Parallel Information Extraction on Shared Memory Multi-processor System

Jiulong Shan, Yurong Chen, Qian Diao and Yimin Zhang
Intel China Research Center
Intel Corporation
8/F, Raycom Infotech Park A, NO. 2, KeXueYuan South Road
Beijing, 100080, China
Tel: +86-10-82611515-1327
Contact Email: jiulong.shan@intel.com

Abstract

Text Mining is one of the best solutions for today and the future's information explosion. With the development of modern processor technologies, it will be a mass market desktop application in the many-core era. In Text Mining system, Information Extraction is a representative module and is the most compute intensive part. In this paper, we study the performance of parallel Information Extraction on shared memory multi-processor systems in order to gain some insights of such applications on the future's many-core architecture. In implementation, Conditional Random Fields (CRFs) algorithm is selected as the core of module Information Extraction. Based on the newest CRFs toolkit FlexCRFs*, we make several serial optimizations and then parallelize it with MPI and System V. IPC/shm. We also conduct a detailed performance analysis of this parallel application on the target system.*

1 Introduction

These days, everyone in the world faces an information overload in their everyday life, in which the most general and important one is text information.

One common scenario for this is that when entering a query in a search engine (such as Google, Yahoo or Baidu), a person will face thousands of return hits. Due to time limitations, he can only read several high-ranked hits recommended by the engine. Sometimes, this will miss some important information and cause an information gap.

To fill this gap, Text Mining will be a good choice. Text Mining is the nontrivial extraction of implicit, previously unknown, and potentially useful information from a large amount of textual data. By using those state-of-the-art algorithms, we can have computers help humans to extend their reading ability.

Meanwhile, the development of Internet's bandwidth and modern processor technologies [10] will make it feasible to download all the pages and to deal with them locally on our desktop. This trend leads to our research work on the Personal Text Mining System.

As shown in Fig. 1, we designed the Personal Text Mining System to help provide the following solutions. First the computer can download all the related hits from the corresponding web servers automatically. After that, with some post-processing, it can display all the information in a more readable way. For example, by using clustering we can find the hidden topics behind those hits and by using multi-document summarization we can provide a brief summary for each topic.

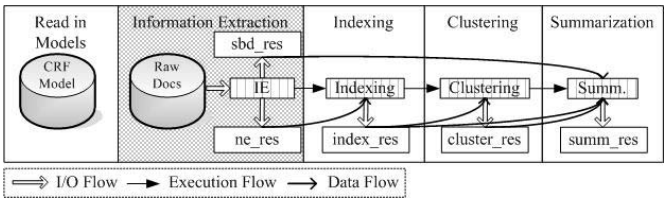


Figure 1. Architecture of Personal Text Mining System

In Personal Text Mining System, Information Extraction (IE) is the basis of all other advanced modules. It includes two sub-tasks: Sentence Boundary Detection (SBD) and Name Entity Extraction (NE). First of all, it segments the input raw documents into individual sentences (*sbd_res*) for further use and extracts the name entities (*ne_res*) from them. After that, the extracted name entities can help put more emphasis on those key terms while doing clustering and summarization. Also, it can provide indicative labels for each topic. In implementation, Conditional Random

Fields (CRFs) [12] algorithm is chosen as the core of module IE.

In the whole system, IE is a representative module. It includes most of the operations needed for Text Mining applications, such as: dealing with stemming and acronym, sequence vectorization and dictionary finding. Also, IE is the most computate intensive part in our Personal Text Mining System, consuming about half of the runtime.

So, in this paper we present the optimization and parallelization of module IE and characterize its performance on two shared memory multi-processor systems. We first identify the hotspots of this module, perform data structure refinements for serial optimization and then choose MPI as the parallel programming language. The above efforts make module IE more powerful for handling larger datasets in a short time scale. *To sum up, this work studies the problem of how to perform Information Extraction more efficiently on modern shared memory multi-processor system and try to provide some useful insights of such applications on the future's many-core architecture for application implementation and architecture design.*

The remainder of this paper is organized as follows: Section II describes the algorithm, implementation and quality of CRFs based Information Extraction. Section III presents the application's optimization and parallelization. Experiment and performance analysis are studied in Section IV. Section V, concludes the work of this paper and outlines the future work.

