Intel Xeon Phi 에서의 Aho-Corasick 알고리즘을 위한 메모리 친화적인 고성능 병렬화

쟌 느앗 프엉, 정요상, 이명호*
명지대학교 컴퓨터공학과, 경기도 용인시 처인구 남동 산 38-2
e-mail: myunghol@mju.ac.kr

요 약

Aho-Corasick (AC) 알고리즘은 실시간 성능을 요하는 많은 응용 분야에 적용되는 스트링 매칭 알고리즘으로서, 한번에 여러 개의 패턴들을 동시에 매칭시키는 것이 가능하다. 본 논문에서는 Intel 의 Many Integrated Core (MIC0 아키텍쳐인 Xeon Phi 칩 상에서 AC 알고리즘을 병렬화한다. 이를 위하여 AC 알고리즘에서 입력 데이터에 대하여 여러 개의 패턴들을 동시에 매칭시키는 데에 사용되는 Deterministic Finite Automaton 구조를 압축시키는 새로운 기법을 제안한다. 이 기법은 캐시 미스를 감소시켜서 Xeon Phi 상에서 AC 알고리즘의 성능을 크게 향상시킨다.

Memory-Efficient High Performance Parallelization of Aho-Corasick Algorithm on Intel Xeon Phi

Nhat-Phuong Tran, Yosang Jeong, Myungho Lee*
Dept. of Compute Science and Engineering, Myongji University
38-2 San Namdong, Cheo-In Gu, Yong In, Kyung Ki Do, Korea

Abstract

Aho-Corasick (AC) algorithm is a multiple patterns string matching algorithm commonly used in many applications with real-time performance requirements. In this paper, we parallelize the AC algorithm on the Intel's Many Integrated Core (MIC) Architecture, Xeon Phi Coprocessor. We propose a new technique to compress the Deterministic Finite Automaton structure which represents the set of pattern strings again which the input data is inspected for possible matches. The new technique reduces the cache misses and leads to significantly improved performance on Xeon Phi.

1. Introduction

AC algorithm is a multiple patterns string matching algorithm commonly used in computer and network security, bioinformatics, among many others. In order to meet the highly demanding performance requirements imposed on these applications, achieving high performance for the AC algorithm is crucial.

Recently, Intel introduced a Many Integrated Core (MIC) architecture chip called Intel Xeon Phi Coprocessor. This coprocessor provides high performance with low power consumption. Programmers can use the existing parallel APIs such as OpenMP, Pthreads, MPI [9] with a bit of wrapper codes to launch the kernel codes parallelized in OpenMP, for example, on the Xeon Phi.

In this paper, we parallelize the AC algorithm on the Intel Xeon Phi. In our parallelization, we propose a simple but efficient compression technique for the Deterministic Finite Automaton structure which represents the set of pattern strings again which the input data is inspected for possible matches. Our technique eliminates the unused columns in the automaton and reduces cache misses significantly. Thus leads to significantly increased performance. Experimental results show that our technique delivers up to 2.28-times speedup compared with the original full DFA.

In the following sections, we present briefly the architecture and programming environment of the Intel Xeon Phi Coprocessor (Section 2), the AC algorithm (Section 3), our parallelization and the DFA compression technique (Section 4), performance evaluation results (Section 5), and finally conclude the paper (Section 6).

2. Many Integrated Core (MIC) Architecture and Its Programming

Intel Xeon Phi 5120D, which is commonly known as the MIC-2 (or Xeon Phi) chip, has 60 cores clocked at 1053 Mhz. These cores are connected through a high performance bidirectional ring interconnect. MIC has 16 memory channels delivering up to 5GT/s. Each core offers four-way simultaneous multi-threading (SMT) and 512-bit wide SIMD vectors, which corresponds to eight double precision or sixteen single precision floating point numbers. Each core has a 32KB L1 data cache, a 32 KB L1 instruction cache and 512KB L2 cache. Thus, in total 60 cores has 30 MB L2 cache. L2 cache is fully coherent using the hardware. The total size of the on-board system memory is 8GB. MIC is connected to the host CPU through the PCI Express bus.

