Accelerating DNA Sequence Analysis using Intel® Xeon Phi™

(PBio at ISPA-2015, ©IEEE)

Suejb Memeti and Sabri Pllana
Linnaeus University, Department of Computer Science,
351 95 Växjö, Sweden
Email: {suejb.memeti, sabri.pllana}@lnu.se

***Abstract*—Genetic information is increasing exponentially, doubling every 18 months. Analyzing this information within a reasonable amount of time requires parallel computing resources. While considerable research has addressed DNA analysis using GPUs, so far not much attention has been paid to the Intel Xeon Phi coprocessor. In this paper we present an algorithm for large-scale DNA analysis that exploits thread-level and the SIMD parallelism of the Intel Xeon Phi. We evaluate our approach for various numbers of cores and thread allocation affinities in the context of real-world DNA sequences of mouse, cat, dog, chicken, human and turkey. The experimental results on Intel Xeon Phi show speed-ups of up to $10\times$ compared to a sequential implementation running on an Intel Xeon processor E5.**

I. Introduction

There is a growing interest in molecular biology community to understand the information that is encoded within the Deoxyribonucleic Acid (DNA) sequences of each organism [1]. A DNA sequence contains specific genetic instructions that make the living organisms function in a proper way. The four basic building blocks (also known as *nucleotide bases*) of a DNA sequence are: *Adenine* (A), *Cytosine* (C), *Guanine* (G) and *Thymine* (T).

Discovery of differences and similarities of organisms and exploration of the evolutionary relationship between them, often require comparisons of the corresponding DNA sequences. Examples include: checking whether one sequence is a subsequence of another, or finding a sub-sequence that appears in the same order in both DNA sequences [2]. The process of searching for certain sub-sequences of length *k*, so called *k-mers*, is performed with pattern matching algorithms.

According to Benson et al. [3] the number of DNA sequences and nucleotide bases in these sequences is doubling every 18 months. Real-world DNA sequences comprise several Gigabytes and the process of extracting the important information demands the adequate use of parallel computing resources to be completed within a reasonable time. A quick DNA analysis may have a decisive role in many applications including: preventing the evolution of different viruses and bacterias during an early phase [4]; early diagnosis of genetic predispositions to certain diseases (such as, cancer, cardiovascular diseases,..) [5]; and DNA forensics (such as, parentage testing, or criminal investigation) [6].

Related research has addressed extensively pattern matching algorithms for GPUs. Lin et al. [7] proposed the Parallel Failure-less Aho-Corasick algorithm for pattern matching on GPUs. Kouzinopoulos and Margaritis [8] show speedup of up to $24\times$ for small input text and pattern sizes for different algorithms on GPU. Bellekens et al. [9] presented a parallel implementation of the Knuth-Morris-Pratt algorithm using the Nvidia GPU hardware. Tumeo and Villa [10] proposed an implementation of the Aho-Corasick algorithm for DNA analysis applications on clusters with GPUs. In comparison to GPUs, besides the ability to provide high performance, the Xeon Phi deserves our attention because of programmability [11], [12] and portability [13]. However, so far not much research was focused on DNA analysis using pattern matching algorithms designed specifically for the Xeon Phi.

In this paper, we present a parallel algorithm for DNA analysis that is designed to exploit the thread-level and the SIMD parallelism available in the Intel Xeon Phi coprocessor. Our pattern matching algorithm is based on finite automata. For thread-level parallelism we use a domain decomposition approach that splits the DNA sequence into chunks evenly among the available threads. To process the patterns occurring in the cross border of sequence chunks, our algorithm uses $m-1$ overlapping characters, where m is the pattern length. With respect to the SIMD-parallelism our algorithm implementation uses the potential of the 512-bit vector registers of the Intel Xeon Phi architecture for transition function of finite automata. We evaluate our approach experimentally with real-world DNA sequences of different living species. For the human DNA sequence a speedup of up to $10\times$ is achieved compared to the sequential version running on Intel Xeon processor E5. Major contributions of this paper include,

- an algorithm for large-scale DNA analysis that is designed for Intel Xeon Phi;
- an experimental evaluation of our DNA analysis algorithm for real-world DNA sequences of mouse (2.7GB), cat (2.4GB), dog (2.4GB), chicken (1GB), human (3.2GB) and turkey (0.2GB);
- a discussion of the state-of-the-art in pattern matching and DNA analysis using many-core architectures (GPUs, Intel Xeon Phi).