2 CRFs based Information Extraction

2.1 Conditional Random Fields Algorithm

In text processing, sentence boundary detection and name entity extraction are two typical tasks of Information Extraction. CRFs brings together the pros of generative probabilistic model and classification model. Lafferty et al. [12], Sha & Pereira [14] and Liu et al. [13] showed that CRFs beats related classification models as well as generative models on part-of-speech tagging task, shallow parsing task and sentence boundary detection task, respectively. Therefore, we choose it as the core of module IE.

CRFs contains two separate parts: training and decoding.

A CRFs model is trained by maximizing the log-likelihood for a metric of input sequence x and label sequence y. Generally, the training process needs about dozens of hours to generate a CRFs model. Once a model is generated, it will not be changed during the decoding process. Therefore, we can regard it as an offline module and focus on the decoding part.

CRFs decoding is also working on the sequence level. It processes the input document sequence by sequence and finds the most likely state sequence y' corresponding to the given observation sequence x.

$$y' = \arg\max_{y} \frac{\exp(\lambda \cdot F(y, x))}{Z_\lambda(x)} \tag{1}$$

In Equation (1), the CRFs' global feature vector for input sequence x and label sequence y can be denoted as:

$$F(y, x) = \sum_{i} f(y, x, i) \tag{2}$$

$$Z_\lambda(x) = \sum_{y} \frac{\exp(\lambda \cdot F(y, x))}{Z_\lambda(x)} \tag{3}$$

Because $Z_\lambda(x)$ does not depend on y, and $F(y, x)$ can be decomposed into a sum of terms for consecutive pairs of labels, so the most likely y' can be found with the viterbi algorithm [6]. For a chain structure, the conditional probability of a label sequence can be expressed concisely in matrix form. Then for a given sequence x, define the transition matrix of position i as:

$$\begin{aligned} M_i[y, y'] &= \exp(\lambda \cdot f(y, y', x, i)) \\ &= \exp(\sum_{k} \lambda_k \cdot f_k(y, y', x, i)) \end{aligned} \tag{4}$$

And in Equation (1), the conditional probability of a label sequence can be rewritten as:

$$P_\lambda(y|x) = \frac{\prod_{i=1}^{L+1} M_i[y, y']}{\prod_{i=1}^{L+1} M_i(x)} \tag{5}$$

where $y_0 = start$ and $y_{n+1} = stop$, the normalization (partition function) $Z_\lambda(x)$ is the $(start, stop)$ entry of the product of these matrices:

$$Z_\lambda(x) = M_1(x) \cdot M_2(x) \ldots M_{L+1}(x) \tag{6}$$

The computational complexity of viterbi decoding in CRFs is $O(L \cdot NL^2)$, where L is the average sequence length and NL is the number of different labels.

Pseudo codes of the CRFs based SBD and NE are listed below (algorithm 1, 2 and 3). Because this work is mainly based on the sequence level, so we only present these pseudo codes on the same level.

2.2 System Development

The implementation of the module IE is based on an open source software toolkit: *FlexCRFs* [11].

FlexCRFs is a CRFs toolkit for segmenting and labeling sequential data written in C/C++ using STL library. It was implemented based on the theoretical model presented

Algorithm 1: CRFs based SBD
Begin
1: **for** each word w in the original document **do**
2: judge whether w is a potential sentence boundary;
3: **if** it's a potential one **then**
4: generate the corresponding observation sequence;
5: perform viterbi decoding to make sure whether it is a sentence boundary;
6: **end if**
7: **end for**
End

Algorithm 2: CRFs based NE
Begin
1: **for** each sentence s in the result of SBD **do**
2: make transformations to sentence s and generate the corresponding observation sequence;
3: perform viterbi decoding on the observation sequence to find the name entities inside it;
4: **end for**
End

in [12] and [14]. Based on the kernel of *FlexCRFs*, we developed the components for SBD, NE and also several utilities to facilitate the required transformation.

