High Performance Implementation of Planted Motif Problem using Suffix trees

Naga Shailaja Dasari
Old Dominion University
ndasari@cs.odu.edu

Desh Ranjan
Old Dominion University
dranjan@cs.odu.edu

Zubair M
Old Dominion University
zubair@cs.odu.edu

ABSTRACT

In this paper we present a high performance implementation of suffix tree based solution to the planted motif problem on two different parallel architectures: NVIDIA GPU and Intel Multicore machines. An (l,d) planted motif problem(PMP) is defined as: Given a sequence of n DNA sequences, each of length L, find M, the set of sequences(or motifs) of length l which have at-least one d-neighbor in each of the n sequences. Here, a d-neighbor of a sequence is a sequence of same length that differs in at-most d positions. PMP is a well studied problem in computational biology. It is useful in developing methods for finding transcription factor binding sites, sequence classification and for building phylogenetic trees. The problem is computationally challenging to solve, for example a (19,7) PMP takes 9.9 hours on a sequential machine. Many approaches to solve planted motif problem can be found in literature. One approach is based on use of suffix tree data structure. Though suffix tree based methods are the most efficient ones for solving large planted motif problems on sequential machines, they are quite difficult to parallelize. We present suffix tree based parallel solutions for PMP on NVIDIA GPU and Intel Multicore architectures that are efficient and scalable. The solutions are based on a suffix tree algorithm previously presented but use extensive adaptation to individual architectures to ensure that the implementations work efficiently and scale well.

KEYWORDS: PMP, DNA, multicore, BitBased, parallel, mSPELLER, gSPELLER.

1. INTRODUCTION

The planted motif problem (PMP) is a fundamental search problem with applications in computational biology, especially in locating regulatory sites, sequence classification and building phylogenetic trees [1], [2], [3]. The *(l, d)* planted motif problem can be defined as: "Given a set of *n* DNA sequences, each of length *L*, find *M*, the set of sequences (or motifs) of length-*l* which have at-least one *d*-neighbour in each of the *n* sequences". A *d*-neighbour is a sequence of length *l* that differs from the motif in at most *d* positions. We refer to a sequence of length *l* as an *l*-mer in the rest of the paper.

Many approaches have been previously proposed to solve the planted motif problem. These approaches can be classified into two categories, heuristic and exact. Heuristic algorithms are very popular but they are not guaranteed to always find the correct answer. CONSENSUS, WINNOWER, Gibbs Sampling, Random Projections are some approaches that fall in this category. Exact algorithms on the other always produce the correct answer. These algorithms are also referred to as exhaustive enumeration algorithms. SPELLER [4], MITRA [5], PMSprune [6], Voting [7], RISOTTO [8] are some approaches that fall under this category. These algorithms can further be classified into pattern-driven and sample driven approaches. Pattern-driven approaches search all the possible $|\Sigma|^l$ *l*-mers to find the motifs. These algorithms have the time complexity of $\Omega(|\Sigma|^l)$. These algorithms are therefore only suitable for smaller values of *l* and perform prohibitively poorer for larger values of *l*. Sample-driven approaches on the other hand enumerate the *l*-mers in the input sequences to find the motifs. Sample-driven approaches are often limited by space requirements.

Sagot introduced a suffix tree based algorithm for solving planted motif problem [4] called SPELLER. This algorithm starts by building a generalized suffix tree for all the input sequences and uses this tree to "spell" all the motifs (called models in [4]). This algorithm is very efficient in terms of space. MITRA [5] uses a variation of suffix tree called Mismatch trees. MITRA works by splitting all the possible pattern space into disjoint subspaces starting with a given prefix thus breaking the problem into sub-problems. MITRA is more efficient than SPELLER in terms of both memory and space. SMILE [9], PSMILE [10], RISOTTO [8] are extensions to the

SPELLER algorithm, RISOTTO being the most recent approach. It uses maximum extensibility to efficiently spell the motifs.

