Search for MUMs in enhanced suffix tree traversal with SIMD instructions.

Julio César García Vizcaíno
Computer Architecture & Operating Systems Department
Universitat Autònoma de Barcelona
Bellaterra (Barcelona), Spain
Email: jcgarcia@caos.uab.cat

***Abstract*—A suffix tree is a data structure very known to solve many classic string matching problems. One problem that can be solved with a suffix tree is the MUM-problem which is found in the MUMmer application. However, the search for MUMs presents challenges due to irregular and unpredictable data accesses in the suffix tree traversal. This paper enhances the traversal of a suffix tree by using contiguous nodes of suffix tree in depth levels and finding the correct child with a SIMD instruction. This little tweak allows having a reduction of search for MUMs time about 18%. Moreover an evaluation of search for MUMs in multicore architectures is reviewed and compared with this enhanced traversal which has a better speedup.**

I. Introduction

Firstly, we define the MUM-problem. Assume we are given an alphabet Σ^*, two sequences R, Q $\in \Sigma^*$, and a number L $>$ 0. The maximal unique matches problem (MUM-problem) is to find all sequences u $\in \Sigma^*$ with: $|u| \geq L$, u occurs exactly once in R and once in Q, and for any character a $\in \Sigma^*$ neither ua nor au occurs both in R and Q.

Let's assume the Reference sequence $R[0, \ldots, n-1]$ of size $|R| = n$ over an alphabet $\Sigma = \$, A, C, G, T$ which has a sentinel character $R[n-1] = \$$ that occurs nowhere else in the Reference genome and is lexicographically less than all the characters that occur in the alphabet. The suffixes of the Reference genome are zero indexed by their position in the original Reference by a full-text index data structure like a suffix tree.

Definition 1: A suffix tree, ST, for an n-character string R is a rooted directed tree with exactly n leaves numbered 0 to n. Each internal node, other than the root, has at least two children and each edge is labeled with a nonempty substring of R. No two edges out of a node can have edge-labels beginning with the same character. For any leaf i, the concatenation of the edge-labels on the path from the root to leaf i exactly spells out the suffix of R that starts at position i. That is, it spells out $R[i \ldots n]$. [1].

A search for a MUM between a Reference, $|R| = n$, and a Query, $|Q| = m$, can be done in a suffix tree in $O(n+m)$ steps. However it is possible to improve the search for MUMs in a suffix tree by using suffix links [2]. This improvement reduce the search for MUMs to $O(m)$ steps.

We also define a suffix link, which is an important trick to achieve linear complexity to search for MUMs in suffix tree.

Definition 2: A suffix link is a pointer from string $\overline{aw}$ to substring $\overline{w}$.

This paper is organized as follows, in Section II is resumed some of the previous work related to the MUM-problem. In Section III is explained how the MUM-problem is solved with a Suffix Tree. Section IV resumes the problems of the current implementation of suffix tree and the search for MUMs in the MUMmer application. Section V reviews a proposal to achieve the goal of reducing the execution time by using a new layout of suffix tree and SIMD instructions to traverse the suffix tree. Section VI shows how a naive implementation to search for MUMs in multicore architectures can be improved with the technique explained in Section V. Section VII resumes this paper.

II. Related work

Search for Maximal Unique Matches to do Whole Genome Alignment was proposed in [3]. There have been some previous work in the parallelization to search for matches in genomic data, like [4]–[6], however these works are focused in fixed patterns and read alignment. On the other hand, there have been achievements in parallelization of Whole genome alignment like [7]. The parallelization of searching MUMs with a suffix tree is research field not covered very deep. In [8] is explained the use of threads to search for MUMs but without access to source code to check their implementation; and in [9] there is an approach similar to this work, however, it is more focused with GPU and CPU hybrid architectures. Our approach is based on the use of multi-core architectures with OpenMP and Suffix Tree to search for MUMs.