In order to execute the decoding step, we have to make some transformations on the original sequences to generate the corresponding observation sequences. It can be seen from later sections that this transformation consumes a large portion of the whole decoding time.

3 Optimization and Parallelization

3.1 Experiment Setting

The following studies are conducted on a 16-way Intel Xeon shared memory multi-processor system. It has 16 x86 processors running at 3.0GHz, 4 levels of cache with each 32MB L4 cache shared amongst 4 CPUs. The sizes of the L1, L2 and L3 caches are 8K, 512K and 4MB respectively. As for the interconnection, the system uses two 4x4 crossbars. We use mpich-1.2.5.2 (affiliated with gcc 3.3.3) compiler to generate the executables with option -O3, to enable the high levels of compiler optimizations. To further study the influence of cache settings, we also analyze the program's performance on another 4-way Intel Xeon (2.8GHz) system with a smaller 2MB L3 cache and no combined L4 cache.

We use three different datasets. The first one is downloaded from the links of Google general search results with query word "beijing". It contains 805 web pages and after filtering (web page cleaning, e.g. removing HTML tags and

Algorithm 3: CRFs based IE
Begin
1: read in the CRFs model for SBD and NE;
2: **for** each input document d **do**
3: apply **Algorithm 1** on document d;
4: apply **Algorithm 2** on the result of previous step;
5: **end for**
End

extra scripts) its size is about 11MB (denoted as "beijing"). The other two are generated from the Reuters News Collections [3]. One is the news archive of 19961126, which contains 3030 XML files and the filtered size is 25MB (denoted as "1126"). Another one is created by ourselves by merging several news articles into one document. There are totaly 4102 documents and the filtered size is 141MB (denoted as "reuter").

3.2 Workload Characterization

According to previous descriptions in Section II, module IE can be roughly divided into two parts: Model Initialization (MI) and Document Processing (DP). The former one reads in all the pre-trained CRFs model data and generates the corresponding data structures in memory, while the latter one consequently performs SBD and NE on each raw document.

By breaking down the runtime with different datasets, we can easily observe that DP is the most time consuming part (consumes about 90% of the total runtime). With the increasing size of dataset, the runtime of DP increases dramatically, while that of MI remains the same and takes only a small percentage. But, for small dataset, like "beijing" and "1126", the runtime of MI is still noticeable, especially while running in parallel.

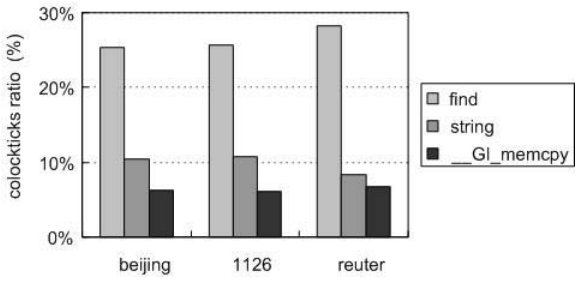


Figure 2. Function Profiling of Module IE with Different Datasets

With Intel VTune Performance Analyzer [5], we also generate function level profiles of the whole module with different datasets, as shown in Fig. 2. Inside them, "find"

is the search operation of STL's *map* container; "string" is the constructor of C++ class string and "__GI_memcpy" is the memory copy function of libc. All these three functions consume about 42% of the total runtime.

3.3 Key Observations

Based on the above workload analysis, together with an understanding of the kernel functions, we can make the following observations:

First, this CRFs based IE is a memory intensive application. This can be seen from Fig. 2 that the 3 top functions are all memory related.

"__GI_memcpy" and "string" mainly occurred in the transformation from original sequence to the observation sequence, which use several buffers for temporal storage. "find" is used to determine whether a sequence exists in a pre-defined dictionary, which needs multiple string comparisons for each single call. So, the serial optimization should be focused on these parts.