Voting is a hashing based approach to solve PMP. Though it is efficient compared to brute force techniques, it was not able to solve problems with *d* greater than 5 as its running time increases exponentially with *d*.

PMS1, PMS2, PMSi, PMSP and PMSprune [6] are the most recent exact approaches. PMS1 is a simple approach. It enumerates all the *l*-mers in the input sequences and finds the motifs making use of radix sort. PMSi, PMSP, and PMSprune are based on similar idea. PMSprune is the most efficient algorithm among those. PMSprune was able to solve to solve the (19,7) instance which was not reported as solve previously in the literature.

All the exact approaches discussed above have been designed to work on serial computers and are not suitable for straightforward parallelization on current multicore or GPU architectures. A recently proposed parallel approach, BitBased, is based on working with bitarrays, and can be effectively parallelized on multicore and GPU architectures [11] [12]. However, the memory requirement is a bottleneck in this approach for solving *(l,d)* PMP problems as *l* and *d* increase. There have not been many attempts to propose a parallel approach based on suffix tree. To the best of our knowledge, PSMILE [10] is the only parallel suffix tree based approach that has been proposed. It can be seen from PSMILE that it is not straightforward to parallelize a suffix tree based approach as it is difficult to distribute the work equally among all the processors. However, the approach has not been tested on current multicore machines.

Tree based algorithms are notoriously hard to parallelize and even more so on GPU architecture that requires execution of large number of concurrent threads to achieve efficiency. Additionally when the tree structure is not balanced, the load distribution across different cores becomes an important issue. Another issue we need to be aware on multicore architectures is that caches are shared by different cores and a cache line that is updated by different cores generates a lot of memory traffic. Therefore it is desirable to have a parallel algorithm that works, where different cores update different portions of the storage area.

The rest of the paper is organized as follows. In the next section, we present the generalized suffix tree and core suffix tree algorithm, called the SPELLER algorithm [4], to solve the PMP. In Section 3. we explain the difficulties in parallelizing this algorithm for (Intel) multicore architecture and present techniques to overcome these difficulties. In Section 4. we present the GPU architecture and discuss the issues in parallelizing suffix tree method for this architecture. We then present adaptations to the core suffix tree method to obtain an implementation that is efficient and scalable for GPU architecture. Section 5. presents experimental results. We refer to our implementation of SPELLER on multicore and GPU as mSPELLER and gSPELLER respectively.

2. THE CORE SUFFIX TREE ALGORITHM

The basic suffix tree based method to solve PMP, called the SPELLER algorithm, was first introduced by Sagot [4]. It works by constructing a generalized suffix tree using the input sequences and then finding the motifs (or *spelling the models*) using this generalized suffix tree. Following that many modifications and extensions were proposed to improve the performance [8] [9] [10]. In this paper we adapt the original SPELLER algorithm for efficient implementation on multicore and GPU. This requires addressing several issues related to memory bottleneck, unbalanced load, conditional computation structure inhibiting concurrent execution of threads, etc. These are discussed and addressed in Section 3. and Section 4.

2.1. Suffix tree

Suffix tree is a data structure that represents all the suffixes of a sequence. Each suffix of the string corresponds to exactly one path from the root of the tree to a leaf. Many algorithms exist to construct suffix tree in linear time. We choose Ukkonen's algorithm. Suffix tree is compact version of suffix trie. Figure 1. shows the difference between a suffix trie and a suffix tree. Though nodes 2, 5, 6, 8 cannot be seen in the suffix tree, they are implicitly present. These nodes are called implicit nodes and the remaining nodes are explicit. A node can be uniquely referenced by *{edgeNum, length}* pair. For example the explicit node 3 can be referenced by {2, 1}, implicit nodes 2, 5, 6, 8 can be referenced as {1, 1}, {1, 2}, {4, 1} and {4, 3}. To represent an edge sequence, i.e. the sequence corresponding to an edge, we use *{fromIndex, toIndex}* pair instead of using the whole sequence. For example in Figure 1. the edge sequence of edge 4 is {1, 3}. For solving planted motif problem we use a generalized suffix tree which is a single suffix tree for a set of sequences. In a generalized suffix tree each suffix in each of the sequences corresponds to exactly one path from the root to a leaf node. If more than one sequences have the same suffix, then the path from root leads to the same leaf node. To avoid that a special symbol that is not in Σ and that is unique to each input sequence is appended to each input sequence. In case of generalized suffix tree we also need to add the sequence

