the same number of nonzero elements, it is good enough to balance the workloads of the processors, as proved by our experiments.

The number of blocks a co-occurrence table is divided can also affect the performance of the program. Dividing the table into too few blocks would make the load balancing between processors hard, while dividing into too many blocks incurs too much communication. According to our experiments, when running on n processors, dividing the table into $2n * 2n$ blocks is a balance point between load balance and communication cost.

5. Experiment Results

The datasets used in the experiments are provided by Cluto[9]. We selected three datasets containing different amount of documents. Table 2 shows the information on these datasets.

Dataset	Documents	Unique Words	Nonzero Elements
cranmed	2431	41681	140658
la12	6279	31472	939407
new3	9558	83487	2295120

Table 2. Datasets used in the experiments. These three datasets are selected from the Cluto[9] datasets.

The experiments are conducted on two systems, namely HPC134 and Tulsa. The HPC134 system consists of two Intel quad-core CPUs. Each CPU encapsulates two chips. And each chip contains two processor cores and an L2 cache shared by the two cores. The Tulsa system has four Intel dual-core CPUs, each of which contains two cores sharing L3 cache. Detailed information about the two machines is shown in Table 3.

5.1. Effect of the Blocking and Job Scheduling Algorithms

During the parallelization of the program, we proposed a blocking algorithm and a job scheduling algorithm to achieve load balance. In this section, we conduct experiments to show the effect of these two algorithms.

	HPC134	Tulsa
# of CPUs	2	4
cores/CPU	4	2
# of cores	8	8
L1 D-cache	32KB	32KB
L2 cache	4MB/2cores	2MB
L3 cache	none	16MB/CPU
main memory	6GB	4GB

Table 3. Machines used in the experiments

The purpose of the blocking algorithm is to divide the datasets evenly, so that better load balance can be achieved. Table 4 shows the $4*4$ blocking results of the three datasets. We compared our algorithm (FD) with a simple algorithm which divides the blocks to make them have same amount of rows and columns. Note that the nonzero elements in $la12$ are almost evenly distributed in the matrix. As a result, both algorithms perform well on $la12$. We will focus our discussion mainly on $new3$ and $cranmed$.

We can see from Table 4 that the simple algorithm is not able to divide the blocks evenly for datasets $new3$ and $cranmed$. The number of nonzero elements in the blocks differs by orders of magnitude, and some blocks even contain no nonzero elements. Moreover, the simple algorithm tends to put most of the elements in the first row, which makes these blocks unable to be processed in parallel. Even if we apply the "heaviest-weighted first", i.e. process the blocks with the most nonzero elements first, there will still be great imbalance, for the blocks in the first row can only be processed by one processor at a time.

On the contrary, our algorithm adapts well to different datasets, and divides them more evenly. All the three datasets are divided into blocks of similar size. Though there is a block with only 944 nonzero element in the dataset $cranmed$, the amount of nonzero elements in the other blocks are nearly the same, with difference less than three times.

As a comparison to our job scheduling algorithm, we implemented a straightforward algorithm, which is called SearchAll (SA). For a processor which is just finished with block B_{ab}, the SA algorithm will first search through all $B_{aj}, j \in [0, n-1]$ and $B_{ib}, i \in [0, n-1]$ for a block available. If it fails, it will search through all the blocks until it finds an available block or there is no blocks left.

Table 5 presents the effects of the two algorithms on the dataset $new3$, where the dataset is divided into

cranmed _simple	19770	18532	18143	15790
	12144	6445	3665	1720
	151	5497	8494	4906
	0	0	1330	10001
cranmed _FD	8432	8397	7917	6925
	8898	8603	7842	6306
	13399	8763	5653	3832
	944	5887	10259	14531
la12 _simple	59037	56566	55519	56383
	51535	48858	48261	49159
	54799	52731	51658	51658
	53798	52398	51080	52288
la12 _FD	53155	52921	52708	52668
	53599	53077	53415	53684
	53056	53045	53139	52200
	51822	52496	52216	52527
new3 _simple	548428	516401	440373	402650
	21477	33193	23358	14323
	0	18037	20690	10373
	0	0	5639	10859
new3 _FD	148261	119947	121885	128855
	136793	127450	126670	128969
	136883	130776	122117	126811
	95165	138280	145837	131102

Table 4. Effect of different blocking algorithms. This table shows the number of nonzero elements in each block when the datasets are divided into 4*4 blocks using FairDividing and simple algorithms.