Second, the parallelization of CRFs based IE can be conducted with different granularities, such as decoding level, sequence level and document level.

This application is not computation intensive, but memory intensive. If working on the decoding level, it will expose more variables to be shared by all the processes and thus cause extra synchronization cost. Also, considering the I/O synchronizations, it is better to choose document level parallelization.

3.4 Serial Optimization

Several serial optimizations have already been done in module IE. These optimizations are mainly related with the STL containers.

In module IE, one common task is to deal with a set of terms, such as maintaining a vector or map of all the related terms, finding whether a term exists in a given set and making some modifications to them. To achieve this and also for programmability, in the original implementation of module IE, some STL containers are widely used across the source codes, like *vector*, *list*, *set* and *map*. As to some other recent developed Text Mining Toolkits, it is also the same for using of STL containers, such as Lemur [1]. This raises some performance issues and in our optimization the following principles are adopted.

The first one is to use right STL container. Different STL containers are designed for different scenarios. We should choose the most suitable one in implementation.

In the original implementation of module IE, the sequence dictionary has about 120,000 items and is defined as: *map<string, element> dict*. The container *map* is constructed as an rb-tree in memory. On average, a find operation will need about *logN* comparisons (*N* is the number of items). For this case, there will be around 5 string comparisons for each find operation and the find operation is invoked frequently.

Besides these unwanted comparisons, it will also cause extra cache misses. Because the size of the dictionary is larger than that of the L3 cache, so some of these "find" operations need to access main memory. On a modern processor it will surely cause lower performance.

After finding this, we use an extension STL container *hash_map* to replace the occurrences of *map*. If we use large enough buckets and suitable hashing function, all the items can be distributed into each bucket evenly and thus can help reduce the average number of comparisons. With this replacement, module IE can achieve a performance improvement between 12% to 18% with different datasets.

The second one is to use non-STL data structure for replacement. In some scenarios, STL can help on the programmability, at the same time affect the performance a lot. So in performance-oriented scenarios, some replacement data structures should be created.

In the original version, a lot of variables are defined as *vector* of string or *map* of string. From the profiles generated by VTune (shown in Fig. 2), we can see a large amount of unnecessary string construction and memory copy operations. This is because, each time a string is inserted into a *vector* or a *map*, it will invoke a corresponding construction and a memory copy operations. Most of them are unnecessary and a waste of the runtime.

So, in optimization, we create class *CompactStrings* for replacement. It manages an internal buffer which avoids those unnecessary operations. Equipped with this and some other string replacements, module IE can achieve 4X performance improvement.

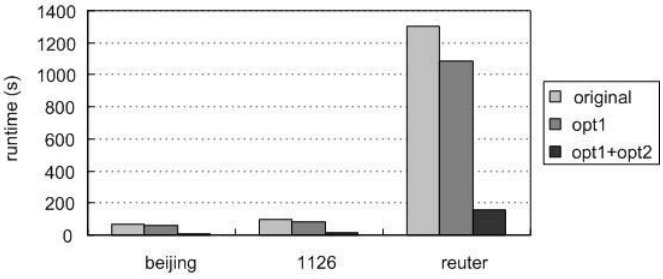


Figure 3. Performance Improvement of Serial Optimization with Different Datasets

The aggregated performance improvement with different datasets is shown in Fig. 3.

3.5 Parallel Implementation

3.5.1 Programming Model Selection

The first step of parallelization is to select a proper parallel programming model. To make the decision, we check the performance of Message Passing Model and Shared Variable Model on some kernel codes separately [15].

Because the target machine is of shared memory architecture, the straight selection should be the shared variable programming model, e.g. Pthread and OpenMP. But, as described above, module IE uses STL containers extensively as its primary data structure. Because STL has its own internal memory management policies, the frequent insert and delete operations will invoke a lot of concurrent accesses to the center heap which is shared by all threads. Therefore, in thread-level parallelization, this will definitely cause contentions and affect the scalability. Although some recent developed heap aware allocators [4] [2] can reduce the impact of this problem, it still exists.