number to represent an edge sequence, i.e. an edge sequence is now represented by the tuple *{fromIndex, toIndex, seqNum}*.

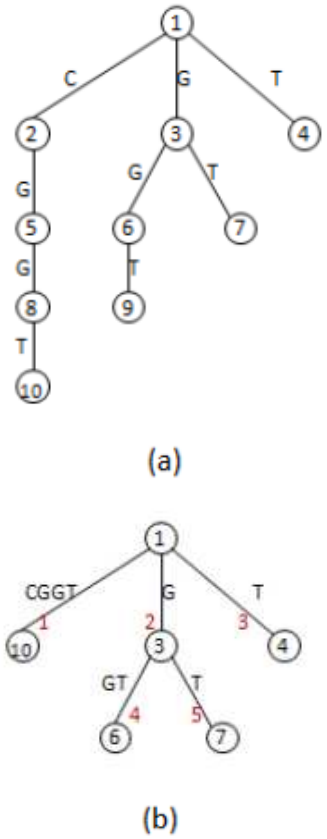

Figure 1. (a) Suffix trie (b) Suffix tree for the Sequence CGGT

2.2. Finding the Motifs

Once the generalized tree, *GT* is constructed using the given *n* input sequences, the SPELLER algorithm proceeds by finding the motifs recursively until the valid motifs are found or the required length is exceeded. Since we use a single suffix tree for all the sequences, we need to additionally store some sequence information in the tree. To do this each node in *GT* is assigned an array of size *n* denoted by *Colors*. *Colors[i]* for a node *x* is 1 if *x* lies on at least one path from root to a leaf that corresponds to a suffix of sequence *i*. It is 0 otherwise. We can use a bit vector to implement *Colors* array. Note that in the SPELLER algorithm color set size, CSS, information is also stored at each node but we don't use it in this paper. Let *p(x)* represent the path from the root of *GT* to the node *x*. (x, x_{err}) is called node occurrence of a sequence *m* if $dist(p(x), m) = x_{err}$ where *dist(y,z)* denotes the Hamming distance between sequences *y* and *z* of equal length. For an error value *d*, the occurrence list of a sequence *m*, Occ_m can be defined as a set of all node occurrences (x, x_{err}) such that x_{err} does not exceed *d* i.e $Occ_m = \{(x, x_{err}) \mid x_{err} \leq d\}$. For $\alpha \in \Sigma$, we can generate the occurrence list of $m\alpha$, $Occ_{m\alpha}$, from the occurrence list of *m*, Occ_m using the following lemma.

Lemma 1. [4] (x, err) is a node occurrence of $m\alpha$ if and only if one of the following satisfy:

a. $(parent(x), err)$ is a node occurrence of *m* and the label on the edge from *parent(x)* to *x* is α..
b. $(parent(x), err-1)$ is a node occurrence of *m* and the label from *parent(x)* to *x* is $\beta \neq \alpha$.

The key idea of the SPELLER algorithm is presented in Algorithm 1. The detailed algorithm can be found in [4]. The original SPELLER algorithm also uses other data structures but we do not use them as we found that they did not improve performance. *SpellModels* is called initially with parameters $k=0$, $m = \varepsilon$, $Occ_m = (root,0)$. It recursively calls the *SpellModels* incrementing the length and appending a residue.