The rest of the paper is organized as follows. Section II provides background information with respect to pattern matching and introduces the Intel Xeon Phi architecture. Our

parallel algorithm for DNA analysis using the Intel Xeon Phi coprocessor is described in Section III. Section IV presents the experimental setup and discusses the experimental results. The work described in this paper is compared and contrasted to the state-of-the-art related work in Section V. Section VI provides a summary of this paper.

II. Background

In this section we first provide background information with respect to the pattern matching with finite automata. Thereafter, we present the major features and the architecture of the Intel Xeon Phi coprocessor.

A. Pattern Matching with Finite Automata (FA)

Finding occurrences of a pattern in a text is a frequent need of many text-editing programs (e.g. find-replace functions), Internet Search Engines (e.g. finding web-pages that are relevant to the provided query), or lexical analyzers (e.g. determining the locations of a pattern within a sequence of tokens). In the context of computational biology, pattern matching algorithms are used for analyzing and processing genetic information by searching for particular patterns in DNA sequences. Formally, in DNA analysis the string matching problem can be expressed as follows: the input text (DNA sequence) is an array $T[1..n]$ where n is the length of the DNA sequence, and pattern $P[1..m]$ where the length of the pattern $m \leq n$. The finite alphabet $\sum$ defines the possible characters of T and P, in this case $\sum = \{A, C, G, T\}$, where each letter corresponds to one of the four nucleotide bases [14].

A Finite Automata (FA) is a simple machine for processing information, which scans the input text T in order to find the occurrences of the pattern P. FA is an efficient technique for pattern matching, because it examines each character from T exactly once. Formally, FA is a quintuple of $(Q, \sum, \delta, q_0, F)$, where Q is a finite set of states, $\sum$ is a finite input alphabet, δ (*transition function*) is the function $Q \times \sum \rightarrow Q$, q_0 is the start state and F is a distinguished set of accepting states [14].

A well known algorithm for detecting any exact occurrences (including the overlapping ones) of multiple patterns is the Aho-Corasick (AC) algorithm [15]. Because of the capability to deliver input-independent performance, we use the AC algorithm as a basis of our algorithm for counting and extracting patterns from a DNA sequence. The AC algorithm builds an automaton by creating states and transitions corresponding to the states. The automaton is able to match multiple and overlapping occurrences, by adding a failure transition when there is no regular transition leaving from the current state. The failure transitions, known as ϵ-transitions, do not consume any input, which make the automaton non-deterministic.

B. Intel® Xeon Phi™ Architecture

The Intel Xeon Phi (codenamed Knights Corner) is a many-core shared-memory coprocessor, which runs a lightweight Linux Operating System. In this paper we use the Intel Xeon Phi coprocessor 7120P. Figure 1 depicts the architecture of our platform, where in the left-hand side is the host comprising one or more Intel x86 CPUs, whereas the right-hand side depicts the Intel Xeon Phi architecture. The Xeon Phi comprises 61 x86 cores, each running at 1.2 GHz base frequency with the max turbo frequency 1.3 GHz [16]. Each core has four hardware threads, in total there are 244 hardware threads per coprocessor, capable of delivering performance up to two TeraFLOP/s at single precision or one TeraFLOP/s at double precision. Each core has a private L2 cache of 512KB that is kept fully coherent by a global-distributed tag directory (TD). The L2 caches are connected through a bidirectional ring bus interconnect, which forms a unified shared L2 cache of 30,5MB. In addition to the cores, there are 16 memory channels, which theoretically deliver up to 352 GB/s memory bandwidth. The memory controllers (GDDR MC) and the PCIe Client Logic provide a direct interface to the GDDR5 memory and the PCIe bus, respectively. The host communicates with the coprocessor through the PCIe bus which is limited to 8GB/s transfer bandwidth. The PCIe bus is a bottleneck for the offload programming model, where data has to be transferred from the host to the coprocessor and vice versa. In order to achieve high offload computational performance, it is recommended that the data is transfered to the coprocessor and kept there (reused) to avoid memory bandwidth bottlenecks while moving the data back and forth.