III. Search for MUMs in suffix tree

A. Organization of suffix tree in memory

MUMmer application uses a suffix tree to search for MUMs. It follows an explanation of how suffix tree is stored in memory and how we traverse the suffix tree.

A suffix tree for string x=abab is shown in Figure 1.

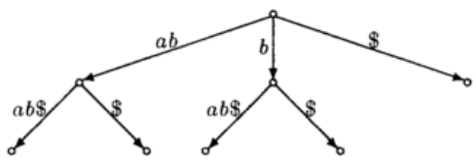


Fig. 1. Suffix tree for string x=abab, [10].

T_{leaf}					
leaf	$\overline{abab\$}$	$\overline{bab\$}$	$\overline{ab\$}$	$\overline{b\$}$	$\overline{\$}$
leaf number j	1	2	3	4	5
$T_{leaf}[j]$	**3**	**4**	nil	nil	nil

T_{branch}				
branching node	*root*	$\overline{ab}$	$\overline{b}$
node number	1	2	3
firstchild	2	**1**	**2**
branchbrother	nil	3	**5**
depth	0	2	1
headposition	1	3	4
suffixlink		3	1

Fig. 2. Tables to store suffix tree for string x=abab, [10].

Implementation of suffix tree suggested in [10] has two structures: a reference structure which consists of an address pointing to a leaf, or to a branching node. branchinfo structure has the following items:

- headposition: the head position of the branching node
- depth: the depth of the branching node
- suffixlink: the suffix link is always to a branching node
- firstchild: the reference to the first child
- branchbrother: the reference to the right brother, if this doesn't exist then it's `NULL`

For the suffix tree in Figure 1 the tables are shown in Figure 2.

The successors of a branching node are therefore found in a list whose elements are linked via the firstchild, branchbrother, and T_{leaf} references.

Every string occurring in a suffix tree ST can uniquely be described in terms of the nodes and edges of ST. This is formalized in the notion of *locations*:

Definition 3: Let ST be a suffix tree and $s \in words(ST)$. The *location* of s in ST, denoted by $loc_{ST}(s)$ is defined as follows:

- If $\overline{s}$ is an internal node, then $loc_{ST}(s) = \overline{s}$.
- If $\overline{s}$ is a leaf in ST, then there is a leaf edge $\overline{u} \stackrel{v}{\rightarrow} \overline{s}$ in ST and $loc_{ST}(s) = (\overline{u}, v, \epsilon, \overline{s})$.
- If there is no node $\overline{s}$ in ST, then there is an edge $\overline{u} \stackrel{vw}{\rightarrow} \overline{uvw}$ in ST such that $s = uv, v \neq \epsilon, w \neq \epsilon$ and $loc_{ST}(s) = (\overline{u}, v, w, \overline{uvw})$.

Following a path is the most important operation to search for MUMs. Function $TraverseSuffixTree$ describes this operation.

Definition 4: Let ST be a suffix tree. For each $s \in words(ST)$ and each string w the function $scanprefix : locations(ST) \times \Sigma^* \rightarrow locations(ST) \times \Sigma^*$ is specified as follows:

$$TraverseSuffixTree(loc_{ST}(s), w, ST) = (loc_{su}, v),$$

where $uv = w$ and u is the longest prefix of w such that $su \in words(ST)$.

Moreover, it is very important to have an efficient access from $loc_{ST}(cy)$ to $loc_{ST}(y)$. This access is provided by a function suffixlink, that uses the suffix links of the inner nodes as a "shortcut".

Definition 5: Let ST be a suffix tree. Function suffixlink, $suffixlink : locations(ST) \ \{root\} \rightarrow locations(ST)$ is defined as:

$$suffixlink(\overline{s}) = \overline{z} \text{ where } \overline{s} \rightarrow \overline{z} \text{ is the suffix link for } \overline{s}$$

$$suffixlink(\overline{u}, av, w, \overline{uavw}) = \begin{cases} loc_{ST}(v) & \text{if } \overline{u} = root \\ rescan(\overline{z}, av) & \text{otherwise} \end{cases}$$

where $\overline{u} \rightarrow \overline{z}$ is the suffix link for $\overline{u}$

Once we have defined the preliminaries in the MUM-problem, we need to answer the question: Where are the MUMs of R and Q of some minimum length L? We show the algorithm used to answer this question using a suffix tree, see Algorithm 1.