Message passing programming model, like MPI, provides process level parallelization. Thus, it will not have this heap contention problem naturally. The drawback of MPI on shared memory architecture is that it does not support data sharing between processes and will cause duplicate data allocations.

In our approach, we choose MPI as the base programming language and use System V. IPC/shm [7] as a plus to enable the inter-process data sharing. The experimental results shown in section IV prove the efficiency of this approach.

3.5.2 Implementation Issues

The basic principle of parallelization is to assign all the processing tasks to each processor evenly. According to the description of Algorithm 3 and the observations above, for MI, we can let different process read in different part of the model separately; and for DP, the best way to parallelize it is performing data parallelization in document level. The corresponding parallel algorithm for IE is listed below:

Algorithm 4: Parallel CRFs based IE
Begin
1: read in different part of the CRFs model for SBD and NE separately;
2: assign all the documents to each process evenly;
3: **for** each assigned input document d **do**
4: apply **Algorithm 1** on document d;
5: apply **Algorithm 2** on the result of previous step;
6: **end for**
End

After designing the parallelization scheme, we meet the following issues in implementation.

One issue is the speedup downfall on the 16-way system coming from duplicate data allocations.

When running the application in parallel, we observe a dramatic speedup downfall from 2P to 4P case. After carefully examing the profiling data and the kernel data structures, we come to know that it is because of the duplicate data allocation for the dictionary of CRFs model.

In traditional MPI programs, all the variables are allocated privately even if they contain the same contents. For the module IE, all the processes need to access the same dictionary of CRFs model. The easiest way to achieve this is to allocate a full copy of it in each process.

The required memory size for each copy of it is about 10MB, and thus in the 4P case it will need 40MB to accommodate all of them. In our 16-way system, every 4 processors share one L4 cache with the size of 32MB. So, if the dictionary is allocated duplicately, in 4P case, its total size will exceed that of L4 cache. Thus, some of the data accesses can not be satisfied in L4 cache and will be forwarded to main memory, which consumes more time.

At the same time, for 2P case, all the data (20MB) can be stored inside L4 cache and does not need main memory access. This explains the speedup downfall from 2P to 4P case.

To fix this problem, we create a process level shared memory with the help of IPC/shm and then map it to all the processes. Thus, there exists only one copy of the dictionary and it can be accommodated in L4 cache. Table 1 shows the effects on speedup of this approach (speedup of the 4P case is increased from 3.38X to 3.92X).

Table 1. Performance Improvements of IPC/shm Approach for Dataset "reuter" on the 16-way System

un-optimized	1P	4P	optimized	1P	4P
runtime (s)	146.27	43.27	runtime (s)	142.43	36.05
speedup	1	3.38	speedup	1	3.95

Another issue is the performance dropdown derived from frequent I/O requests on small files.

While processing the input documents, each process needs to read them in line by line. If the sequences are read in from disk separately, it will require more I/O accesses and then consume more time. Furthermore, while running in parallel, multiple concurrent I/O accesses will cause resource contentions and further decrease the performance.

First, this problem can be relieved by buffered I/O, to read the whole document into a pre-allocated buffer at one

time and then read line by line from the buffer. Second, we can accumulate some small input files into a bigger one while downloading them from the Internet. Both of these two efforts aim at diminishing the number of I/O operations and the effects of them can be seen from the speedup results of different datasets (as shown in Fig. 7).

To ease the problem of concurrent I/O accesses in multi-process case, we can adopt the techniques of parallel I/O [9] to do the help.

The last one is the load imbalance problem.

Because the processing time of each document is not proportional to its size, it is impossible to distribute the workloads statically. Table 2 shows the load imbalance ratios of static scheduling for dataset "beijing". In it, the load imbalance ratio equals the maximum runtime difference divided by the smallest runtime.

