High Performance Parallelization of Boyer-Moore Algorithm on Many-Core Accelerators

Yosang Jeong, Myungho Lee
Dept of Computer Science and Engineering
Myongji Unviersity
116 Myongji Ro, Cheo-In Gu
Yong In, Kyung Ki Do, Korea
myunghol@mju.ac.kr

Dukyun Nam, Jik-Soo Kim, Soonwook Hwang
Supercomputing R&D Center
Korea Institute of Science and Technology Information (KISTI)
245 Daehak-ro, Yuseong-Gu, Daejeon, Korea

***Abstract*—Boyer-Moore (BM) algorithm is a single pattern string matching algorithm. It is considered as the most efficient string matching algorithm and used in many applications. The algorithm first calculates two string shift rules based on the given pattern string in the preprocessing phase. These rules help skip parts of the target input string where there is no match to be found. Using the two shift rules, pattern matching operations are performed against the target input sting in the second phase. The second phase is a time consuming process and needs to be parallelized to achieve the high performance string matching. In this paper, we parallelize the BM algorithm on the latest many-core accelerators such as the Intel Xeon Phi and the Nvidia Tesla K20 GPU, along with the general-purpose multi-core processors. We partition the target input data amongst multiple threads for parallel execution. Data lying on the threads' boundaries need to be copied redundantly so that the pattern string lying on the boundary can be found. As the target length increases, the algorithm incurs increased matching operations. Also, as the pattern length increases, the number of possible matches decreases. This can potentially lead to the unbalanced workload distribution among threads. Furthermore, the redundant data copy significantly overloads the on-chip shared memories of the GPU for a large number of threads. We use the dynamic scheduling and the multithreading techniques to solve the load balancing problem. We also use the algorithmic cascading technique to reduce the burden on the shared memories of the GPU. Our parallel implementation leads to ~17-times speedup on the Xeon Phi and ~45-times speedup on the Nvidia Tesla K20 GPU compared with a serial implementation on the host Intel Xeon processor.**

Keywords—Boyer-Moore algorithm; many-core accelerator; parallelization; dynamic scheduling; multithreading; algorithmic cascading

I. INTRODUCTION

String matching is an important algorithm commonly used in computer and network security, bioinformatics, among many other applications. Boyer-Moore (BM) algorithm [1] is a classic single-pattern string matching algorithm developed by Robert S. Boyer and J. Strother Moore. This algorithm is known to be one of the most efficient string matching algorithms. Given a sequence of characters (pattern), the BM searches for possible matches in the input string (or target). The algorithm first calculates the two string shift rules based on the given pattern in the preprocessing phase. These rules help skip parts of the target input string where there is no match to be found. Then the pattern matching is performed using these rules against the target input data in the second phase. The second phase is a time consuming process and needs to be parallelized to achieve the high performance string matching.

Recently, many-core accelerator chips such as the Graphic Processing Units (GPUs) from Nvidia and AMD, Intel's Many Integrated Core (MIC) architectures, among others are becoming increasing popular. The influence of these chips is rapidly growing in the High Performance Computing (HPC) server market and in the Top 500 list, in particular. They have a large number of cores and multiple threads per core, levels of cache hierarchies, large amounts (> 5GB) of the on-board memory, and > 1 Tflops peak performance for the double precision arithmetic per chip. They are mostly utilized as co-processors and execute parallel program kernels commanded by the host CPU with respect to the input data provided from the host memory to the on-board device memory. Using the many-core accelerators, a number of innovative performance improvements have been reported for HPC applications and many more are still to come.

In this paper, we develop a high performance parallelization for the BM string matching algorithm on the many-core accelerator chips such as the Intel Xeon Phi and the Nvidia Tesla K20 GPU. We partition the target input data amongst multiple threads for parallel execution of the BM algorithm. Data lying on the threads' boundaries are copied redundantly so that the pattern string lying on the boundary can be found. As the length of the target increases, the execution time increases accordingly. Furthermore, as the pattern length increases, the number of match occurrences decreases in general. Also, the positions in the target input data where the matches occur are spread all over the target data randomly. This leads to irregular execution times among threads participating in the parallel execution, because the threads which find a smaller number of matches finish earlier than the threads with a larger number of matches. Our parallelization approach deals with the irregular workload distribution with a careful target data partitioning to generate appropriate number of data chunks. Also we use the dynamic scheduling and the multithreading to deal with the load balancing problem. The overheads associated with the

978-1-4799-5841-2/14 315.00 © 2014 IEEE
DOI 10.1109/ICCAC.2014.20

redundant data copy increases significantly as the pattern size increases and the number of threads increases. For example, the size of redundantly copied data is more than two times the size of the target when the pattern size is 50 and the number of threads reaches ~178 million on the GPU. This overloads the shared memories of the GPU. We use the algorithmic cascading technique [5] to reduce the burden on the shared memories of the GPU. Our parallel implementation leads to ~17-times speedup on the Xeon Phi and ~45-times speedup on the Nvidia Tesla K20 GPU compared with a serial implementation.