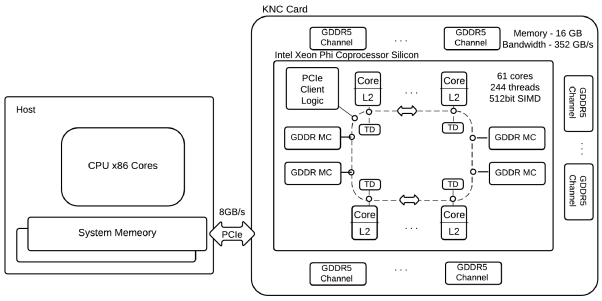


Fig. 1. The Xeon Phi Architecture

An important aspect of the coprocessor is its vector processing unit, which feature Intel Advanced Vector Extensions (AVX) 512-bit SIMD instruction set. Thus it can execute 16 single-precision (16 wide $\times$ 32 bit) or 8 double-precision (8 wide $\times$ 64 bit) operations per cycle. Exploiting the vector units in an efficient way is one of the key aspects in achieving high performance on Intel Xeon Phi Coprocessor [17].

III. Design and Implementation of an Algorithm for DNA Analysis using Intel Xeon Phi

The key features of our algorithm (Section III-A) and implementation for parallel DNA analysis on Intel Xeon Phi are: (1) decomposition of the input DNA sequence across the available threads, (2) exploiting the SIMD parallelism, and (3) reducing the memory references using a suitable representation for the State Transition Table (Section III-B).

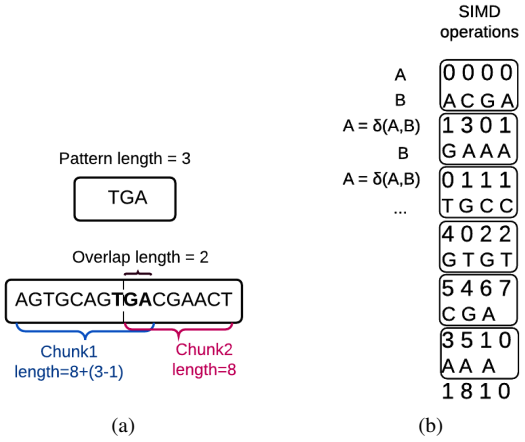
A. Parallel DNA Analysis Algorithm

Fig. 2a illustrates our parallelization strategy that is based on domain decomposition, which means the input DNA sequence

Algorithm 1 Parallel DNA Analysis

Input: STT, final state f, input T, number of threads p, vector length v, pattern length m
Output: Count of pattern matches and their location

1: **procedure** AC(dfa, f, T, p, v, m)
2: $n = T.length$
3: $chunkLength = n/p$
4: $vChunkLength = (chunkLength+m-1)/v+m-1$
5: **for** $i = 1$ **to** p **do**
6: $q[v]$▷ store the current state for each SIMD chunk
7: $chunkStart = i * chunkLength$
8: **for** $j = 1$ **to** $vChunkLength$ **do**
9: $vStart = j * chunkStart$
10: **for** $k = 1$ **to** v **do** ▷ this loop is vectorized
11: $c = v * k + vStart$
12: $q[k] = \delta(q[k], T[c])$ ▷ load the next node from dfa, by following the transition from the current node $q[k]$ labeled by the symbol $T[c]$
13: **if** $q[k] \geq f$ **then** ▷ check if the transition to next node is final
14: **print** matching pattern at position c
15: **end if**
16: **end for**
17: **end for**
18: **end for**
19: **end procedure**



(a) (b)

Fig. 2. Thread-level and SIMD parallelism; (a) splitting the DNA sequence into chunks; (b) vectorization of the transition function.

is evenly split into chunks among the available threads (Alg. 1, lines 5 – 18). While splitting the input, there is a risk of not being able to match the occurrences of patterns that cross the chunks boundaries. Other researchers have addressed this issue by using speculation based on most visited states [18], by using Suffix-Arrays [19] or using an intersection of successor states and predecessor states of the FA [20]. In this paper we find these occurrences by overlapping the input chunks by $m-1$ characters (Alg. 1 Line 4), where m is the pattern length. This method is applicable to multiple patterns with equal length, otherwise it can happen that two threads match the same pattern with a length shorter than $m-1$.

Our algorithm exploits vector units of Intel Xeon Phi, by splitting the chunks further into v parts where v represents the vector length (Alg. 1, lines 10 – 16). The operations (such as, determining the next state (Alg. 1, Line 12), or checking if the next state is a final one (Alg. 1, line 13)) are performed on multiple data points simultaneously.