Table 2. Load Imbalance Ratios for Dataset "beijing" on the 16-way System

Speedup	1P	2P	4P	8P	16P
original	0.00%	48.89%	79.28%	125.34%	163.42%
optimized	0.00%	0.01%	0.03%	0.15%	3.35%

To solve it, we create a process level TaskQ [16] to distribute the tasks evenly. In implementation, by fully taking the advantage of the target shared memory architecture, we first create a shared task queue with the help of IPC/shm and then use IPC/sem [7] to synchronize the queue's accessing. With this approach, the load imbalance ratio can be decreased to within 5%.

4 Experiment and Performance Analysis

To characterize the performance of parallel IE running on shared memory architecture, we investigate the application from different aspects, including the memory hierarchy performance metrics and scalability performance (all the data shown in this section are collected by using VTune).

4.1 Memory Hierarchy Performance

4.1.1 Bandwidth and Memory Latency

Table 3 shows the FSB bandwidth utilization rate with different processor numbers on the two target shared memory multi-processor systems (with dataset "reuter").

Generally speaking, memory bandwidth is a key factor which may limit the speedup on multi-processor systems, especially for the shared-bus SMP system. From the results we can see, on both systems, the bandwidth requirement goes up steadily and is far from saturation (2.1GB/s for the 4-way system and 25.6GB/s for the 16-way system). Similarly, in Table 4, the memory latency on both systems remains flat with different processor numbers. In another aspect, it also confirms that the bandwidth is not saturated on both systems.

Table 3. FSB Bandwidth of module IE on both systems for Dataset "reuter" with Different Processor Numbers

MB/s	1P	2P	4P	8P	16P
4-way	91.4	177.9	353.2	N/A	N/A
16-way	84.2	166.8	339.4	679.2	1363.6

Table 4 also shows the memory latency of the 4-way system is larger than that of the 16-way. Partly, this is a benefit from the larger L3 cache and the shared L4 cache of the 16-way system.

Table 4. Memory Latency of module IE on both systems for Dataset "reuter" with Different Processor Numbers

clockticks	1P	2P	4P	8P	16P
4-way	358.94	372.15	387.75	N/A	N/A
16-way	191.60	194.91	210.17	196.95	192.33

4.1.2 Cache Misses

Fig. 4 shows the runtime comparison with dataset "reuter" on both systems. Why are the runtimes on the 4-way system longer than that on the 16-way system? The cache miss rate results shown in Fig. 5 and 6 reveal the reason.

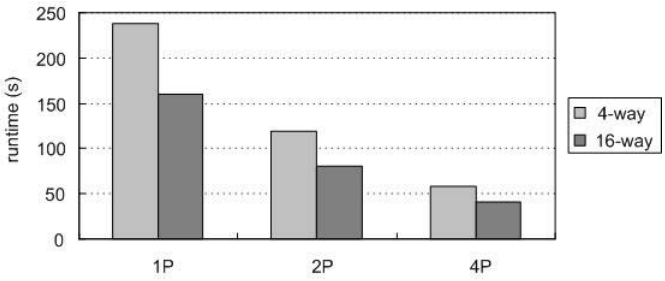


Figure 4. Runtime Comparisons of Module IE for Dataset "reuter" between the 4-way and the 16-way System

Fig. 5 depicts the cache miss rates for the whole cache hierarchy on the 16-way system. It shows a relative high

L2 cache miss rate, around 20%. This mainly comes from in-continuous data access caused by the "find" operation of *hash_map*.

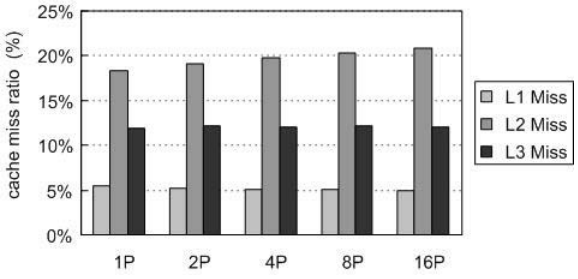


Figure 5. Cache Miss Rate of module IE for Dataset "reuter" on the 16-way System with Different Processor Numbers