The rest of the paper is organized as follows: Section II explains the computational characteristics of the BM algorithm. Section III describes the architectures of the latest many-core accelerator chips including the Intel Xeon Phi and the Nvidia Tesla K20 GPU. Section IV introduces our parallelization approach for optimizing the workload balancing among parallel threads and the algorithmic cascading technique to reduce the burden on the shared memory. It also explains our parallelization methodology on different architectures. Section V shows the experimental results on the general-purpose Intel multi-core processor, the Xeon Phi, and the Nvidia Tesla K20. Section VI wraps up the paper with conclusions.

II. BOYER-MOORE ALGORITHM

Boyer-Moore (BM) algorithm [1] is a single-pattern string matching algorithm developed by Robert S. Boyer and J. Strother Moore in 1977. Given a sequence of characters (pattern), the BM searches for possible matches in the input string (or target). The BM algorithm consists of two phases: preprocessing phase and pattern matching phase. In the preprocessing step, two string shift rules are calculated in advance using the given pattern string. They are the Bad Character Shift rule (calculated when there is no match) and the Good Suffix Shift rule (calculated when there is a match). Using these shift rules, the pattern matching is attempted against the target in the second phase. The matching starts from the rightmost character of the pattern to the left. Whether there is a match or not, the position shift for the next match attempt is directed to the right.

A. Bad Character Shift

The pattern matching starts from the rightmost character of the pattern against the target. Whenever there is a mismatch found for a character in the middle of the pattern matching, we can skip a number of characters in the target string. The Bad Character Shift uses this matching rule. Assume that there is a pattern string "GCAGAGAG" with 8 characters as shown in Table I below. A pattern matching is attempted in Table II for a target string with 22 characters. We assume the following two cases:

- Case 1: a pattern matching is attempted at pattern[7] ("G") and target[7] ("C"). As they do not match, a "C" character is searched in the pattern string which is located at pattern[1]. Therefore, we skip 6 characters and locate the "C" character of the target at pattern[7].
- Case 2: the next pattern matching is attempted starting from target[13] and pattern[7]. As they do not match and we cannot find the missing character "T" in the pattern, 8 characters (length of the pattern) are skipped and the pattern is located from target[14] to target[21].

As the result of the above, a Bad Character Shift table is constructed as in Table III. Figure 1 and 2 above illustrate the two cases for Bad Character Shift.

TABLE I. PATTERN ARRAY

0	1	2	3	4	5	6	7
G	C	A	G	A	G	A	G

TABLE II. TARGET ARRAY AND MATCHING PROCESS

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21
	G	C	A	G	A	A	B	C	D	E	F	G	A	T	G	C	A	G	A	G	A	G
P1	G	C	A	G	A	G	A	G														
P2							G	C	A	G	A	G	A	G								
P3															G	C	A	G	A	G	A	G

TABLE III. BAD CHARACTER SHIFT

Characters with mismatch	G	C	A	Other characters
Shift length	2	6	1	8 (= length of pattern)

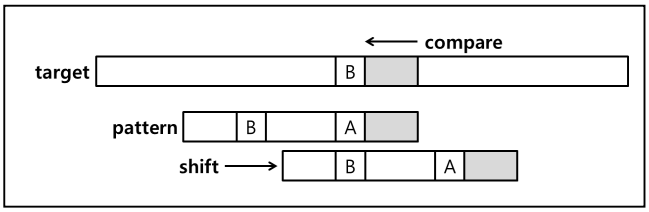

Fig. 1. Bad Character Shift Case 1

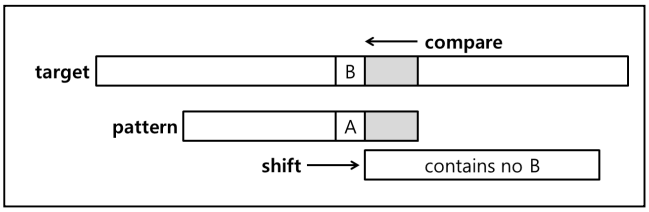

Fig. 2. Bad Character Shift Case 2

B. Good Suffix Shift

The Good Suffix rule searches for either a matching suffix or a symmetric prefix-suffix pair in the target. After finding such strings, we can skip a number of characters accordingly. Assume that there is a pattern string "GCAGAGAG" with 8 characters. A suffix or a prefix-suffix pair is searched for the pattern as shown in Table IV:

- For single character "G" located at pattern[0] and pattern[7], we find a common prefix-suffix pair (Good Suffix (GS) case 2.
- For characters with lengths 2 or 3, there is no common suffix or prefix-suffix pair.

- For characters with length 4, we find a common suffix "AG" (GS case 1).
- For characters with length 5, there is no common suffix or prefix-suffix pair.
- For characters with length 6, we find a common suffix "AGAG" (GS case 1).
- For characters with length 7, there is no common suffix or prefix-suffix pair.
- For characters with length 8, we find a common prefix-suffix pair "GCAGAGAG" (GS case 2).

According to the above, we construct a suffix array table (Table V).