3. ADAPTING SPELLER ON MULTICORE

Tree based algorithms are not straight forward to parallelize, especially if the tree is unbalanced. It is especially challenging to balance the load among multiple processors. SPELLER is a tree based algorithm and suffix tree by nature is very unbalanced. A previous attempt to parallelize SPELLER can be found in [10]. In [10] the count of the residues is used as the basis for distributing the load among multiple nodes.

In this paper we present a simpler and more balanced approach for parallelizing SPELLER. Note that we do not parallelize the construction of suffix tree in this paper. We only parallelize the spelling part of the approach. The main idea behind our approach is to start spelling from a length $l' > 0$ as opposed starting from length 0 in the original SPELLER algorithm. We first generate a node list containing all the nodes, both explicit and implicit, at level *i* and then use the node list to generate the occurrence list for a sequence of length *l'*. For a node *x*, let *p(x)* denote the sequence that leads from the root to node *x*. Let *NodeList(i) = {(x, p(x)) | x is a node at level i }*. We have seen that occurrence list of a sequence represents all the nodes that can be reached using the sequence or a *d*-neighbor of a sequence. Node list on the other hand represents all the nodes at a given level. So to obtain occurrence list for a sequence of length *i* from a node list of level *i* we need to filter out the nodes from the node list that do not correspond either to the sequence itself or *d*-

neighbors of the sequence. Algorithm 3. gives the procedure to obtain occurrence list from a node list.

Algorithm 1. Finding the Motifs

1: procedure *SpellModels(k, m, Occ_m)*
2: if $k = l$ then
3: output *m*
4: else
5: for each α in Σ do
6: generate $Occ_{m\alpha}$ using Occ_m
7: Let $Colors_{m\alpha}$ be the sum of *Colors* of the node occurrences of *mα*
8: if all the bits are set in $Colors_{m\alpha}$ then
9: *SpellModels(k + 1,mα,$Occ_{m\alpha}$)*
10: end if
11: end for
12: end if

As we have seen in Section 2., the function *SpellModels* is called with arguments *(0, λ, Occ_λ)* where 0 represents the length of the model, *λ* is an empty sequence representing the model and Occ_λ is the occurrence list of *λ* which is *(root, 0)*. In our approach we replace a single call to the function *SpellModels* with a loop as shown in Algorithm 2. The loop can then be easily parallelized by distributing the sequences of length *l'* among all the processing nodes. Note that the sequences can be assigned either statically or dynamically among the processors. If they are distributed statically, i.e equally among all the processors, the load is more unbalanced as some processors might be assigned more sequences that needs to be spelled to a longer length while some processors might have very few of such sequences keeping them idle for a longer time. So, to avoid that, we use dynamic distribution of sequences. In this case the processors are only assigned a small number of sequences initially. When a processor is done with its sequences it fetches the next available sequence to work on.

4. ADAPTING SPELLER ON GPU

GPU is a massively parallel, multi-threaded, manycore processor. Each GPU device is an array of streaming multiprocessor which in turn consists of a number of scalar processor cores. GPU is capable of running thousands of threads concurrently. It is able to do so by employing SIMT(single-instruction multiple-threads) architecture. The threads are created, scheduled and executed in groups called warps. All the threads in a warp share a single instruction unit. The threads in a GPU are extremely light weight and they can be created and executed with zero scheduling overhead.

Algorithm 2. Finding Motifs in Parallel

1: for each m_i of length *l'* do
2: *SpellModels(l', m_i, Occ_{mi})*
3: end for

CUDA is a parallel programming model that enables programmers to develop scalable applications to be executed on GPU. It exposes a set of extensions to C and C++. A CUDA program is organized into sequential host code which is executed on CPU and calls to functions called kernels which are executed on GPU. A kernel contains the device code that is executed by the GPU threads in parallel. CUDA threads can be grouped into thread blocks. Using CUDA one can define the number of blocks and the number of threads per block that can execute a kernel.