The Xeon Phi is used in either the native mode or the

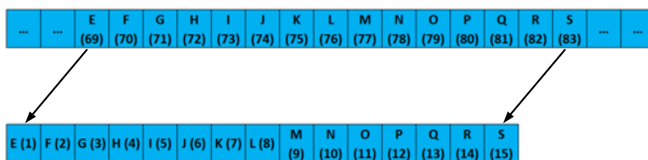
offload mode. In the native mode, the application runs directly on Xeon Phi. In order to use this mode, the application is compiler using *–mmic* option. In the offload mode, the application runs on the host side and only the selected regions (compute density region) of the code is offloaded to the Xeon Phi coprocessor. Besides, parallel programming APIs such as Pthreads, OpenMP, Intel Cilk Plus, and OpenCL can be used to program the applications to run on the Xeon Phi.

3. AC Algorithm

The AC algorithm is a multiple pattern matching algorithm which can match multiple patterns simultaneously for a given finite set of string (or dictionary). The AC algorithm consists of two phase. In the first phase, a pattern matching machine called the AC automaton is constructed from a finite set of patterns. In the second phase, the constructed AC machine is used to find locations of the patterns in the given input text [1]. The AC automaton invokes three functions: a goto function *g*, a failure function *f*, and an output function *output*. Depending on whether the failure function is used separately or combined with other functions, we can construct the Deterministic Finite Automata (DFA) or the Non-deterministic Finite Automata (NFA) correspondingly to represent the given set of pattern strings. In this work, we use the DFA which can be represented as a 2-dimensions matrix (called State Transition Table). The rows represent the states in the DFA and the columns represent the input characters. Suppose that we have 256 input characters (mapped to 256 characters of the extended ASCII table), the STT needs 257 columns for the characters and one extra column indicating if the current state is a matched state.

4. Our Parallelization and DFA Compression Technique

In order to parallelize the AC algorithm, we divide the input text into many chunks. Each chunk is assigned to each thread. Multiple threads perform pattern matching operations while referring to the STT. On Xeon Phi, multiple threads are mapped to the cores and the four hardware threads in each core. The assignment, however, incurs a problem when a pattern overlays between two or more consecutive input chunks. In order to solve this problem, we span each thread by adding X characters after the chunk that it is assigned, where X is the maximum pattern length in the set of patterns.



(Figure 1) Removing unused columns in the STT

As explained in Section 3, the DFA has 257 columns for the extended ASCII character set and 1 column indicating if the current state is a matched state. However, in most text matching applications, not all of 256 characters are used in the extended ASCII table. Thus, there are a lot of columns with value 0 in the STT which significantly increase the memory size to store the STT. In this paper, we attempt to minimize the number of columns. Assume that the given set of patterns is {he, she, his, hers}. The characters in this set ranges from ‘E’ to ‘S’. Thus we define the longest sequence for the above set of patterns as ‘E’ to ‘S’. Although characters ‘F’, ‘G’, ‘J’, ‘K’, ‘L’, ‘M’, ‘N’, ‘O’, ‘P’, and ‘Q’ are not used in the patterns, we still include these characters to sequence.

Consequently, using our STT compression technique, the ‘E’ character sits in the first column. The position of the ‘E’ column in the original uncompressed STT is 69 (correspond to the position in the extended ASCII table). The new position of the character can be calculated using the following formula: *new_pos = old_pos – sequence_first_pos* + 1. The reduced STT size leads to much smaller number of cache misses. Thus, it improves the performance significantly. Figure 2 shows the algorithm for matching patterns using the new STT.

```
1 int map_function(int old_pos, int first_sequence_pos)
2 {
3       return old_pos – first_sequence_pos + 1;
4 }
5 void new_search(char *s, unsigned long length, int **new_stt)
6 {
7       for(i=0; i<length; i++)
8       {       new_x = map_function(s[i]);
9               if(new_x>=1 and new_x<=last_sequence_pos)
10                      search in new_stt;
11      }
12 }
```