TABLE IV. FINDING SUFFIX OR PREFIX-SUFFIX PAIR

prefix, suffix, Good Suffix (GS) case1 or 2	
G C A G A G A G G C A G A G A G	Length=1: GS case2 Common prefix-suffix=" G"
G C A G A G A G G C A G A G A G	Length=2: No common suffix or prefix-suffix pair
G C A G A G A G G C A G A G A G	Length=3: No common suffix or prefix-suffix pair
G C A G A G A G G C A G A G A G	Length=4: GS case1 Common suffix=" AG"
G C A G A G A G G C A G A G A G G C A G A G A G	Length=5: No common suffix or prefix-suffix pair
G C A G A G A G G C A G A G A G G C A G A G A G	Length=6: GS case1 Common suffix="AGAG"
G C A G A G A G G C A G A G A G G C A G A G A G	Length=7: No common suffix or prefix-suffix pair
G C A G A G A G G C A G A G A G G C A G A G A G	Length=8: GS case2 Common prefix-suffix= "GCAGAGAG"

TABLE V. SUFFIX ARRAY

Length-1	0	1	2	3	4	5	6	7
Length of suffix or prefix-suffix	1	0	0	2	0	4	0	8

TABLE VI. TARGET ARRAY AND MATCHING PROCESS

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21
	G	C	A	T	A	G	A	G	D	E	F	G	A	T	G	C	A	G	A	G	A	G
P1	G	C	A	G	A	G	A	G														
P2					G	C	A	G	A	G	A	G										

Table VI shows the matching process for a pattern "GCAGAGAG" against the target string. As explained above, the pattern matching is first attempted at the last character of the pattern (pattern[7]) and target[7]. The pattern[4:7] and target[4:7] match and the first mismatch occurs at target[3]. Referring to the Suffix_Array[3] in Table V, we find that there is a common suffix with length 2. Therefore we shift 4 characters to locate pattern[2:3] under target[6:7]. Figure 3 illustrates the GS case 1. Figure 4 illustrate the GS case 2 where there is a common prefix-suffix pair. In this case, we skip the characters so that the prefix is located under the target with which the previous suffix was matched.

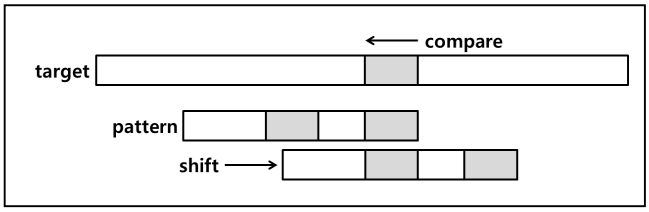

Fig. 3. Good Suffix Shift case 1

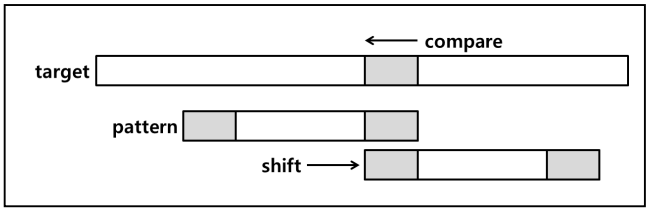

Fig. 4. Good Suffix Shift case2

C. *Time Complexity of the BM Algorithm*

The time complexity for the BM algorithm is O(3m) when there is no match found in the target and O(mn) when at least one match is found (m: length of pattern, n: length of target) [2]. According to the Galil Rule [4], the BM is superior to other string matching algorithms because of its linear time complexity. However, when there is (are) a match (matches) found, its complexity depends on the length of the target. Thus, when the length of the target increases, its execution time also increases accordingly. Furthermore, when we parallelize the BM algorithm, we would partition the target amongst a number of threads. Some threads would find matches with the piece of the target data assigned, whereas other threads may find no matches. Thus the workload distribution varies significantly.

III. ARCHICTECTURES OF MULTI-CORE MICROPROCESSORS AND MANY-CORE ACCELERATORS

In this paper, we implement the parallel BM algorithm on the many-core accelerator chips such as the Nvidia Tesla K20 based on the Kepler GK110 architecture and the Intel Xeon Phi besides the general-purpose multi-core microprocessor. The Nvidia Tesla K20 GPU integrates 2496 cores and delivers 1.17Tflops per chip performance for the double precision

arithmetic. Intel's Xeon Phi integrates 61 x86 cores and delivers > 1 Tflops (double precision). These chips are mostly utilized as co-processors and execute parallel program kernels commanded by the host CPU. In the following subsections, we describe these architectures

A. Nvidia Tesla K20 GPU

These days, the GPU is becoming increasingly popular for many applications with its innovative performance improvements and the developments of the user friendly programming environments. The latest GPU architecture is characterized by a large number of fine-grain programmable cores which are distributed in multiple hardware thread blocks. The large number of cores has replaced separate processing units for shader, vertex, and pixel in the earlier GPUs. Also, the clock rate of the latest GPUs has ramped up significantly compared with the earlier GPUs. These have drastically improved the floating-point performance of the GPUs, far exceeding that of the latest CPUs.