Fig. 2b illustrates the SIMD operations assuming that the input is the same as in Fig. 2a and the vector length is 4. First we create an array of v elements (Alg. 1, line 9), where each element starts from state q_0. The first SIMD δ operations are performed on the characters at positions 0, 4, 8, and 12, the second SIMD operations are performed on characters at position 1, 5, 9, and 13, and so on. The SIMD loop (Alg. 1 Line 10) that performs the δ operations is going to be executed $T.length/v + m - 1$ times. The estimated speedup according to the vectorization reports (Fig. 3) is 2.6× compared to the scalar δ function.

```
remark #15475: --- begin vector loop cost summary ---
remark #15476: scalar loop cost: 36
remark #15477: vector loop cost: 12.930
remark #15478: estimated potential speedup: 2.600
remark #15479: lightweight vector operations: 42
remark #15487: type converts: 3
remark #15488: --- end vector loop cost summary ---
```

Fig. 3. Vectorization report

B. Implementation Aspects

The Aho Corasick (AC) with failure links algorithm has a drawback due to its non-deterministic transitions for a single character. Our solution to this issue is illustrated with an example in Figure 4. Our improved AC automaton finds the right transition (indicated by a dashed line) for each state, thus eliminating the failure transitions. Having a valid transition for each possible symbol to another state in the automaton, guarantees that for each character there is always the same number of operations to be performed.

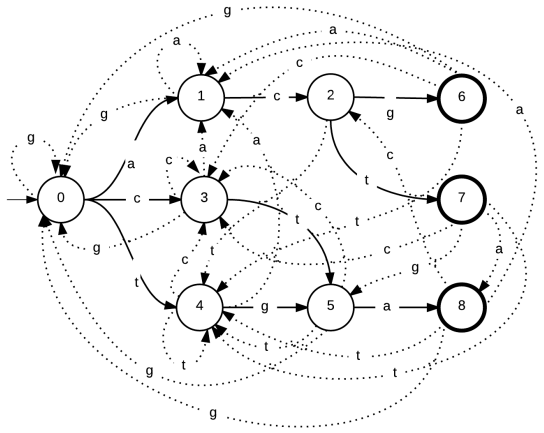


Fig. 4. An example of our improved AC automaton that matches occurrences of the following patterns: *acg, act, cta,* and *tga.*

Checking whether the *next state* is a final one requires to store the final states in a set, and then perform a find operation in this set. We have simplified this step by reordering the number of states, such that the regular states are numbered from 0 to $a-b-1$, and the final states are numbered from $a-b$ to a, where a is the total number of states and b is the total number of final states. Determining whether a pattern has been found is done by comparing if the *next state* is greater or equal than $a-b$.

A typical representation of the *State Transition Table* (STT) would be a matrix of $x \times y$ elements, where x is the number of states, and y is the size of alphabet $\sum$. The drawback of this representation is that a mapping between the characters of the alphabet and the items on the header of the STT is required (such as, $A \rightarrow 0, C \rightarrow 1, G \rightarrow 2\ and\ T \rightarrow 3$). We avoid this issue by representing the automaton as a sparse STT, where the characters of the alphabet are represented by their ASCII code. The size of the STT becomes $x \times z$, where z is the number of items in the ASCII table (that is 256 including the extended ASCII codes). Only the cells that belong to the ASCII codes that represent the characters on the alphabet contain the address to the *next state*, the other ones contain a transition to the start state (q_0).

Table I depicts the sparse STT representation of the automaton shown in Fig 4. Our representation of sparse STT is more expensive in terms of memory space, but it is a reasonable trade-off between memory space and access speed.

TABLE I
STT STRUCTURE THAT REPRESENTS THE AC AUTOMATON FROM FIG. 4

	← ASCII alphabet (256) →								
Q	...	*A*	...	*C*	...	*G*	...	*T*	...
	...	65	...	67	...	71	...	84	...
0		q_1		q_3		q_0		q_4	
1		q_1		q_2		q_0		q_4	
2		q_1		q_3		q_6		q_7	
3		q_1		q_3		q_0		q_5	
4	...	q_1	...	q_3	...	q_5	...	q_4	...
5		q_8		q_3		q_0		q_4	
6		q_1		q_3		q_0		q_4	
7		q_8		q_3		q_5		q_4	
8		q_1		q_2		q_0		q_4	

C. Algorithm Analysis

We focus on the most time-consuming parts of our algorithm (Alg. 1). The analysis assumes the worst case; for example, the *if-statement* at line 13 is assumed to be always true. The estimated time is expressed as follows:

$T = t_2 + t_3 + t_4 + p * (t_5 + t_6 + t_7) + p * r(t_8 + t_9) + p * r * v(t_10 + t_11 + t_12 + t_13 + t_14)$

where t_i indicates execution time of line i, p is the number of processing units, r is the chunk length, and v is the vector length. If a is $t_2 + t_3 + t_4$, b is $t_5 + t_6 + t_7$, c is $t_8 + t_9$, d is $t_10 + t_11 + t_12 + t_13 + t_14$, then we obtain the following,

$T = a + p * b + (p * r) * c + (p * r * v) * d$

Asymptotically we can express the time complexity of our algorithm as:

$\mathcal{O}(p(b + r(c + v * d)))$

The total parallelization overhead of our algorithm can be summarized as: $v(m-1) + p(m-1)$.

IV. EXPERIMENTAL EVALUATION

In this section we describe the experimentation environment used for the evaluation of our proposed algorithm and we discuss the obtained performance results.

A. Experimentation Environment

We have implemented our algorithm using C++11 programming language and OpenMP. The algorithm is compiled using the Intel Compiler icc 15.0.0, with enabled O2 optimization option. We have addressed the variability in the performance measurements by repeating the experiment 20 times for each problem size and number of threads. We used the Intel Vtune Amplifier 2015 for performance data collection.

To evaluate our algorithm the experiments were performed on an Intel Xeon Phi 7120P coprocessor. The Xeon Phi device contains 61 cores, each core supports four hardware threads. The coprocessor software includes the μOS version 2.6.38.8 and the Intel Manycore Platform Software Stack (MPSS) version 3.1.1. One of the 61 cores is used to run the coprocessor software, and the remaining 60 cores are used for DNA analysis in our experiments.

For the experimental evaluation we have selected the DNA sequences of mouse, cat, dog, chicken, human and turkey from the GenBank sequence database of the National Center for Biological Information [21]. Information about the genome references and the length of the DNA sequences are listed in Table II.

TABLE II
DNA DATA-SETS

	Gneome Reference	*Size (MB)*
Mouse	GRCm38.p2	2830
Cat	Felis_catus-6.2	2490
Dog	CanFam3.1	2440
Chicken	Gallus_gullus-4.0	1060
Human	GRCh38	3250
Turkey	Meleagris_gallopavo	193

In our experiments we use a set of patterns (see Table III) from the *regex-dna* [22] benchmark that match and extract specific 8-mers from a DNA sequence.

B. Results

We evaluated our algorithm on the Xeon Phi with different numbers of threads for each of the DNA sequences listed in Table II. Furthermore, we have varied the threads affinity, by allocating the threads under *compact, balanced,* and *scatter* mode. The *compact* mode completely fills a core with threads (that is, allocates threads to all available hardware threads of a core) before assigning threads to another core. The *balanced* mode evenly distributes the threads among cores. This mode

TABLE III
PATTERNS OF THE *regex-dna* BENCHMARK.

$agggtaaa$	$tttaccct$
$(c\|g\|t)gggtaaa$	$tttaccc(a\|c\|g)$
$a(a\|c\|t)ggtaaa$	$tttacc(a\|g\|t)t$
$ag(a\|c\|t)gtaaa$	$tttac(a\|g\|t)ct$
$agg(a\|c\|t)taaa$	$ttta(a\|g\|t)cct$
$aggg(a\|c\|g)aaa$	$ttt(c\|g\|t)ccct$
$agggt(c\|g\|t)aa$	$tt(a\|c\|g)accct$
$agggta(c\|g\|t)a$	$t(a\|c\|g)taccct$
$agggtaa(c\|g\|t)$	$(a\|c\|g)ttaccct$

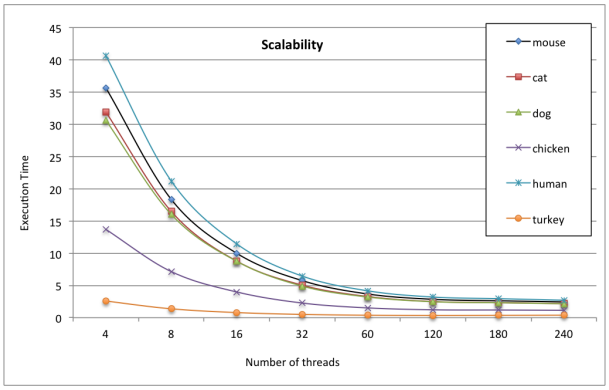


Fig. 5. The scalability of our algorithm on the Xeon Phi for various number of threads and problem sizes.

keeps thread IDs close to each other, which increases the probability that threads on the same core are using data that is close to each other. The *scatter* mode evenly distributes threads among cores in a round-robin fashion, which in contrast to the *balanced* mode assures that the thread IDs are not close to each other [23].