Algorithm 1: Search for MUMs in a suffix tree.

```
input : R, Q, L
output: List of MUMs of length ≥ L, with start
        position in R and Q and length
begin
    ST ← buildST(R);
    loc ← TraverseSuffixTree(Q[0],ROOT,ST);
    for i=1 to |Q| do
        if isLeafNode(loc) and loc.length ≥ L then
            /* Leaf saves the position of
               a suffix in ST.                 */
            ;
            if R[leaf − 1] ≠ Q[i − 1] then
                MUMcands ←
                saveMUMcand(R_leaf,i,length);
        if isRootNode(loc) then
            loc ← TraverseSuffixTree(Q[i +
            1],ROOT,ST);
        else
            suffixlink(ST,loc);
            loc ← TraverseSuffixTree(Q[i +
            1],loc.node,ST);
    /* Get unique MUMs from list of
       MUM-candidates.                         */
    ;
    MUMs ← cleanMUMcand(MUMcands)
```

In Algorithm 1 there are two functions related to suffix tree. TraverseSuffixTree performs the traversal of the suffix tree with the current string and it starts in some given node. suffixlink is a function which gets the suffix link from the current node pointed out by loc.node. If b is an internal node v of suffix tree, then the algorithm can follow its suffix link to a node $s(v)$. If b is not an internal node, then the algorithm can back up to the node v just above b. If v is the root, then the search starts in root. But if v is not the root, then the algorithm follows the suffix link from v to $s(v)$.

IV. Suffix tree bottleneck

Suffix tree is the archetypical index structure used in bioinformatics. For large-scale bioinformatics applications, however, memory consumption really becomes a bottleneck

and a factor of higher latency. Moreover, the algorithm to search for MUMs involves comparing the search of a character to the character stored at a specific node at every level of the tree, and traversing a child node based on the comparison results. Only one node at each level is actually accessed, resulting in ineffective cache line utilization, to the linear storage of the tree. Result of the comparison is required before loading the appropriate next child node. Moreover, meanwhile the Algorithm 1 shows a $O(m)$ complexity by using suffix links, this feature has the disadvantage of a quite random access to the next node used to traverse the suffix tree. As a consequence a search for MUM typically involves a long-latency at each level of the tree, leading to ineffective utilization of the processor resources.

The best scenario (short-latency and no TLB/cache miss) in a search for MUM is a layout of a whole branch in a cache line size. The worst scenario (long-latency and TLB/cache miss) is a layout of only one branch read in a cache line size. In Figure 1 the suffix tree is implemented with two tables as in Figure 2. The layout of branch table has a poor temporal and spatial locality.

Use of tables leaf and branch in Figure 2 helps to traverse a suffix tree, but since each branch node is implemented as a linked list this is a serious performance issue. In Figure 3 is shown some of the first nodes of a suffix tree, the squared numbers represent the number of node in the branch table. It can be shown that only a few nodes are contiguous, a contiguous linked list has a better locality and less latency.

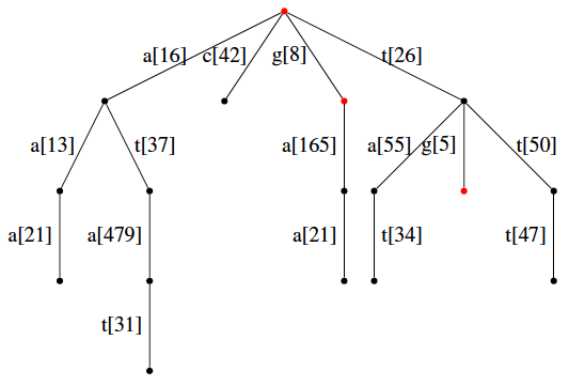


Fig. 3. First nodes of suffix tree. Red dots are adjacent memory references starting at root node.