As shown in Figure 1, there are multiple thread blocks or Streaming Multiprocessors on the GPU chip. In each thread block, there are multiple thread processors (or cores). Multiple threads assigned to each thread block executes in parallel in the SIMD (Single Instruction Multiple Data) mode. Each thread executes the same instruction directed by the same Instruction Unit on its own data streaming from the device memory to the on-chip cache memories and registers. When a running group of threads (or WARP) encounters a cache miss, for example, the context is switched to a new thread group (or WARP) while the cache miss is serviced for the next few hundred cycles. Thus the GPU executes in a multithreaded fashion.

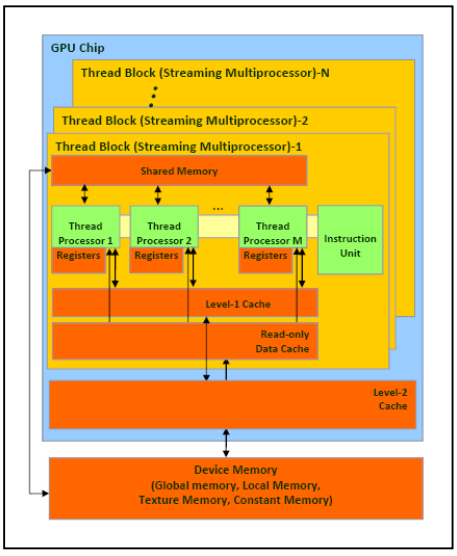

Figure 5. Architecture of a latest GPU (Nvidia Tesla K20)

The GPU is built around a sophisticated memory hierarchy (see Figure 5) [10]. There are registers and local memories belonging to each thread processor. The local memory is an area in the off-chip device memory. Shared memory, level-1 (L-1) cache, and read-only data cache are integrated in a thread block of a GPU. The shared memory is a fast (as fast as registers) programmer-managed memory. In the latest high-end Nvidia Kepler architecture (Tesla K20), there is also L-1 data cache and read-only data cache integrated in each thread block. Level-2 (L-2) cache is integrated on the GPU chip and used amongst all the thread blocks of the GPU. Global memory is an area in the off-chip device memory accessed from all the thread blocks through which the GPU can communicate with the host CPU. There are constant memory and texture memory regions in the device memory also. Data in these regions can be cached in the L-2 cache and the read-only data cache.

In order to efficiently utilize the advanced flexible hardware design of the latest GPU, programming environments such as CUDA [9, 10] from NVidia, OpenCL [12] from Khronos Group, OpenACC [13] from a subgroup of OpenMP Architecture Review Board (ARB) have been recently developed. Using these environments, users can have a more direct control over the large number of GPU cores and its sophisticated memory hierarchy. The flexible architecture and the programming environments have led to a number of innovative performance improvements in many application areas and many more are still to come.

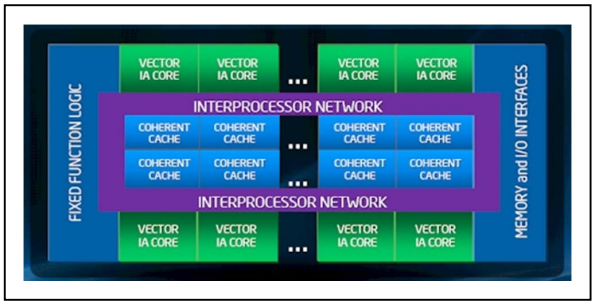

Figure 6. Architecture of Intel Xeon Phi

B. Intel Xeon Phi

The Intel Xeon Phi Coprocessor is the second generation product (codenamed Knights Corner) based on the Intel Many Integrated Core (MIC) architecture which combines many Intel CPU cores onto a single chip. In this paper, we use the Xeon Phi 5110P for our parallel implementation of the BM:

- This coprocessor has 60 in-order compute cores support 64-bit x86 instructions. These cores are connected by a high performance bidirectional ring interconnect (see Figure 6). It also has one service core, thus total 61 cores on the chip.
- It has 16 memory channels delivering up to 5GT/s. The total size of the system memory is 8GB.
- Each core is clocked at 1053 Mhz and offers the four-way Hyper-Threading (HT) 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 a combined 30 MB L2 cache. L2 cache is fully coherent using the hardware support.

The Intel Xeon Phi coprocessor is connected to a host Intel Xeon processor through a PCI-Express bus.

The Xeon Phi coprocessor offers the full capability to use the same tools, programming languages and models as an Intel Xeon Processor. The Xeon Phi can run applications written in industry-standard programming languages such as Fortran and C/C++. Parallel models can be used for the Xeon Phi such as OpenMP, MPI, Pthreads, Intel Clik Plus, Intel Threading Building Block, etc [6]. There are two models which can be used to run application codes on the Xeon Phi:

- Native mode: an application runs directly on the Xeon Phi. In order to use this mode, –mmic option is used in the compile command.
- Offload mode: the application runs on the host side and only the selected regions (compute-intensive regions) are offloaded to the coprocessor.