(Figure 2) Algorithm for matching characters in the new STT

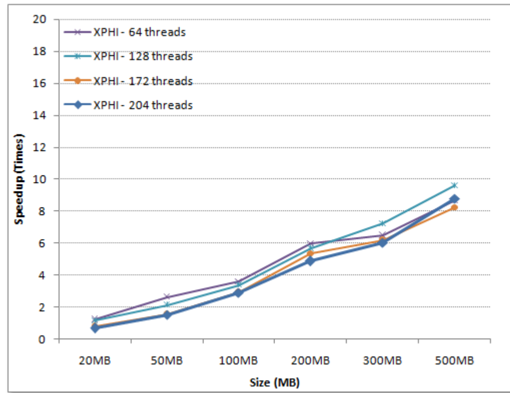
5. Experimental Results

We implemented the parallelization and the STT compression technique for the AC algorithm as described in Section 4 on the Intel Xeon Phi 5120D. We conducted two experiments:

- V1: serial execution of the AC algorithm on the 6-core host microprocessor, Intel Xeon E5-1620 with 15 MB cache
- V2: parallelization on the Xeon Phi using OpenMP and execution in the offload mode

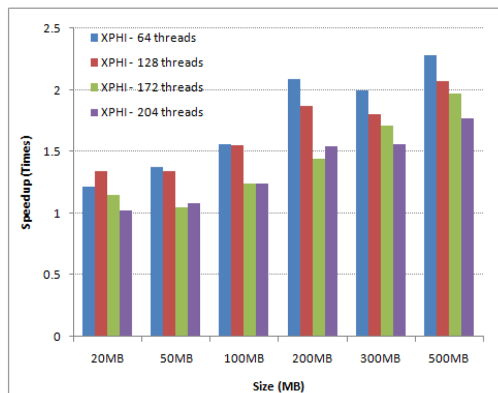
In order to measure the performance of each version, we conducted experiments using different input lengths and different number of patterns. The numbers of patterns used are 100, 5000, and 50000. The input lengths used are 20MB, 50MB, 100MB, 200MB, 300MB and 500MB. In all experiments we conducted, we ignore the time spent in the construction phase of the STT (in serial version also) which run on single CPU core. In our opinion, this is fair because the STT construction is performed only once for a given finite set of strings, whereas the pattern matching process is performed a large number of times.

Figure 3 shows the speedups of the AC algorithm on the MIC (V2) compared with the V1. The speedup steadily improves over the increased input data sizes. The best performance was observed when 128 threads were used. Beyond 128 threads, the performance levels off.



(Figure 3) Speedup of V3 compared with V1 using different input data sizes and fixed numbers of patterns = 5000

We also conducted experiments to compare the performance of the compressed STT versus using the full STT. Figure 4 shows the speedup of the compressed STT over the original full STT on the Xeon Phi. The compressed STT runs faster by 1.019 – 2.28-times. The speedup is larger for the small number of threads, because using small number of threads the effects of the multithreading to hide the cache miss latency is low. Thus the STT compression is more effective.



(Figure 4) Speedup of the compressed STT over the full-STT using different input sizes and fixed numbers of patterns = 50,000

6. Conclusion

In this paper we parallelized the AC algorithm on the Intel Xeon Processor and the Intel Xeon Phi Coprocessor, in particular, using our parallelization and the STT compression technique. Experiments show that the parallelized AC algorithm shows good scalablilty on the Intel Xeon Processor. Also, our new technique to compress the STT removes the unused columns in STT and leads to the speedup in the range of 1.019 – 2.28 compared with the original full STT.

7. Acknowledgement

This research was supported by Basic Science Research Program through the National Research Foundation of Korea (NRF) funded by the Ministry of Education, Science, and Technology (Grant No: 2012R1A1A2042267).

References

[1] A.V. Aho and M.J. Corasick, “Efficient string matching: An aid to bibliographic search”, Communications of the ACM, vol. 20, Session 10, Oct. 1977, pp. 761–772.
[2] Marc Norton, “Optimizing Pattern Matching for Intrusion Detection”, http://docs.idsresearch.org/OptimizingPatternMatchingForIDS.pdf, July 2004
[3] Soumya Sen, “Performance Characterization and Improvement of Snort as an IDS”
[4] NVIDIA, “CUDA Best Practices Guide: NVIDIA CUDA C Programming Best Practices Guide – CUDA Toolkit 4.0”, May, 2011.
[5] J. Jeffers and J. Reinders, “Intel Xeon Phi Coprocessor High Performance Programming”, Morgan Kaufmann, Walthman, MA, USA, 2013