The linked list have items in disjoint areas of memory. To traverse the list the cache lines cannot be utilized effectively. One could say that the linked list is cache line hostile, or that the linked list maximizes cache line misses. The disjoint memory will make traversing of the linked list slow because RAM fetching will be used extensively.

A. *Memory reference locality*

This layout in Figure: 2 and 3 of a suffix tree and the random search of strings do not help to a better use of processor resources. The use of suffix links reduces the time complexity of a search for a MUM in a suffix tree but it is not very cache-friendly.

As a proof of a poor memory reference locality of this data structure, two experiments were carried out. The features of the computer used are: 2 Processor Intel(R) Xeon(R) E5645 @ 2.4GHz of 6 cores each one, 32KB L1 cache, 256KB L2 and 12MB L3 shared cache per socket. RAM: 96 GB. GCC 4.7.0 with OpenMP support + Linux.

The genomes used were a reference genome of 312Kbp[1] and 169Mbp, a query genome of 2.91Gbp and a minimum length of MUM of 20bp.

By using the memory heriarchy, we may overlap the latency of a random traversal in a suffix tree. The first experiment is with a small suffix tree (reference of 312Kbp), which is kept in the L3 cache (6MB of memory consumption for suffix tree), the query genome of 2.91Gbp is kept in main memory. The total time to search for MUMs was 149.58 seconds. The second experiment is with a bigger sufffix tree (reference of 169Mbp), which is kept in main memory (3380GB of memory consumption for suffix tree), the query genome of 2.91Gbp is kept in main memory. The total time to search for MUMs was 268.58 seconds. Furthermore, the functions in Algorithm 1 were profiled to get how many times are accessed in order to have an idea of the bottlenecks while traversing a suffix tree with suffix links. In experiment 1, 6.18% of times the traversal started in the root of suffix tree. 60.38% of times suffix links were used. By using suffix links 59.99% of times the function rescan was used. These measures show how heavy and intensive is the used of suffix links and how it may degrade the performance. In experiment 2, 4.12% of times the traversal started in the root of suffix tree. 66.47% of times suffix links were used. In suffix links 69.23% of times the function rescan was used.

V. Enhanced traversal of Suffix Tree with SIMD instruction

In Section IV-1 is shown where the bottlenecks are in the MUMmer application. Function rescan involves to find the correct child to traverse with next node. However, because nodes are not arranged in a sequential way every access to a child node incurs in a high latency. To avoid this latency, a new arrangement is made in the suffix tree. So that suffix tree is traversed with algorithm breadth-first search and rearranged all nodes by depth level. This new order still has a latency but it is reduced the access time to child nodes because every child node is next to the other. If traversal of suffix tree requires to find the correct edge to continue, then this new order allows reading every child node, and its information, in a sequential way. Since we are focused in the search for MUMs time, this new layout of the suffix tree gives the opportunity to use SIMD instruction to find the correct child node as it is used in [11].

To find the correct child in MUMmer application involves to read every child node and compare the first character against the current character in the query string. Therefore there are as many comparisons as children nodes are in the current parent node, because MUMmer application works with DNA the maximum number of children are 4 (a,c,g,t).

With the current advances in SIMD instructions we can use a SIMD instruction to find the correct child with only one

[1]Measure unit in DNA genomes.

instruction. Then use the mask provided by SIMD instruction to get the offset from the first child and continue with the traversal of suffix tree. It is used a library[2] which provides the functions to store, read and compare the first character of every child node.

An example of the use of this library to find the correct child:

```
Vec4i a(65,67,71,84);
Vec4i b(71,71,71,71);
Vec4i c = a==b; //c = (0,0,-1,0)
```

The result of this comparison is used to compute the offset to find the correct child and continue with the lcp comparison of the current string in that edge and the query string.