	1P	2P	4P	8P
TF_tot	1.1493	1.2048	1.3430	2.0199
TF_ovh	0.0001	0.0002	0.0006	0.0550
TF_imb	0.0000	0.0531	0.0298	0.0386
SA_tot	1.1535	1.2474	1.4506	2.2139
SA_ovh	0.0002	0.0583	0.1718	0.3983
SA_imb	0.0000	0.0623	0.0574	0.0391

Table 5. Comparison of two job scheduling algorithms. This table shows the time costs of two job scheduling algorithms, TwoForward and SearchAll, on dataset *new*3 in seconds. The "tot" rows show the total time spent in each EM iteration, while "ovh" indicate the time spent executing the algorithm, and "imb" present the imbalance between the processors. 1P, 2P, 4P and 8P stands for 1, 2, 4, and 8 processors.

$2n * 2n$ blocks for n processors. TF_tot is the total CPU time used in an TEM iteration (i.e., the wall clock time multiplied by number of processors), using our TwoForward (TF) scheduling algorithm; TF_ovh is the time spent in executing the algorithm; and TF_imb is the imbalance between the processes. The blocking algorithm used is the FD algorithm described above.

From Table 5, we can see that:

1. TF algorithm is better than SA in terms of total execution time;
2. the overhead of TF is much less than SA, and it grows slower with the number of processors;
3. both algorithm exhibits marginal load imbalance, which proves the effectiveness of our blocking algorithm and the block dispatching algorithm.

Another trend worth noticing is that the margin of total execution time between TF and SA is less than the gap in the execution time of the algorithms. The main reason is that SA makes better use of the cache. The TF algorithm tends to visit the blocks along the diagonal, which will incur more cache misses. However, the TF algorithm is still better than SA, for the SA algorithm is too heavy-weighted (i.e., the execution of the algorithm is too time-consuming). Therefore, the cost of executing the algorithm easily overwhelms the benefit of higher cache hit ratio.

Figure 3 shows the total wasted time of the FD dividing algorithm and TF block scheduling algorithm, including the overhead of the two algorithms and time wasted for load imbalance, as percentage of the total execution time. As comparison to the FD dividing algorithm, we implemented a "simple" algorithm that divides the blocks so that they have same number of rows and columns. The datasets are divided into $2n * 2n$ blocks when n processors are used. We use 1P, 2P, 4P and 8P to indicate the number of processors used. We will also use these symbols in the following figures and tables.

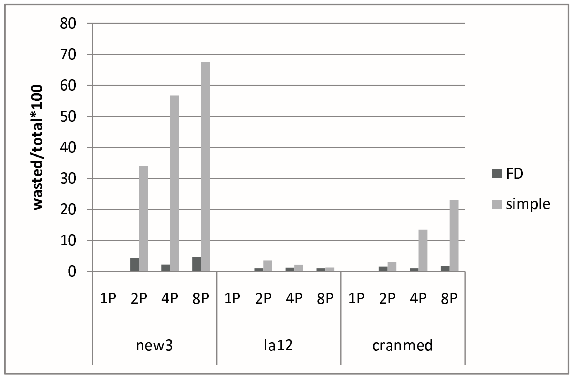

Figure 3. Overhead of the blocking algorithms as percentage of total execution time. 1P, 2P, 4P and 8P indicate the number of processors.

From the Figure 3, we can see that when using FD with TF, the wasted time only takes a small fraction of the total time (less than 5%). On the other hand, using "simple" algorithm incurs great performance panalty for poor load balance, and hence much wasted time. In the case of *new*3 with eight processors, the wasted time even takes up more than 65% of the total execution time. However, the wasted time of FD and "simple" on the dataset *la*12 are almost the same. The main reason is that the nonzero elements in *la*12 are nearly evenly distributed across the co-occurrence table.

In conclusion, the FD and TF algorithm have enabled us to achieve good load balance at low cost.

5.2. Speedup and Memory Performance

In this section, we examine the speedup of the program and its memory characteristics. For the ease of analysis, we divide the datasets into $16 * 16$ blocks in this experiment.