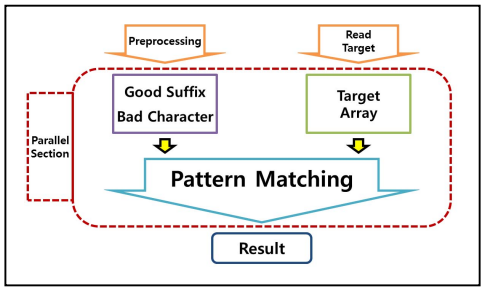

Figure 7. Parallel Execution of Boyer-Moore Algorithm

IV. High Performance Parallelization of Boyer-Moore Algorithm

As explained earlier in Section II, the BM algorithm consists of the preprocessing phase and the pattern matching phase. In this paper, we parallelize the second phase of the BM where the two shift rules generated in the first phase are used for the pattern matching against the target data loaded in the memory (see Figure 7). The time complexity for the BM algorithm grows with the increase in the length of the target when there is (are) a match (matches). When there is no match, the complexity depends on only the length of the pattern. Therefore, when we parallelize the BM algorithm we need to minimize the execution time for the large target sizes. Furthermore, when we partition the target data amongst a number of threads for the parallel execution, some threads would find matches whereas other threads may find no matches. Thus the workload distribution has large variances. In order to optimize the performance for the BM, we need to balance the workload amongst the threads participating in the parallel execution. Furthermore, when we partition the target input data amongst multiple threads for parallel execution, data lying on the threads' boundaries are copied redundantly so that the pattern string lying on the boundary can be found. The overheads associated with the redundant data copy increases significantly as the pattern size increases and the number of threads increases. Thus we need to reduce these overheads.

A. *Parallelization on Multi-Core Processor and Xeon Phi*

We first parallelize the BM algorithm on the Intel multi-core processor and on the Intel Xeon Phi which has a large number of x86 cores. We parallelize the algorithm by partitioning the input target data into a number of chunks. Then we let each thread match the pattern with respect to its target data chunk. When we generate the target data chunks, we copy (pattern_length-1)-bytes of data redundantly from its neighboring chunks. Although different threads have almost the same target length, they have different contents. The number of pattern occurrences in each thread may vary significantly. Thus the application of the two shift rules may result in different execution time for different threads. Figure 8 illustrates the execution scenario where each thread is assigned a target chunk with almost the same length, however, some threads have finished its execution and sit idle while other threads are still working on the pattern matching. Thus the overall parallel execution time is determined by the laggard thread. In order to minimize the inefficiency of the idle threads, we partition the target data into smaller chunks and have each thread work on a number of chunks. We use dynamic scheduling policy by letting a thread which finishes early for its assigned chunk ask for another chunk. Figure 9 shows the dynamic execution scenario.

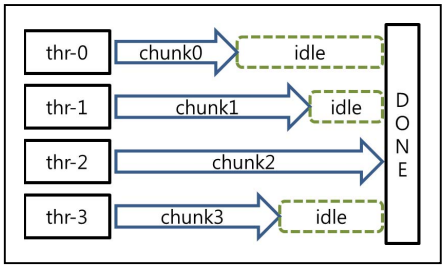

Figure 8. Static scheduling where each thread work on single data chunk

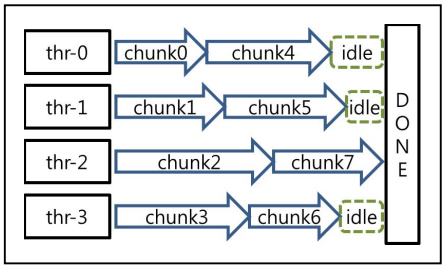

Figure 9. Dynamic scheduling where each thread works on multiple chunks

We also parallelize the BM algorithm on the Intel Xeon Phi. Basically we use the same parallelization methodology as we use for the multi-core microprocessor. However, we use the off-load mode of the Xeon Phi [6] so that the parallel kernel can be offloaded from the host multi-core processor to the Xeon Phi. The target data is first copied from the host memory to the on-board device memory of the Xeon Phi. Then the host CPU commands the execution of the parallel kernel on the Xeon Phi. Afterwards, the execution result is copied back to the host memory. There are up to 60 cores for the user process on the Xeon Phi (plus one service core). Each

core can have up to 4-way Hyper-Threading [8]. We also apply the dynamic scheduling on the Xeon Phi using up to 240 threads.