Table IV shows the execution time for the three different thread allocation affinity modes. The values in bold indicate the fastest execution time. We can see that for 240 threads, the *balanced* mode is the fastest one for all tested DNA sequences. For 180 threads the *scatter* mode performs the best in the case of dog's DNA sequence.

The performance data for scalability (Fig. 5) and the speedup (Fig. 6) is collected for the *balanced* thread affinity.

Fig. 5 shows the scalability of our algorithm when we increase the number of threads on the Intel Xeon Phi co-processor. Our algorithm scales well up to 120 threads for most of the tested DNA sequences. Increase of the number of threads to 180 or 240, results with a modest performance improvement due to the thread management overhead. The performance gain when using a larger number of threads is higher for larger DNA sequences. Thus the best scalability we observe for the human DNA sequence, which is the largest DNA sequence used in our experiments.

Fig. 6 presents the achieved speedup. The maximal speedup of $10\times$ is achieved for the human DNA sequence using 240 threads compared to a sequential version running on an Intel Xeon E5-2695v2 CPU.

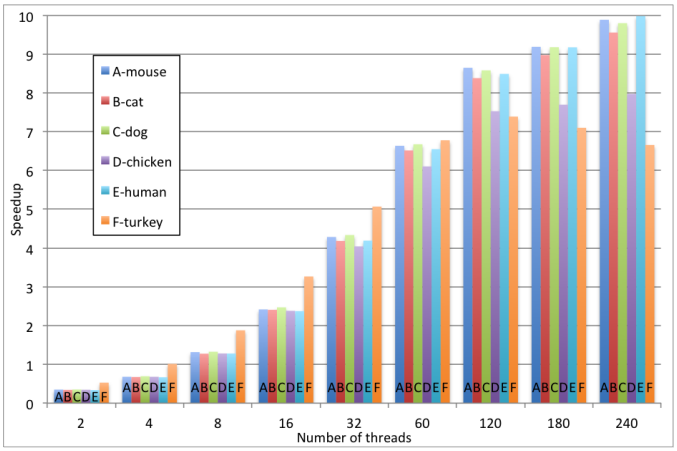


Fig. 6. Speedup of our algorithm with respect to a sequential version running on an Intel Xeon E5-2695v2 CPU.

TABLE IV
THE EXECUTION TIME ON THE COPROCESSOR FOR DIFFERENT NUMBER OF THREADS AND THREAD AFFINITY MODES.

Affinity	Compact			Balanced			Scatter		
Threads	240	180	120	240	180	120	240	180	120
Mouse	2.45	2.72	3.36	**2.44**	**2.63**	**2.85**	2.66	2.76	3.00
Cat	2.21	2.43	2.99	**2.19**	**2.34**	**2.51**	2.27	2.35	2.52
Dog	2.19	2.39	2.95	**2.16**	2.32	**2.48**	2.27	**2.31**	2.52
Chicken	1.18	1.23	1.43	**1.15**	**1.19**	1.22	1.25	1.21	**1.21**
Human	2.77	3.07	3.82	**2.71**	**2.95**	3.20	2.88	2.97	**3.19**
Turkey	0.41	0.38	0.39	**0.39**	**0.36**	**0.35**	0.43	0.38	0.36

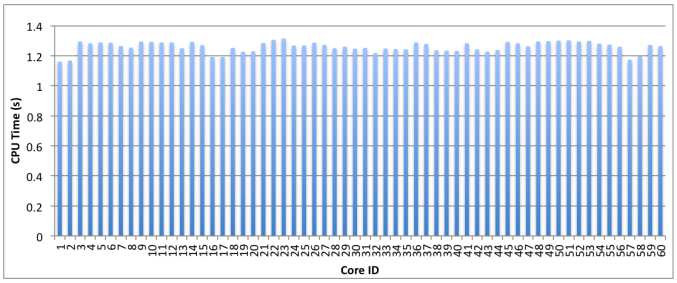


Fig. 7. CPU time for each core. The human DNA sequence is processed with the *balanced* thread affinity mode.