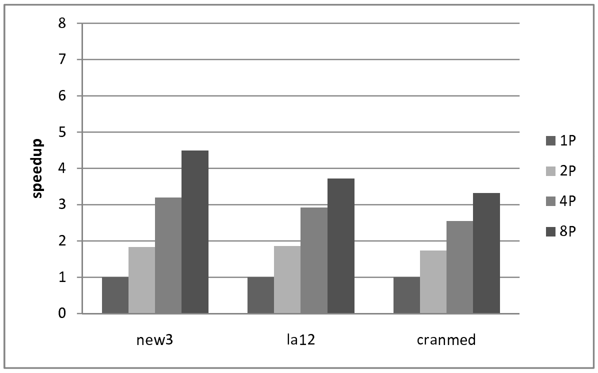

Figure 4. Speedup of the program on HPC134

The speedup result on HPC134 system is shown in Figure 4. We can see that the scalability of the program on the HPC134 system is unsatisfactory. Although the speedups from 1*P* to 2*P* are acceptable, those of 4*P* and 8*P* are poor. The major cause for poor speedup is the limited memory bandwidth.

For comparison, we tested the speedup of the program on the Tulsa system. The Tulsa system also has eight cores, but it has much higher memory bandwidth than HPC134. Figure 5 shows the memory bandwidth of the two systems, which is obtained with STREAM[12] compiled using ICC 9.1 and compiler options "-O3 -xW -openmp". STREAM reports four metrics, namely "Copy", "Scale", "Add" and "Triad", the difference between each other being about 10%. The memory bandwidth we present here refers to the result of "Triad", for its memory access pattern is the most similar to that of our program.

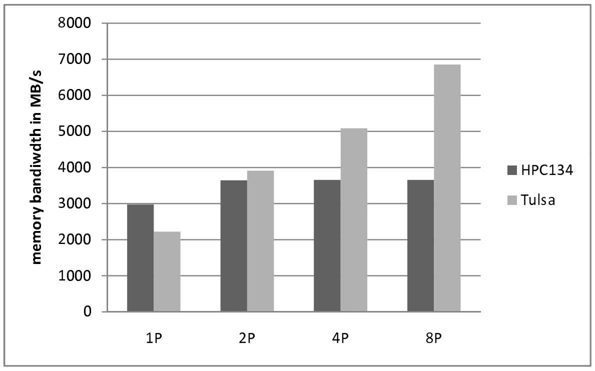

Figure 5. Memory bandwidth of the two systems

We can see from Figure 5 that the Tulsa system has much higher memory bandwidth than HPC134, and the available bandwidth grows gradually with the number of processors, while that of HPC134 grows little.

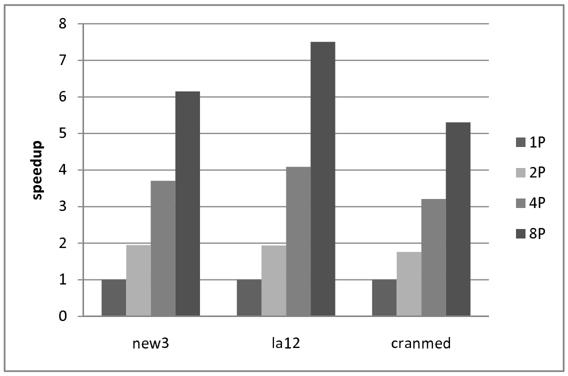

Figure 6. Speedup of the program on Tulsa

Figure 6 shows the speedup on Tulsa. As expected, the speedups on Tulsa are higher than those on HPC134.

To give further insights into the effect of limited memory bandwidth on our program, we used Intel VTune to sample the program and get the memory access related stalls. Figure 7 depicts the breakup of the total execution time into memory related pipeline stalls and other parts. This experiment is performed on the HPC134 system.