4.1. Memory Organization

The device RAM is virtually and physically divided into different types of memory: global, local, constant and texture memory. Apart from device RAM the threads can also access on-chip shared memory and registers as shown in Figure 2.. Global memory and texture memory have highest latency compared to the other types of memory. A thread has exclusive access to its local memory. All the threads in a block can access on-chip shared memory. All the threads across all thread blocks have access to global, texture and constant memory. Constant and texture memories are read only while global is both read and write.

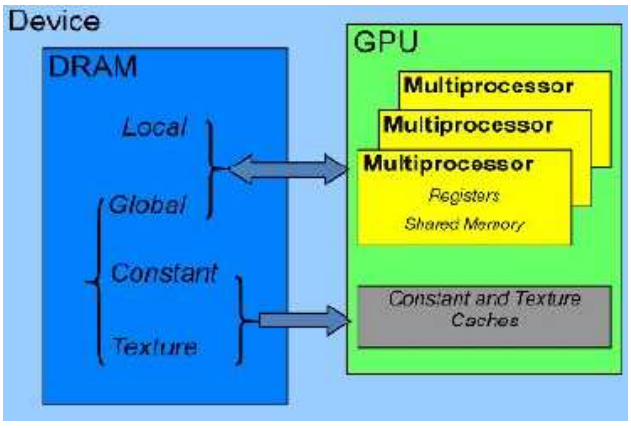

Figure 2. GPU Memory

4.2. Performance Considerations

A CUDA program should be properly designed taking advantage of the resources for better performance. Since GPU uses SIMT architecture in which all the threads in a warp use a single instruction unit, the best results can be achieved when all the threads in a warp execute without diverging. When threads diverge they are executed serially, thus decreasing performance.

Global memory has very high latency. But by coalescing the global memory accesses, high throughput can be achieved. For example if the threads in a warp access contiguous address, then only two transactions are issued. But if the threads access separate addresses then 32 transactions are issued.

Shared memory is divided into equally sized blocks called banks. If two threads in a half warp access the same bank, this would result in bank conflict and the accesses are serialized thus reducing the effective bandwidth. In order to avoid this, the programmer should try to make sure that the threads in a half warp access different banks.

The memory latencies can be hidden by executing other warps when a warp is paused. So to keep the hardware busy there should be enough active warps. Occupancy is the ratio of number of active warps per multi-processor to the maximum possible number of active warps. If the occupancy is too low, then the memory latency cannot be hidden resulting in performance degradation. So the programmer should try to increase the occupancy to effectively use the hardware. GPUs have proved efficient for many applications. It is very challenging to efficiently implement SPELLER on GPU. One of the reasons being the memory limitations of GPU. GPU offers different kinds of memory with varying memory latencies, some with caches and some on chip memories.

Algorithm 3. Generating Occurrence list from Node list

Input: *NodeList(l'), m*
Output: Occ_m
1: for each $(node_{l'}, seq_{l'})$ in *NodeList(l')* do
2: $error = dist(m, seq_{l'})$
3: if $error \leq d$ then
4: add $(node_{l'}, error)$ to Occ_m
5: end if
6: end for