Figure 7 shows the CPU time for each core when the human DNA sequence is processed with the *balanced* thread affinity mode. Theoretically we may expect the CPU time of each core to be the same, because each core has the same amount of symbols to process. The CPU time of individual cores may depend not only on the chunk length but also on how many occurrences of the patterns are in the corresponding sequence chunk.

V. RELATED STATE-OF-THE-ART

In this section we discuss the state-of-the-art in pattern matching and DNA analysis using many-core architectures (such as, GPU and Intel Xeon Phi).

Villa et al. [24] implemented the Aho-Corasick string matching algorithm on a Cray XMT system. Tumeo and Villa implemented the algorithm presented in [24] for GPU clusters [10]. Their implementation is based on splitting the input into chunks, and then processing each chunk in a separate thread. In contrast to our approach, their algorithm for pattern matching relies on the features of the Cray XMT or GPU architecture, whereas our algorithm is tailored for DNA analysis on Intel Xeon Phi architecture.

An acceleration of exact string matching Knuth-Morris-Pratt algorithm on GPU is conducted by Bellekens et al. [9]. They achieve nearly a $29\times$ speedup compared to the sequential version of the KMP algorithm. Similarly, Kouzinopoulos and Margaritis [8] conducted an experiment on the Naive, KMP, Boyer-Moore-Horspool and Quick-Search string matching algorithms in the context of DNA sequencing using the CUDA toolkit. In contrast our work addresses large-scale DNA analysis on Intel Xeon Phi.

Lin et al. [7] evaluated their Parallel Failure-less AC algorithm on GPU and showed improvement of $14.74\times$. This algorithm allocates a new thread to each character of the input to identify any pattern starting from that character, which means that it creates n number of threads, where n is the input length. In their experiments the length of input string is up to 256MB. While this approach is tailored for pattern matching on GPU, we focus on DNA analysis on Xeon Phi.

Li et al. [25] implemented in CUDA the Wu-Manber algorithm, which is used for approximate matching of nucleotides in DNA sequences on GPU. In contrast our algorithm performs exact pattern matching on Intel Xeon Phi.

To the best of our knowledge, our approach for large-scale DNA analysis is the first one that exploits the thread level and SIMD parallelism available on the Intel Xeon Phi coprocessor. In our experiments we have evaluated our approach with real-world DNA sequences of several GB.

VI. Summary and Future Work

Fast DNA analysis is important in many applications, such as, preventing the evolution of different viruses during an early phase, early diagnosis of genetic predispositions to certain diseases, or DNA forensics.

In this paper we have presented an approach for accelerating DNA analysis using the Intel Xeon Phi coprocessor. The proposed parallel algorithm is based on finite automata and is used for counting and extracting the location of k-mers in a DNA sequence. Our approach exploits the thread-level and SIMD parallelism of the Intel Xeon Phi coprocessor, and therefore it is suitable for large-scale DNA sequences. Experiments with real-world data-sets of several GB demonstrate that the algorithm scales well with respect to various numbers of threads and problem sizes. The best scalability we observed for the human DNA sequence, which was the largest DNA sequence used in our experiments.

Future work will address the DNA analysis on the upcoming generation of the Intel Xeon Phi coprocessor known as the *Knights Landing*.

References

[1] S. Aluru, N. M. Amato, and D. A. Bader, "Editorial: Special section on high-performance computational biology," *IEEE Transactions on Parallel and Distributed Systems*, vol. 17, no. 8, pp. 737–739, 2006.

[2] D. R. Bentley, "Decoding the human genome sequence," *Human Molecular Genetics*, vol. 9, no. 16, pp. 2353–2358, 2000.

[3] D. A. Benson, M. Cavanaugh, K. Clark, I. Karsch-Mizrachi, D. J. Lipman, J. Ostell, and E. W. Sayers, "Genbank," *Nucleic acids research*, vol. 41, no. D1, pp. D36–D42, 2013.

[4] F. S. Collins, E. D. Green, A. E. Guttmacher, and M. S. Guyer, "A vision for the future of genomics research," *Nature*, vol. 422, no. 6934, pp. 835–847, 2003.

[5] Mellmann *et al.*, "Prospective genomic characterization of the german enterohemorrhagic escherichia coli o104: H4 outbreak by rapid next generation sequencing technology," *PloS one*, vol. 6, p. e22751, 2011.

[6] M. A. Luftig and S. Richey, "Dna and forensic science," *New Eng. L. Rev.*, vol. 35, p. 609, 2000.