As we can see, the pipeline stalls caused by memory access take up a large part of the total CPU cycles, and they grow dramatically with the number of threads. The high stall_mem/total ratio of 1*P* indicates that our program is memory-intensive. Although we divided the datasets into blocks which can fit into L2 cache, it is im-

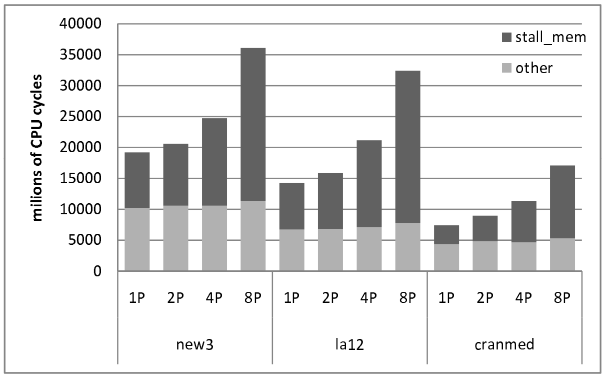

Figure 7. Memory related stalls on HPC134

possible to fit the datasets into cache as a whole. Hence there are still a lot of cache misses, making the program bandwidth-hungry.

The growth of stalls from 1*P* to 4*P* can be explained by the contention of processors for bandwidth. And we can also see that the stalls of 8*P* nearly doubles that of 4*P*. That is partly because eight processors competing for bandwidth brings more contention, and partly because the cache-per-core halves from 4*P* to 8*P*, which incurs more cache misses.

As for the "other" part, it stays nearly the same in all cases, although there is trivial increase caused by the increased execution time of the TF algorithm and the overhead of OpenMP. The only exception is the *cranmed* dataset, which shows non-negligible increase in the "other" part. We analyzed the program and found that the barrier is causing the problem. The barrier is used by OpenMP to synchronize the threads. The more threads used, the more time the barrier takes. However, for larger datasets like *new*3 and *la*12, the time spent by barrier is just a small fraction compared to the total execution time. But in the case of *cranmed*, the barrier time becomes a significant part, which lowers the speedup. As a result, the speedup of *cranmed* is generally lower than the other two, as shown in Figure 4 and Figure 6.

In conclusion, the limited memory bandwidth of the HPC134 system prohibits our program from getting good speedup. In order to achieve better speedup of this kind of memory-intensive programs on shared memory systems, higher bandwidth is needed.

6. Conclusions

In this paper, we present the optimization, parallelization and characterization of a PLSA implementation. By optimizing the implementation, we have made the algorithm acceptable for usage on datasets of over 10,000 documents. Based on the optimized code, we parallelized the program using OpenMP, and proposed a block dividing algorithm and a thread scheduling algorithm. The parallel implementation gets fair speedup on our multi-core systems. The experimental results show that the program is memory-intensive, which needs higher memory bandwidth to achieve good performance on shared memory systems.

References

[1] Hofmann T.: Probabilistic latent semantic indexing. SIGIR '99 8/99 Berkley

[2] S. Deerwester, S. T. Dumais, G. W. Furnas, Landauer. T. K., and R. Harshman. Indexing by latent semantic analysis. Journal of the American Society for Information Science, 41, 1990.

[3] OpenMP Architecture Review Board. OpenMP specifications.
Available at `http://www.openmp.org`

[4] Deerwester S., Dumais S. T., Furnas G. W., Landauer T. K., and Harshman R.: Indexing by latent semantic analysis. Journal of the American Society for Information Science

[5] The Lemur Toolkit for Language Modeling and Information Retrieval
Available at `http://www.lemurproject.org`

[6] P.W. Foltz and S. T. Dumais. An analysis of information clustering methods. Communications of the ACM, 35(12):51-60, 1992.

[7] T.K. Landauer and S.T. Dumais. A solution to Plato's problem: The latent semantic analysis theory of acquisition, induction, and representation of knowledge. Psychological Review, 104(2):211-240, 1997.

[8] J.R. Bellegarda. Exploiting both local and global constraints for multi-span statistical language modeling. In Proceedings of ICASSP'98, volume 2, pages 677-680, 1998.

[9] CLUTO-Family of Data Clustering Software Tools
Available at `http://glaros.dtc.umn.edu/gkhome/views/cluto`

[10] A.P. Dempster, N.M. Laird, and D.B. Rubin. Maximum-likelihood from incomplete data via the EM algorithm. J. Royal Statist. Soc. B, 39:1-38, 1977.

[11] HyperTransport Specification 3.0
Available at `http://www.hypertransport.org`

[12] STREAM: Sustainable Memory Bandwidth in High Performance Computers
Available at `http://www.cs.virginia.edu/stream/`