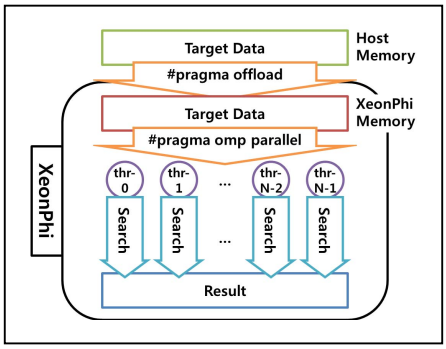

Figure 10. Parallel execution using off-load mode on the Intel Xeon Phi

B. *Parallelization on GPU*

The GPU executes parallel program kernels commanded by the host CPU on the data copied from the host memory and stored in the Global Memory (GM) region of the device memory. In order to parallelize the BM on the GPU, the input target data stored in the GM is partitioned and loaded into the Shared Memory (SM) of a hardware block (or Streaming Multiprocessor: SMX). Then threads in each SMX search the pattern in their assigned target data chunk in parallel. As in the multi-core and the Xeon Phi parallelization, we copy (pattern_length-1)-bytes of data from its neighboring chunks because a pattern may lie on the boundaries of consecutive target data chunks. Figure 11 illustrates the parallel pattern matching performed by threads in each SMX for the data copied from the GM to the SM. The irregular execution times on different threads can be efficiently masked off by a high degree of multithreading provided by the GPU.

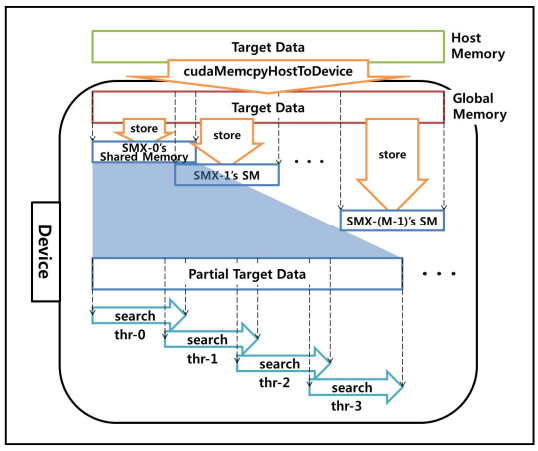

Figure 11. Parallel execution of BM on the GPU

When the target data size increases, the amount of data mapped to each SMX increases also. Furthermore the size of the redundantly copied data grows significantly with the size of the pattern and the number of threads. For example, the size of redundantly copied data is ~2.08-times the size of the target when the pattern length is 50 and the number of threads reaches ~178 million. In our parallelization, we divide the data mapped to each SMX by the size of the shared memory (SM) to generate a number of blocks. Thus each SMX handles multiple blocks. This helps hide the GM access latencies by the multithreaded execution of the SMX. However, when the number of blocks mapped to each SMX increases beyond the capacity of the SMX (and the shared memory in each SMX), there will be blocks which do not get mapped to the SMX. This will saturate the SMX and also will leave many blocks idle. Furthermore it results in increased mapping overheads. In order to solve the problem, we cascade multiple blocks algorithmically (or algorithmic cascading: ACC) [5]. This increases the size of the block and the amount of work for each thread, and decreases the number of blocks mapped to each SMX. Figure 12 shows the parallel execution using the ACC technique. Unlike in the execution of Figure 11 where each block is executed by each thread, every other block gets cascaded, for example, and executed by the same thread. Thus it increases the size of target data per thread and decrease the number blocks. This reduces the number of idle blocks which are not mapped, thus improve the benefit of multithreading. It also reduces the mapping overheads and leads to the improved performance.

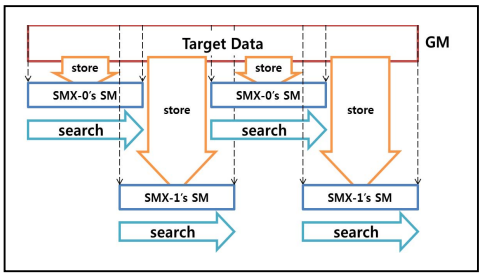

Figure 12. Algorithmic cascading (ACC) of two blocks from four blocks

V. EXPERIMENTAL RESULTS

In this section, we show the experimental results of parallelizing the BM algorithm on multi-core processors, the Intel Xeon Phi, and the GPU. Table VII summarizes the experimental environments. The multi-core processors are 2 Intel Xeon E5-2620 (2.0 Ghz, 6 cores each), the host memory is 32GB in size, the Xeon Phi has 60 compute cores with 8GB of on-board memory, the GPU is Nvidia Tesla K20 with 2496 single-precision cores and 5GB of device memory. We used the CentOS 6.3. For the parallel implementation on the multi-core and the Xeon Phi, we used OpenMP [3]. For the parallel implementation on the GPU, we used CUDA [9, 10].

TABLE VII. EXPERIMENTAL ENVIRONMENT

CPU	Intel® Xeon® E5-2620 2.00GHz x 2 (6 cores x 2)
Memory	32GB
Xeon Phi	Intel® Xeon Phi™ Coprocessor 5110P (8GB, 1.053 GHz, 60 core)
GPU	Nvidia Tesla K20 (5GB, 2496 CUDA core)
OS	CentOS 6.3

A. *Results on Multi-Core Processors and Xeon Phi*

Figure 13 shows the throughput performance (in Gbps) of the serial implementation and two parallel implementations with the static scheduling and the dynamic scheduling using 12 threads running on 12 cores (2 x 6 cores). In all the experiments, we used 1,000 chunks. Figure 14 shows the speedups of the two parallel implementations over the serial implementation. Overall the dynamic scheduling gives 11-times speedup over the serial implementation using 12 cores. (The static scheduling gives 10-times speedup.) The dynamic scheduling gives ~10% gain over the static scheduling. It helps balance the workloads among multiple threads.