In gSPELLER, we distribute the work among all the blocks in the same way as we do for multicore. We distribute all the sequences of length l', where l' is the length at which we start spelling, among all the blocks. Though the suffix tree itself requires less memory, the runtime memory requirements of SPELLER is high. SPELLER recursively calls *SpellModels* function generating an occurrence list at each level of recursion. Let c_i be the total number of nodes, both implicit and explicit, of level i. Let l' be the length at which we start spelling. The runtime memory requirement is therefore $O(c_i \ . \ (l-l'))$. For example, let's say the tree consists of 10000 nodes on average for each level and $l'=5$. For a *(15, 4)* problem, we would require 1.2 MB of runtime memory, assuming 12B for each occurrence list node. But this is for a single processing node. The more the number of processing units, the more the run time memory. We have seen that using CUDA we can declare very large number of blocks. For example in Tesla, we can declare 65535 blocks in a single row of a grid. But with only 4GB of global memory, we are limited to a maximum of 3333 blocks. To overcome this limitation we once again use dynamic allocation of sequences, as we do in the case of multicore, instead of statically assigning the sequences equally among all the blocks. Also note that though one can declare large number of blocks using CUDA, the maximum number of blocks that can be active at a time is very less, 64 for Tesla. So we only declare 64 blocks and dynamically distribute the sequences among the blocks. We use a variable called *nextSequence* to assign the sequences to the blocks. Whenever a block is done with the sequence assigned to it, it gets the next sequence using the *nextSequence* variable and updates the variable. Note that one must use atomic operation to achieve this.

The CUDA programming language that we use to implement applications on GPU does not support recursion. But the *SpellModels* function is a recursive function. So to implement *SpellModels* on GPU, we use two stacks. One stack contains the occurrence arrays and the other stack contains the information about the sequence to which the occurrence array belongs to.

In the original SPELLER algorithm, we read the occurrence list of a sequence \$m\$ and generate the occurrence list for $m\alpha$ where $\alpha \in \Sigma$ and continue spelling using $m\alpha$. When $m\alpha$ is done we again read the occurrence list of m and generate the occurrence list for $m\beta$ where $\beta \in \Sigma \setminus \{\alpha\}$. Observe that we read the occurrence list of \$m\$ four times once for each residue. We use global memory to store the occurrence list and the reads and writes to global memory are very expensive due to the high latency rate of global memory. So to avoid multiple

reads of a single occurrence list, we read the occurrence list of *m* and generate the occurrence lists for all the residues at once so that we don't have to read the occurrence list of *m* again. To do this we need four times the memory which is still achievable as we only use 64 blocks.

4.3. Tree Representation

We have seen in Section 2. that to get an edge sequence we need *fromIndex, toIndex* and *seqNum* values. In order to get a residue on the edge sequence one must first obtain the *fromIndex, toIndex, seqNum* and use these values to get the index of the residue and then read the residue from the sequence corresponding to *seqNum*. This adds up to four reads. To decrease the number of reads we include the sequence itself instead of the index information. We replace *fromIndex, toIndex* and *seqNum* with *bitSeq* and *length* where *bitSeq* is the bit sequence corresponding to the sequence and *length* is the length of the sequence. Bit sequence can be obtained by replacing A by 00, C by 01, G by 10 and T by 11. For example the bit sequence corresponding to edge sequence TAACG is 1100000110 and *length* is 5. This would require only two reads one for *bitSeq* and one for *length*. But this can be done only if the sequence length is less than or equal to 16(assuming *bitSeq* is an integer) because each residue needs two bits and there are 32 bits in an integer. If an edge has sequence length greater than 16 we split the edge into multiple edges each of length less than or equal to 16. We split the edges only if the edges are at a level less than or equal to *l* because otherwise we don't need them. For example if we are solving (15,5) problem, we only split the edges that are at a level less than or equal to 15 as we use only those edges for spelling.

The suffix tree is represented using two arrays node array and edge array. Each array element in the node array corresponds to a tree node and similarly each array element in the edge array corresponds to a tree edge. We use breadth first traversal to convert the suffix tree from the tree structure into array structure. Both the node array and edge array are bound to texture memories. Note that gSPELLER works only if the suffix tree fits in the memory as in the case of the sequences that we have tested.