A. Experimental evaluation

Goal is to reduce the execution time in the search for MUMs between a Reference and Query genome of some minimum length. To improve time, an evaluation of the search for MUMs algorithm was measured and findings are that the most time-consuming function of the algorithm is the traversal of suffix tree. A new layout of suffix tree is proposed and based on that layout the use of SIMD instructions to optimize the part of algorithm to find the correct child.

This little tweak allows having a reduction in the execution time. This reduction in time is because a whole while-loop is changed by a single SIMD instruction. Moreover, the contiguous children nodes have the advantage of reducing the latency and information stored in every child node can be loaded quicker.

To verify that our approach, can have a better performance to search for MUMs between a Query and Reference genome, we check that output of MUMmer and our approach has the same MUMs and with a reduction of execution time. This test was carried out in the following node:

- Hardware:
  - 2 Processor Intel(R) Xeon(R) E5645 @ 2.4GHz of 6 cores each one, 32KB L1 cache, 256KB L2 and 12MB L3 shared cache per socket.
  - RAM: 96 GB
- Software:
  - Linux Kernel 2.6.32-220.el6.x86_64 #1 SMP
  - gcc 4.7.0 with OpenMP support
- Genomes:
  - Reference:
    - Fly fruit genome, 169 [Mbp]
    - Silkworm genome, 487.6 [Mbp]
  - Query:
    - Fly fruit genome, 166 [Mbp]
    - Red imported fire ant genome, 335.7 [Mbp]

In the first experiment, Figure 4, it is shown a reduction in execution time of 17% compared to the current implementation of MUMmer application. As it was explained before, this reduction in time is because children nodes are contiguous (less latency) and to find the correct child uses SIMD instructions (less instructions).

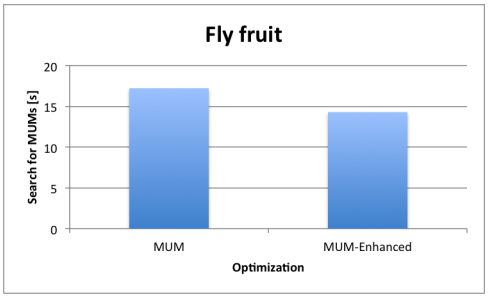
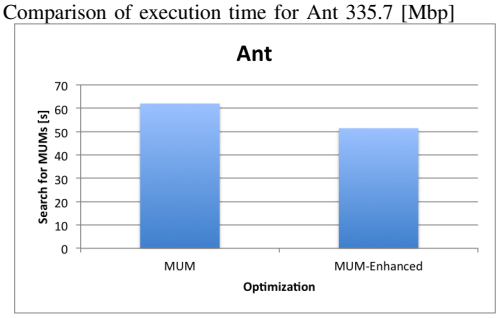


Fig. 4. Comparison of execution time for Fly fruit 166 [Mbp]

In the second experiment, Figure 5, it has a bigger genome size and it is shown that we have a reduction in time of 17.06%.

Fig. 5. Comparison of execution time for Ant 335.7 [Mbp]



VI. Search for MUMs in Multicore Architectures

The sequential version of the MUM search between a Reference and Query genome trades high computation time when executed in a single machine. Previously the algorithm to search for MUMs was described. Now a proposal of a parallelization of MUM search within multi-core architecture is explained.

There are two resources to improve in this algorithm: Memory usage and CPU. The former was generally improved because it allows being executed in architectures where there is no memory restriction. To improve the performance of the algorithm a data-level parallelism technique is deployed.

This technique is divided in three phases following:

1) Splitting query genome data (chunks) according to the number of available cores using 1 thread per core. Chunk size is computed with the total of threads used, thus the chunk size is fixed.
2) Parallel execution of the task of Algorithm 1 for every chunk, after finishing its task every thread has its own list of MUM-candidates.
3) Get the final list of MUMs from every MUM-candidate list of all threads.

