J. Parallel Distrib. Comput. 76 (2015) 49–57

Contents lists available at ScienceDirect

J. Parallel Distrib. Comput.

journal homepage: www.elsevier.com/locate/jpdc

A bit-parallel algorithm for searching multiple patterns with various lengths

Ko Kusudo [a,b], Fumihiko Ino [a,*], Kenichi Hagihara [a]

[a] Graduate School of Information Science and Technology, Osaka University, 1-5 Yamada-oka, Suita, Osaka 565-0871, Japan
[b] Fujitsu Limited, 1-5-2 Higashi-Shimbashi, Minato-ku, Tokyo 105-7123, Japan

HIGHLIGHTS

- We present an extension of a bit-parallel algorithm for fast string search.
- The algorithm maximizes the stability of search throughput for multiple patterns of different lengths.
- Bit and data parallelism are exploited via AVX2 and OpenMP, respectively.
- Rapid identification of matching patterns is realized by a data padding scheme that regularizes control flow.
- Stable search throughput is achieved for arbitrary text and patterns that fit into a word.

ARTICLE INFO

Article history:
Received 24 March 2014
Received in revised form
29 October 2014
Accepted 4 November 2014
Available online 15 November 2014

Keywords:
String search
Bit-parallel algorithm
Acceleration
AVX

ABSTRACT

In this paper, we present an Advanced Vector Extensions (AVX) accelerated method for a bit-parallel algorithm that realizes fast string search for maximizing stable search throughput. An advantage of our method is that it accelerates string search by regularizing both control flow and data structures. This regularization facilitates the exploitation of the latest vector instruction set to achieve efficient parallel search of multiple patterns of different lengths. We use AVX instructions to increase search throughput per CPU core and employ OpenMP directives to realize data-parallel search of strings. As a result, we found that our data structure doubled search throughput as compared with a previous bit-parallel approach that used a data structure for patterns of the same length. We also found that our method achieved stable search throughput for arbitrary data if the pattern size is large, but small enough to fit into a word. Some experimental results are provided to understand the advantage and disadvantage of our method with a comparison to Aho–Corasick based methods. We believe that our method is useful for large genome texts with many partial matches.

1. Introduction

String search identifies the location in a large string or *text* at which one or more substring *patterns* appear. Numerous research areas require the acceleration of multipattern search to handle large volumes of real-time data [19,6]. For example, network intrusion detection systems monitor packets flowing on the network to protect computer systems from malicious access [33,32]. String search is also useful for locating specific amino acid sequences in biological databases [30,25,8]. Given performance requirements of such applications, a sophisticated efficient parallel implementation is required to achieve string search acceleration. However, scaling search throughput with the number of processing elements is not easy, because string search can be regarded as an irregular problem that suffers from burdensome issues such as irregular control flow and unpredictable data access. In particular, solving these issues is essential to efficiently parallelize string search on a single instruction, multiple data (SIMD) parallel machine, because different control paths can result in SIMD serialization, which drops the efficiency of parallel execution. Therefore, accelerating string search is a challenging problem for the parallel processing community.

Many researchers [33,41,31] tried to parallelize the Aho–Corasick (AC) algorithm [1], which simultaneously searches multiple patterns to reuse loaded symbols between multiple patterns. Because string search is a memory-intensive problem rather than a computationally intensive problem, this data reuse approach is

* Corresponding author.
E-mail address: ino@ist.osaka-u.ac.jp (F. Ino).

http://dx.doi.org/10.1016/j.jpdc.2014.11.003
0743-7315/

useful to achieve acceleration. The AC algorithm represents multiple patterns as a trie data structure, which is regarded as a deterministic finite automaton (DFA) that detects a match by loading symbols from the beginning of the text. If there is no valid state transition for the input symbol, then the automaton detects a transition failure and performs backtracking to investigate the possibility of a match from other locations. The time complexity of the AC algorithm is given by $\mathcal{O}(n + \hat{m} + z)$, where n is the length of the text, $\hat{m}$ is the sum of the lengths of the patterns, and z is the total number of occurrences of the patterns [1]. The AC algorithm can be easily implemented on a multiple instruction, multiple data (MIMD) parallel machine by exploiting the data parallelism inherent to string search. This data-parallel AC (DPAC) algorithm divides the text into chunks, allowing threads to process them in parallel. Thus, MIMD machines are more tolerant to the irregularity than SIMD machines, because the former machines can issue different instructions to different processing elements. The DPAC algorithm was deployed on many parallel machines such as the graphics processing unit (GPU) [32,30,41,31,28,29,34], field programmable gate array (FPGA) [14,18], Cell Broadband Engine [36], and supercomputer [35]. Similar to the AC algorithm, which has the cost $\mathcal{O}(z)$ of printing the output, the DPAC algorithm has different time complexities for best and worst cases. Furthermore, owing to irregular control flow that affects branch prediction accuracy and cache hits, search throughput can vary according to the number of matching strings [31,35].