4.4. Filtering

Instead of constructing the tree with all the n sequences, we only construct the tree for a smaller number of sequences, $n' \leq n$, and find the motifs that are present in all these n' sequences. These motifs are called candidate motifs. The candidate motifs are then filtered out by checking if they are present in the remaining *(n-n')* sequences. This approach had been previously used in [11], [12]. Unlike in [11], the main purpose of using filtering is not to reduce memory requirement but to improve performance. With decrease in number of sequences used for construction of suffix tree, the number of tree nodes decreases and so the size of occurrence list also decreases. Hence the time spent in reading and writing from global memory also reduces improving the overall performance. Note that the value of n' should not be too low in which case the time spent in filtering candidate motifs exceeds the time taken for obtaining the candidate motifs. It is straight forward to parallelize the filtering step. The candidate motifs are distributed among all the processors in case of multicore and threads in case of GPU.

Table 1. Time Taken by mSPELLER on multicore, gSPELLER on GPU and their comparison with other approaches

Algorithm	(13,4)	(15,5)	(17,6)	(19,7)	(21,8)
mSPELLER-16	2s	16.5s	2.5m	23.6m	3.7h
mSPELLER-8	2.7s	22.9s	3.6m	33.8m	-
mSPELLER-4	4.9s	43.5s	6.6m	1.1h	-
mSPELLER-1	18s	2.8m	27.3m	4.3h	-
gSPELLER-1	2.6s	26.34s	4.53m	46.9m	7.8h
gSPELLER-2	1.4s	13.5s	2.3m	24.1m	3.95h
gSPELLER-3	1s	9s	1.6m	16.7m	2.8h
gSPELLER-4	0.8s	7.2s	1.2m	13m	2.2h
BitBased-16	2s	11s	2.4m	30.6m	6.9h
PMSprune	53s	9m	69m	9.2h	-

5. EXPERIMENTAL RESULTS

We have implemented mSPELLER on a 4 quadcore 2.67 GHz Intel Xeon X5550 machine with a total of 16 cores and gSPELLER on Nvidia TESLA C1060 with 240 cores and Nvidia TESLA S1070 with 960 cores. We have tested our code with 20 input sequences of length 600 each. We tested it on random sequences with motifs planted at random positions in the 20 sequences. Table 1. presents the results of mSPELLER, gSPELLER on different number of cores/devices and their comparison with other approaches. mSPELLER-*x* shows the results of mSPELLER on *x* number of cores and gSPELLER-*x* shows the results of gSPELLER on *x* GPU devices. We compare the results with the results of BitBased approach on the same machine. It can be seen from Table 1. that the mSPELLER-16 performs better than BitBased on larger problems. The reason being that BitBased falls into memory issues for larger problems and uses the iterative approach which reduces its performance, whereas mSPELLER has no such memory issues. Also it can be

seen that gSPELLER-1 which has 240 cores does not perform well compared to mSPELLER-16. This is because of high thread divergence of gSPELLER. One of