[2]www.agner.org/optimize

To get a MUM requires an important feature its uniqueness. Uniqueness can only be found when a whole genome is checked, see Figure 6. If some part of it is only evaluated we could miss the rest of the genome. In other words, after finding MUMs within a chunk it is not possible to determine if the MUM found is or not a "unique" MUM, globally in the query genome, because these MUMs are unique only in the chunk that has been read, the rest of the genome it is not known until all query genome has been read. In Figure 6 is shown the consequences of using chunks for query genome. Meanwhile we may speed up the MUM search we need to deal with the discard of MUM-candidates which are not a real MUM. So that, after finding all MUM-candidates for every thread we need to verify which of MUM-candidates are MUMs.

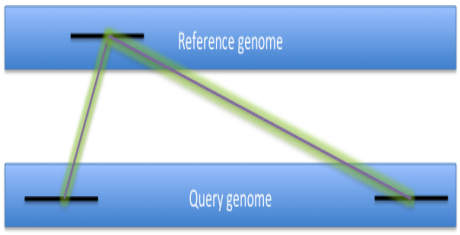


Fig. 6. MUM in reference genome but not in query genome.

Phase 3 extracts the list of MUM from the set of MUM-candidates of every chunk in the query genome. In order to get the real MUMs we filter MUM-candidates that are overlapped by bigger MUM-candidates.

Implementation

Split query genome: As it was previously explained, the approach is to use a fixed size division of query genome. To split query genome the algorithm needs to know in advance how many chunks will be used (one thread/one chunk). Then every chunk is computed with two pointers (left and right end) which points out query genome in main memory.

Finding MUMs: The parallelization is carried out with OpenMP, the Algorithm 1 is applied for every chunk. The algorithm to search for MUMs is a process which can be executed without any data dependency.

Firstly, from Algorithm 1 some variables are shared among the threads in our OpenMP implementation: Suffix Tree, Reference genome and Query genome. Secondly, some variables are private for every thread: the location pointer to Suffix Tree, the pointer to the chunk of Query genome assigned to each thread and the array of MUM-candidate type.

A. Experimental evaluation in Multicore archictectures

As it was described in Section V-A, same node is used to measure the execution time by using the previous naive technique.

In Figure 7 is shown two experiments were the naive technique is deployed with OpenMP and one with the enhanced traversal of suffix tree. In first experiment, Fly fruit, a reduction in execution time is about 30% compared to naive technique. This reduction in time is because the new layout in suffix tree provides a better locality and the reduction of instructions with the SIMD instructions. In second experiment, Ant, there is a minor reduction with naive technique about 27%. However, this second experiment shows a better use of resources in multithreading processors with bigger genomes. The speedup is 34% better in the SIMD technique with 24 threads. Goal is

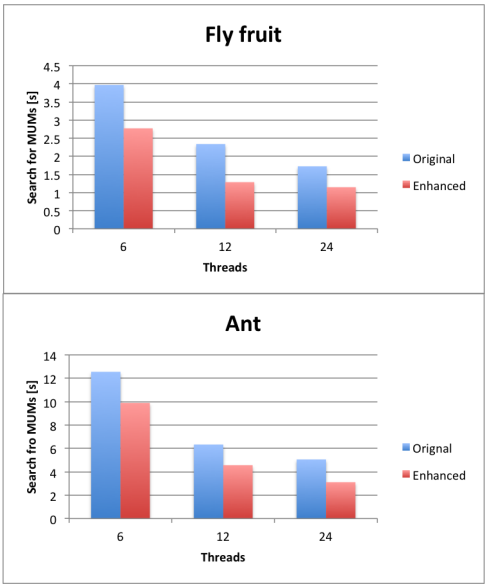


Fig. 7. Comparison of optimization in Multicore architectures.

to reduce the execution time to search for MUMs. Therefore, experiments and measures with multicore architectures are focused in the search for MUMs algorithm, and more in detail in the traversal of suffix tree (main bottleneck in Algorithm 1).