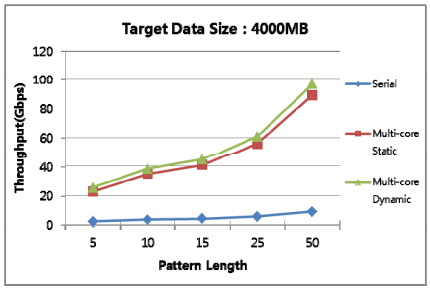

Figure 13. Throughput comparison of serial implementation, parallel implementations with static and dynamic scheduling

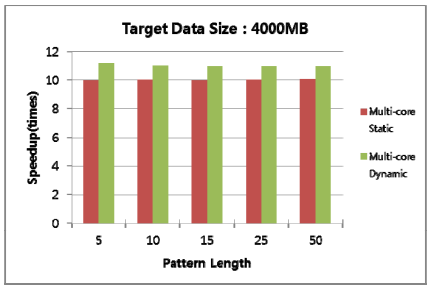

Figure 14. Speedup comparisons

The performance improvements get reduced as the length of the pattern increases. This is because we copy (pattern_length-1)-bytes of redundant data from its neighboring chunks. As the pattern length increases, the size of data chunk assigned to each thread increases which results in the enlarged problem size. For instance, when the number of target data chunks is 1,000, 3,996 characters are matched redundantly when the pattern size is 5. When the pattern size increases to 50, 48,951 characters are matched redundantly.

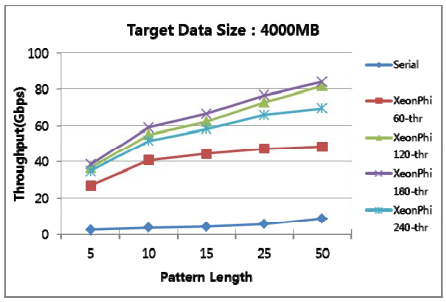

Figure 15. Throughput comparisons

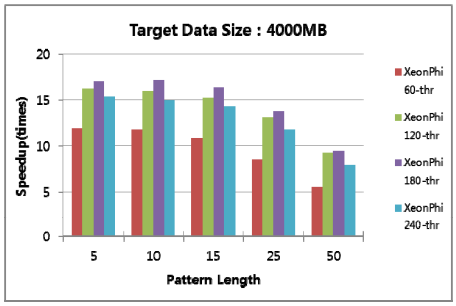

Figure 16. Speedup comparisons

Figure 15 shows the throughput performance (in Gbps) on the Xeon Phi using serial implementation and parallel implementations using 60, 120, 180, 240 threads. The best performance was obtained using 180 threads on 60 cores, thus 3-way Hyper-Threading was used in each core. The resulting throughput is over 80Gbps and the average speedup is 14.75 and maximum speedup is 17.14 over the serial implementation.

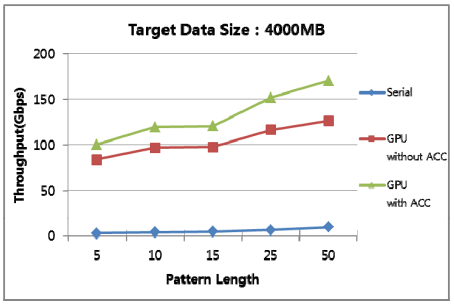

Figure 17. Throughput comparison of serial implementation, parallel implementations with and without algorithmic cascading

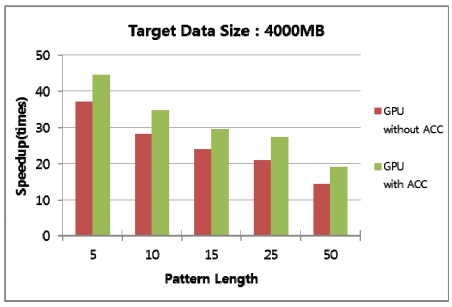

Figure 18. Speedup comparisons

B. *Results on GPU*

Figure 17 shows the throughput performance (in Gbps) on the Nvidia Tesla K20 GPU compared with the serial implementation. The GPU experiments were conducted with and without using the ACC technique. Using the ACC, we obtained an average speedup of 31 and the maximum speedup of 44.5. The ACC gives 26% better performance than without using the ACC. As explained earlier in subsection *A*, the redundant data copy affects the performance. The amount of redundant data copy using the ACC is significantly reduced to

degree_of_ACC * no_of_blocks * (no_of_threads/block -1) * (pattern_length -1). (When the degree of ACC = 64, pattern length = 50, the no of blocks = 21,730, and the no of threads per block = 128, total 8,654,450,560 characters are matched redundantly.)