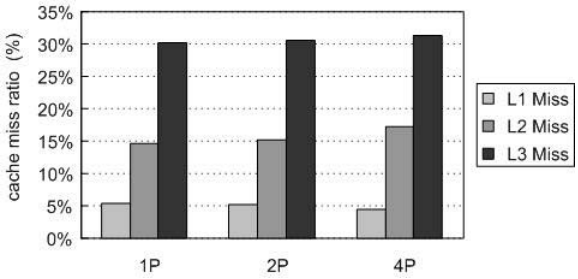


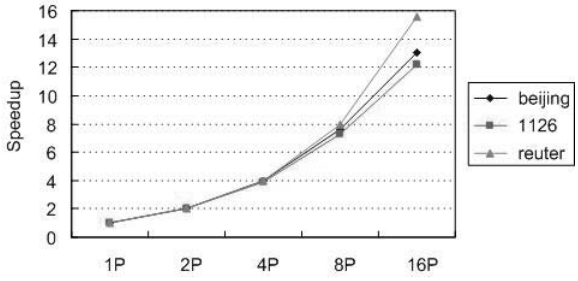
Figure 6. Cache Miss Rate of module IE for Dataset "reuter" on the 4-way System with Different Processor Numbers

Fig. 6 shows the cache miss rates on the 4-way system. The L1 and L2 cache misses are almost the same as those of the 16-way system. The noticeable difference comes from the L3 cache miss rate, where L3 cache misses nearly triple on the 4-way system. This is caused by the smaller L3 cache size.

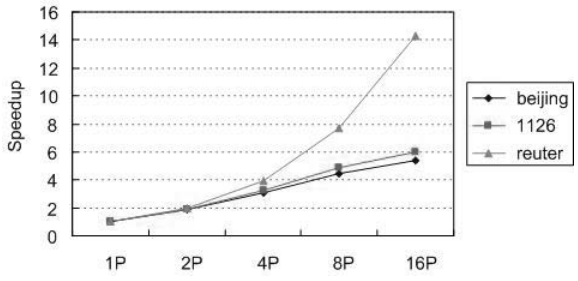
Typically, the penalty of L3 cache miss costs several hundreds of cycles to fetch the data from the main memory. Thus, on one hand, the higher L3 cache miss rate will incur more memory accesses than that of the 16-way system and cause performance dropdown. On the other hand, in the 16-way system each 4 processors share a large 32M combined L4 cache. This L4 cache can fill part of the L3 cache miss with smaller access latency, and reduce the penalty of L3 cache miss. These two aspects explain the runtime differences between the two target systems.

4.2 Scalability Performance

With the support of memory performance data, Fig. 7 depicts the overall speedups of DP only (a) and MI + DP (b) on the 16-way system for all the three datasets.



(a) Speedup of DP only



(b) Speedup of MI + DP

Figure 7. Speedup of Module IE on the 16-way System with Different Datasets

When only considering DP, the speedup curve of those two small datasets "beijing" and "1126", goes up linearly on 2, 4 and 8 processors, but starts deteriorating when all the 16 processors are used. This slowdown mainly comes from the reading requests for small files. Linux system reads in files from disk in blocks and also uses some read-ahead buffers to accelerate the process. If the size of each file is too small, it can not fully utilize the capacity of Linux and thus results in low efficiency [8].

For dataset "beijing" the average document size is around 14KB, and that of dataset "1126" is 8.5KB. The cost for each file loading (disk accessing) can not be concealed by the runtime of processing. On the contrary, dataset "reuter" has an average document size of 35KB and thus the file loading time can almost be neglected comparing with the processing time. In this experiment, the corresponding speedup performance of dataset "reuter" is near linear (15.6X for 16P case). This further explains the bad effects of frequent I/O requests for small files on parallel applications.

When counting in MI, the speedup performance of dataset "beijing" and "1126" are not so good (only $\sim$6X

for 16P case). This is because the runtime of MI takes a noticeable portion in multi-processor case. The operations in MI are tightly coupled and only have a small amount of dividable sub-tasks for distribution. Although, we have also made parallelization on MI, the speedup of MI is still very low (no more than 1.3X). However, this can be ignored when the input dataset is large enough (such as dataset "reuter").