VII. Conclusion

Traverse of suffix tree and use of suffix links are bottlenecks for a better processor performance. In suffix trees larger than the Last Level of Cache, the performance is dictated by the number of cache lines loaded from the memory, and the hardware features available to hide latency due to potential TLB and Last Level of Cache misses. Firstly, an enhanced traversal of suffix tree for a single thread shows a better use of processor resources and a reduction in search time. Secondly, a multicore technique is explained and it is compared with the enhanced traversal of suffix tree which has a better speedup.

Traversal of suffix tree is the main bottleneck in the MUM-problem and it has been given an optimization with SIMD instructions in order to reduce the execution time. However, there are some improvements to be done. For instance, the lcp comparison at each node can be done with SIMD instructions and to have a reduction in execution time.

Moreover, random traversal of suffix tree could be avoided by using some kind of scheduler to traverse suffix tree in a top-down fashion and taking advantage of data locality in last level of cache.

Acknowledgment

This work was supported by grant from "Ejecución eficiente de aplicaciones multidisciplinares: nuevos desafíos en

la era multi/many core", with reference TIN2011-28689-C02-01.

REFERENCES

[1] D. Gusfield, *Algorithms on strings, trees, and sequences: computer science and computational biology*. Cambridge University Press, 1997. [Online]. Available: http://www.amazon.ca/exec/obidos/redirect?tag=citeulike09-20&path=ASIN/0521585198

[2] W. Chang and E. Lawler, "Sublinear expected time approximate string matching and biological applications," *Algorithmica*, 1991. [Online]. Available: http://www.eecs.berkeley.edu/Pubs/TechRpts/1991/CSD-91-654.pdf

[3] A. L. Delcher, S. Kasif, R. D. Fleischmann, J. Peterson, O. White, and S. L. Salzberg, "Alignment of whole genomes MUMMER," *Nucl Acids Res*, vol. 27, no. 11, p. 2369, 1999.

[4] M. Oğuzhan Külekci, W.-K. Hon, R. Shah, J. Scott Vitter, and B. Xu, "Ψ-RA: a parallel sparse index for genomic read alignment." *BMC genomics*, vol. 12 Suppl 2, p. S7, Jan. 2011. [Online]. Available: http://www.pubmedcentral.nih.gov/articlerender.fcgi?artid=3194238&tool=pmcentrez&rendertype=abstract

[5] H. Mongelli, "Efficient Two-Dimensional Parallel Pattern Matching with Scaling."

[6] C. S. Kouzinopoulos, P. D. Michailidis, and K. G. Margaritis, "Parallel processing of multiple pattern matching algorithms for biological sequences : Methods and performance results," *Systems and Computational Biology - Bioinformatics and Computational Modeling*, 2005.

[7] X. Meng and V. Chaudhary, "Exploiting multi-level parallelism for homology search using general purpose processors," *October*, 2005. [Online]. Available: http://www.computer.org/portal/web/csdl/doi/10.1109/ICPADS.2005.152

[8] G. Encarnac and N. Roma, "Advantages and GPU Implementation of High-Performance Indexed DNA Search based on Suffix Arrays," *Architecture*, pp. 49–55, 2011.

[9] M. C. Schatz, C. Trapnell, A. L. Delcher, and A. Varshney, "High-throughput sequence alignment using Graphics Processing Units." *BMC bioinformatics*, vol. 8, p. 474, Jan. 2007. [Online]. Available: http://www.pubmedcentral.nih.gov/articlerender.fcgi?artid=2222658&tool=pmcentrez&rendertype=abstract

[10] S. Kurtz, "Reducing the Space Requirement of Suffix Trees," *Software Practice and Experience*, vol. 29, no. February, pp. 1149–1171, 1999.

[11] C. Kim, J. Chhugani, N. Satish, E. Sedlar, A. D. Nguyen, T. Kaldewey, V. W. Lee, S. A. Brandt, and P. Dubey, "FAST : Fast Architecture Sensitive Tree Search on Modern CPUs and GPUs," *Architecture*, pp. 339–350, 2010. [Online]. Available: http://portal.acm.org/citation.cfm?id=1807167.1807206