[7] C.-H. Lin, C.-H. Liu, L.-S. Chien, and S.-C. Chang, "Accelerating pattern matching using a novel parallel algorithm on gpus," *Computers, IEEE Transactions on*, vol. 62, no. 10, pp. 1906–1916, Oct 2013.

[8] C. S. Kouzinopoulos and K. G. Margaritis, "String matching on a multicore gpu using cuda," in *Informatics, 2009. PCI'09. 13th Panhellenic Conference on*. IEEE, 2009, pp. 14–18.

[9] X. Bellekens, I. Andonovic, R. Atkinson, C. Renfrew, and T. Kirkham, "Investigation of gpu-based pattern matching," in *The 14th Annual Post Graduate Symposium on the Convergence of Telecommunications, Networking and Broadcasting (PGNet2013)*, 2013.

[10] A. Tumeo and O. Villa, "Accelerating dna analysis applications on gpu clusters," in *Application Specific Processors (SASP), 2010 IEEE 8th Symposium on*, June 2010, pp. 71–76.

[11] J. Dokulil, E. Bajrovic, S. Benkner, S. Pllana, M. Sandrieser, and B. Bachmayer, "High-level support for hybrid parallel execution of c++ applications targeting intel xeon phi coprocessors." in *ICCS*, ser. Procedia Computer Science, vol. 18. Elsevier, 2013, pp. 2508–2511.

[12] S. Pllana, S. Benkner, E. Mehofer, L. Natvig, and F. Xhafa, "Towards an intelligent environment for programming multi-core computing systems." in *Euro-Par Workshops*, ser. Lecture Notes in Computer Science, vol. 5415. Springer, 2008, pp. 141–151.

[13] C. W. Kessler, U. Dastgeer, S. Thibault, R. Namyst, A. Richards, U. Dolinsky, S. Benkner, J. L. Trff, and S. Pllana, "Programmability and performance portability aspects of heterogeneous multi-/manycore systems." in *DATE*. IEEE, 2012, pp. 1403–1408.

[14] J. Hopcroft and J.D. Ullman, *Introduction to Automata Theory, Languages and Computation*. Addison-Wesley, Cambridge, 1979.

[15] A. V. Aho and M. J. Corasick, "Efficient string matching: An aid to bibliographic search." *Commun. ACM*, vol. 18, pp. 333–340, 1975.

[16] G. Chrysos, "Intel® xeon phi coprocessor-the architecture," *Intel Whitepaper*, 2014.

[17] X. Tian, H. Saito, S. Preis, E. N. Garcia, S. Kozhukhov, M. Masten, A. G. Cherkasov, and N. Panchenko, "Practical simd vectorization techniques for intel xeon phi coprocessors." in *IPDPS Workshops*, 2013.

[18] D. Luchaup, R. Smith, C. Estan, and S. Jha, "Speculative parallel pattern matching." *IEEE Transactions on Information Forensics and Security*, vol. 6, no. 2, pp. 438–451, 2011.

[19] A. Chacn, J. C. Moure, A. Espinosa, and P. Hernndez, "n-step fm-index for faster pattern matching." in *ICCS*, ser. Procedia Computer Science, vol. 18. Elsevier, 2013, pp. 70–79.

[20] S. Memeti and S. Pllana, "Parem: A novel approach for parallel regular expression matching," in *Computational Science and Engineering (CSE), 2014 IEEE 17th International Conference on*, Dec 2014, pp. 690–697.

[21] "National center for biotechnology information u.s. national library of medicine," http://www.ncbi.nlm.nih.gov/genbank, accessed: Apr. 2015.

[22] "The computer language benchmarks game," http://benchmarksgame.alioth.debian.org/, accessed: Apr. 2015.

[23] M. Barth, K. Sweden, M. Byckling, C. Finland, N. Ilieva, N. Bulgaria, S. Saarinen, M. Schliephake, V. Weinberg, and L. Germany, "Best practice guide intel xeon phi v1." 2013.

[24] O. Villa, D. G. Chavarra-Miranda, and K. J. Maschhoff, "Input-independent, scalable and fast string matching on the cray xmt." in *IPDPS*. IEEE, 2009, pp. 1–12.

[25] H. Li, B. Ni, M. H. Wong, and K.-S. Leung, "A fast cuda implementation of agrep algorithm for approximate nucleotide sequence matching." in *SASP*. IEEE Computer Society, 2011, pp. 74–77.