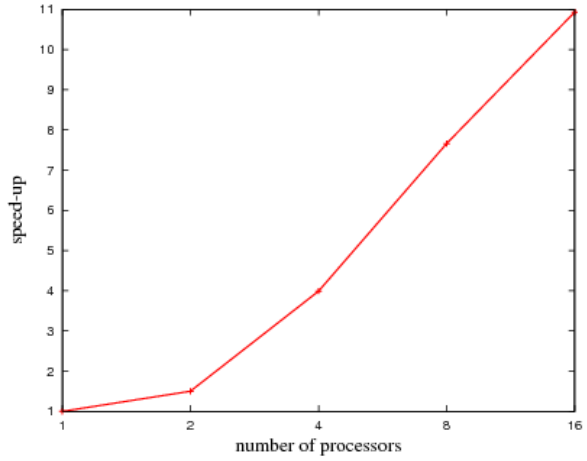

Figure 3. Plot Showing Scalability Results of mSPELLER for (17, 6) Problem

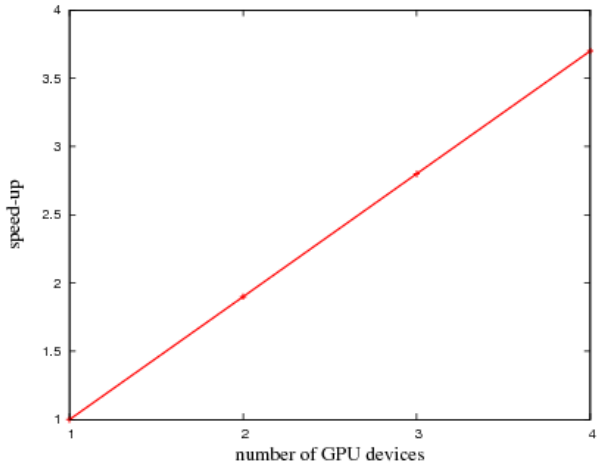

Figure 4. Plot Showing Scalability Results of gSPELLER for (17, 6) Problem

the main criteria for an algorithm to perform well on GPU is minimal thread divergence. But the gSPELLER algorithm is comprised of many conditional statements leading to thread divergence. Also the threads need to be synchronized at many places which adversely effect the performance. For example, as we have seen in Section 4. gSPELLER uses stack to store the occurrence list. The top of the stack should be carefully modified for getting correct results. The stack top should not be modified by a thread while some other thread is using the stack top. To avoid that, the threads must be synchronized before and after each time the stack top is modified. This significantly reduces the performance of gSPELLER. It can be seen from the Figures 3. and 4. that mSPELLER and gSPELLER scale well with increase in number of cores and devices respectively for (17,6) instance. All other instances also have a similar scale-up.

REFERENCES

[1] P. A. Pevzner and S.-H. Sze , "Combinatorial approaches to finding subtle signals in DNA sequences," in *ISMB*, pp. 269–278, 2000.

[2] M. K. Das and H.-K. Dai, "A survey of DNA motif finding algorithms," *BMC Bioinformatics*, Vol. 8, No. S-7, 2007.

[3] T. Ji, K. Gopavarapu, D. Ranjan, B. Vasudevan, C. Sengupta-Gopalan, and M. O'Connell, "Tools for cis-element recognition and phylogenetic tree construction based on conserved patterns," in *Computers and Their Applications*, pp. 1–6, 2007.

[4] M.-F. Sagot, "Spelling approximate repeated or common motifs using a suffix tree," in *LATIN*, pp. 374–390, 1998.

[5] E. Eskin and P. A. Pevzner, "Finding composite regulatory patterns in DNA sequences," in *ISMB*, pp. 354–363, 2002.

[6] J. Davila, S. Balla, and S. Rajasekaran, "Fast and practical algorithms for planted (l, d) motif search," IEEE/ACM Transactions on Computational Biology and Bioinformatics, Vol. 4, pp. 544–552, 2007.

[7] F. Y. L. Chin and H. C. M. Leung, "Voting algorithms for discovering long motifs," in *APBC*, pp. 261–271, 2005.

[8] N. Pisanti, A. M. Carvalho, L. Marsan, and M.-F. Sagot, "Risotto: Fast extraction of motifs with mismatches," in *LATIN*, pp. 757–768, 2006.

[9] L. Marsan and M.-F. Sagot, "Extracting structured motifs using a suffix tree - algorithms and application to promoter consensus identification," in *RECOMB*, pp. 210–219, 2000.

[10] A. M. Carvalho, A. L. Oliveira, A. T. Freitas, and M.-F. Sagot, "A parallel algorithm for the extraction of structured motifs," in Proceedings of the 2004 ACM symposium on Applied computing, pp. 147–153, 2004

[11] N. S. Dasari, R. Desh, and M. Zubair, "An efficient multicore implementation of planted motif problem," in Proceedings of the International Conference on High Performance Computing and Simulation, pp. 9–15, 2010.

[12] N. S. Dasari, R. Desh, and M. Zubair, "Solving planted motif problem on GPU," in International Workshop on GPUs and Scientific Applications, 2010.