The Parallel Failureless Aho–Corasick (PFAC) algorithm [15] extends the AC algorithm [1] to achieve efficient parallelization on a GPU [17]. As compared with CPUs, GPUs not only have higher peak memory bandwidths but also more processing cores. Such rich computational resources are useful for accelerating a memory-intensive problem. The PFAC algorithm creates a thread for every symbol in the text to identify patterns starting at any location. Each thread manages the current status of a deterministic finite state machine. Consequently, search throughput decreases if each thread suffers from multiple state transitions in its state machine. The time complexity for a thread of PFAC ranges from $\mathcal{O}(1)$ to $\mathcal{O}(M)$, where M denotes the longest length of the patterns to be searched simultaneously [15]. Therefore, performance degradation occurs if the text has many partially matching patterns and their matching lengths are relatively long. Such unstable search throughput is not desirable for packet and genome analyses, which must often process high volumes of data. Further, GPUs have smaller memory capacity than CPUs; consequently, maximum search throughput can be restricted by the peak bandwidth of the PCI Express bus, which can diminish the benefits of high-bandwidth video memory.

Another acceleration approach is to exploit bit parallelism in string search. An advantage of the bit-parallel algorithm [3] is that the number of memory references is determined only by the text and pattern lengths. In other words, the bit-parallel algorithm has the same best- and worst-case time complexity, so that it can provide highly robust throughput when faced with various patterns varying in the text and pattern contents. The bit-parallel algorithm was extended by Prasad et al. [23,24] to simultaneously search multiple patterns of the same length.

In this paper, to realize fast string search for maximizing stable search throughput, we present a high-throughput method that accelerates a bit-parallel algorithm on a multicore CPU. Our method extends Prasad's algorithm [23,24] such that it simultaneously searches multiple patterns of different lengths. In particular, our method is unique in regularizing both control flow and data structures for string search, namely an inherently irregular problem. This regularization facilitates the exploitation of the latest SIMD instructions that are useful to maximize the performance on a CPU. Our method efficiently searches multiple patterns of different lengths by using a data padding scheme that hides the irregularity of pattern lengths. This scheme is integrated into a two-level parallel algorithm that exploits not only data parallelism via OpenMP directives [4] but also bit parallelism via the latest vector instruction set called Advanced Vector Extensions (AVX) 2 [10]. The latter increases search throughput on a CPU core, while the former increases search throughput on a CPU socket. Our CPU-based solution demonstrates competitive search throughput without using special hardware devices such as the GPU and FPGA.

In addition to this introduction, this paper is organized as follows. Section 2 presents related studies in the area of string search. Section 3 summarizes the bit-parallel algorithm. Section 4 presents our proposed method, and shows how it increases the search throughput of the bit-parallel algorithm. Section 5 presents experimental results, and Section 6 provides the conclusions and suggestions for future work.

2. Related work

Table 1 shows a comparison of previous string matching implementations with their deployed hardware. Prasad et al. [23,24] extended the bit-parallel algorithm [3] to search multiple biological patterns in the text simultaneously. Their algorithm assumes that all simultaneously searched patterns have the same length and a word is large enough to store the patterns. Because the current x64 architecture uses 64-bit words, the total length of simultaneous patterns must therefore be less than 64 symbols. In contrast, our AVX-based algorithm covers multiple patterns of up to 256 symbols in length and accepts simultaneous patterns of different lengths. Our algorithm also exploits data parallelism via OpenMP directives, as detailed in Section 4. Xu et al. [38] implemented Prasad's algorithm on a GPU and achieved a search throughput of 0.1 Gbps. A similar bit-parallel algorithm was presented by Yadav et al. [39]. Bit-parallel algorithms have a disadvantage in terms of the pattern size. The total length $\hat{m}$ of patterns must be less than the word size.

Külekci [12] presented a filter-then-search algorithm called Streaming SIMD Extensions filter (SSEF) for searching a single pattern. The SSEF algorithm reduced time complexity by detecting possible matches at low cost, which were then given to the succeeding verification process. This algorithm was implemented using Streaming SIMD Extensions (SSE) instructions [9] to accelerate filtering process for single pattern matching. The SSEF algorithm was ten times faster than the bit-parallel length independent matching (BLIM) algorithm [13], which overcame the word size limitation of the bit-parallel algorithm. However, the time complexity of the SSEF algorithm ranges from $\mathcal{O}(nm)$ to $\mathcal{O}(n/m)$ according to preprocessing effects, where n is the length of the text and m is that of the pattern. The SSEF algorithm was extended by Faro and Külekci [5] for multiple pattern matching. Their achieved search throughput ranged from 0.4 to 0.6 Gbps on a Core i7 processor.

Oh and Ro [22] presented multi-threaded multiple pattern matching with suffix grouping (MTMP-SG), which extended the Wu–Manber algorithm [37] to overcome performance degradation that appears when the text includes many matching patterns. Their extended algorithm groups the target patterns in terms of their suffixes, and distributes the patterns over multiple threads. Their CPU-based implementation ran at a throughput of up to 0.4 Gbps, which was faster than a GPU-accelerated AC implementation when searching more than 5000 patterns simultaneously. They concluded that their implementation could be accelerated using SIMD instructions, which we tackled in the present work.

The PFAC algorithm [15] extended the AC algorithm to achieve efficient parallelization on a manycore GPU. To realize this, the PFAC algorithm eliminated the backtracking procedure of the AC