5 Conclusion

Text Mining is a potential mass market desktop application in the many-core era. In it, Information Extraction is a basis module and can help improve the quality of other tasks in our Personal Text Mining System, such as clustering and summarization. In this paper, we present the optimization and parallelization of CRFs based IE and study its performance on shared memory multi-processor systems. The experimental results show that parallel IE scales pretty well on the target systems.

Besides the scalability performance, we can also draw the following conclusions.

First, from the programmer's perspective, we should choose the right programming model, data structures and use them efficiently to avoid unnecessary performance drops. In this work, we use MPI and System V. IPC/shm to support process level parallelism on shared memory architecture, which can avoid the heap contention problem of thread level parallelism. In the future's many-core system, there should a unified programming model, which is adaptable to the application's execution pattern.

Second, in regarding to the system, it proves that the larger L3 cache and the combined L4 cache can help improve the performance of such parallel applications. This can give hints to the architects of many-core system. On one hand, the larger cache can reduce the cache misses dramatically; on the other hand, the shared L4 cache (Last Level Cache) not only can reduce the penalty of L3 cache misses, but also can exploit the data-sharing behavior and reduce the bandwidth demands.

For future work, we will continue on study the performance characteristics of module IE as well as the other modules in our Personal Text Mining System. Also, system level parallelization and scheduling will be studied.

References

[1] Intel Vtune Performance Analyzer. In *http://developer.intel.com/software/products/vtune/*.

[2] LeapHeap. In *http:// www.leapheap.com*.

[3] Reuter News Collections. In *http://www.reuters.com*.

[4] The Hoard Multiprocessor Memory Allocator. In *http:// www.hoard.org*.

[5] The Lemur Toolkit for Language Modeling and Information Retrieval. In *http://www.lemurproject.org*.

[6] M. Bebbington. Tutorial - Viterbi Algorithm. In *Proc. of the First Workshop on Hidden Markov Models and Complex Systems*, 2005.

[7] M. Beck, H. Bohme, and M. Dziadzka. *Linux Kernel Programming*. Addison-Wesley, 2002.

[8] R. Bryant, R. Forester, and J. Hawkes. Filesystem Performance and Scalability in Linux 2.4.17. In *USENIX Annual Technical Conference*, 2002.

[9] G. E. Fagg. Message Passing with MPI Parallel IO. Technical report, ARC Georgetown University, 2003.

[10] P. Gelsinger. Microprocessors for the new millennium: Challenges, opportunities and new frontiers. In *ISSCC Tech. Digest*, pages 22–25, 2001.

[11] X. P. Hieu and M. L. Nguyen. FlexCRFs: A Flexible Conditional Random Fields Toolkit. In *http://www.jaist.ac.jp/ hieuxuan/flexcrfs/flexcrfs.html*.

[12] J. Lafferty, A. McCallum, and F. Pereira. Conditional Random Fields: Probabilistic Models for Segmenting and Labeling Sequence Data. In *Proc. ICML-01*, pages 282–289, 2001.

[13] Y. Liu, E. Shriberg, A. Stolcke, and M. Harper. Using Conditional Random Fields for Sentence Boundary Detection in Speech. In *Proc. of the 43rd Annual Meeting of the ACL*, pages 451–458, 2005.

[14] F. Sha and F. Pereira. Shallow Parsing with Conditional Random Fields. In *Proc. of the 2003 Human Language Technology Conference and North American Chapter of the Association for Computational Linguistics (HLT/NAACL-03)*, 2003.

[15] J. Shan. Scalability Issues of STL Containers in Multithread Programs. Technical report, ICRC, Intel Corp., 2005.

[16] E. Su, X. Tian, and M. Girkar. Compiler Support of the Workqueuing Execution Model for Intel SMP Architectures. In *Proc. of the Fourth European Workshop on OpenMP (EWOMP 2002)*, 2002.