C. Summary of Experimental Results

Figure 19 and 20 show performance comparisons among the serial implementation, parallel implementations on two 6-core Intel Xeon using 12 threads with the dynamic scheduling for the load balancing, on the Intel Xeon Phi using 60 cores and 3-way hyper-threading per core, and on the Tesla K20 GPU using the ACC technique. Figure 19 shows the throughput comparisons and Figure 20 shows the speedup comparisons among different parallel implementations over the serial implementation. Overall, the GPU implementation shows the best performance, followed by the Xeon Phi, then the multi-core implementation. As the length of the pattern increases, the speedup decreases due to the computations performed on the redundantly copied data on the data chunk boundaries. Also, as the pattern length increases and the number of threads increases, the number of threads without any pattern match occurrences increases. This leads to load imbalances among the threads and increase the overall parallel execution time.

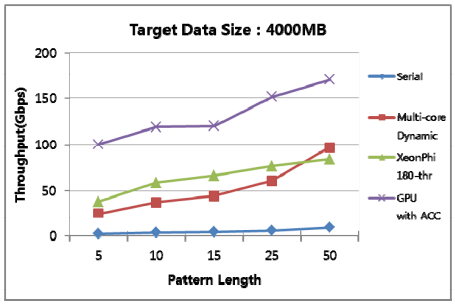

Figure 19. Throughput comparisons

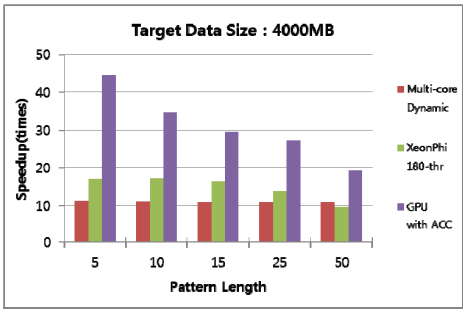

Figure 20. Speedup comparisons

VI. Conclusion

In this paper, we proposed a high performance parallelization of the BM algorithm which significantly improves the load balancing among parallel threads for irregular workload distributions and redundant data copy overheads on many-core accelerator chips including the Intel Xeon Phi and the Nvidia GPU. Our parallelization approach partitions the given set of target input data to generate a number of data chunks. In order to optimize the load balancing among threads, we carefully decide the chunk size and the resulting number of chunks. Then we employ the dynamic scheduling to smooth the execution time variances among different threads. On the Xeon Phi, the multithreading using the Hyper-Threading technology is also used to further the performance improvements. The multithreading on the GPU also improves the load balancing among threads by assigning a large number of threads per each core, thus the overall run time variance gets minimized. Furthermore, the GPU implementation uses the algorithmic cascading (ACC) to minimize the scheduling overheads. Experimental results show ~17-times speedup on the Xeon Phi and ~45-times speedup on the Nvidia Tesla K20 GPU compared with a serial implementation.

Acknowledgment

This Research has been performed as a subproject (P14008) of project "National Supercomputing Technology Development and Research" and supported by the KOREA INSTITUTE of SCIENCE and TECHNOLOGY INFORMATION (KISTI).

References

[1] Boyer, Robert S., and J. Strother Moore. "A fast string searching algorithm." Communications of the ACM 20.10 (1977): 762-772.

[2] Cole, Richard. "Tight bounds on the complexity of the Boyer-Moore string matching algorithm." SIAM Journal on Computing 23.5 (1994): 1075-1091.

[3] Dagum, Leonardo, and Ramesh Menon. "OpenMP: an industry standard API for shared memory programming." Computational Science & Engineering, IEEE 5.1 (1998): 46-55.

[4] Galil, Zvi. "On improving the worst case running time of the Boyer-Moore string matching algorithm." Communications of the ACM 22.9 (1979): 505-508.

[5] Harris, Mark. "Optimizing parallel reduction in CUDA." NVIDIA Developer Technology 2 (2007): 45.

[6] Jeffers, James, and James Reinders. Intel Xeon Phi Coprocessor High Performance Programming. Newnes, (2013).

[7] Kouzinopoulos, Charalampos S., and Konstantinos G. Margaritis. "String Matching on a multicore GPU using CUDA." Informatics, 2009. PCI'09. 13th Panhellenic Conference on. IEEE, 2009.

[8] Marr, Deborah T., et al. "Hyper-Threading Technology Architecture and Microarchitecture." Intel Technology Journal 6.1 (2002).

[9] NVIDIA, CUDA. "Programming guide." (2008).

[10] NVIDIA, CUDA. "Toolkit." (2009).

[11] Rao, Someswara, K. Butchi Raju, and Chinta S Viswanadha Raju. "Parallel String Matching with Multi Core Processors-A Comparative Study for Gene Sequences." Global Journal of Computer Science and Technology 13.1 (2013).

[12] Stone, John E., David Gohara, and Guochun Shi. "OpenCL: A parallel programming standard for heterogeneous computing systems." Computing in science & engineering 12.3 (2010): 66.

[13] "The OpenACC Application Programming Interface version 1.0," http://www.openacc.org/sites/default/files/OpenACC.1.0_0.pdf, (2011).

[14] Zhou, Junrui, et al. "Implementation of String Match Algorithm BMH on GPU Using CUDA." Energy Procedia 13 (2011): 1853-1861.