Available online at www.sciencedirect.com
ScienceDirect
Procedia Computer Science 00 (2012) 1–10
Procedia Computer Science

International Conference on Computational Science, ICCS 2012

Search of maximal unique matches in multi-core architectures

Julio César García Vizcaíno[1], Antonio Espinosa[1], Juan Carlos Moure[1], Porfidio Hernández[1]

Computer Architecture and Operating Systems, Campus UAB, Edifici Q, 08193 Bellaterra (Barcelona), Spain.

Abstract

Maximal Unique Matches are common substrings that match a reference and a query genome. They are exact, unique and maximal; that is, they cannot be extended in left or right direction without incurring a mismatch. The computation of MUMs in large genomes is a heavy and repetitive task, so there is a fair chance of parallelize and execute this search in multi-core architectures. This research resembles an approach to find MUMs in genomic sequences with a suffix tree in parallel way. The reference genome is indexed by using a suffix tree in main memory and then parallelized algorithm finds the MUMs of query genome which is readed by several threads. This approach is based on MUMmer, a genome alignment tool, which is able to find Maximal Unique Matches (MUMs).

Keywords: Indexed Search, Bioinformatics, Maximal Unique Match, Multi-core architectures, Parallelization

1. Introduction

The problem of searching maximal unique matching for a minimum length between a reference string and a query string has been identified in several applications, one of them is MUMmer [1]. Altough MUMmer's algorithm can perform searches of maximal unique matches (MUMs) with the following use of resources:

- High use of main memory to store the reference string.
- A null use of multi-core architectures.

If the length of reference and query are very huge, the amount of operations to perform in the search of MUMs increases, see Table 1. The use of parallelism could help reduce the execution time for the search of maximal unique matches. One approach of parallelism is to take advantage of multi-core architectures nowadays, Figure 1 shows the process of our data-level parallelism technique.

Email addresses: juliocesar.garcia@caos.uab.es (Julio César García Vizcaíno), antonio.espinosa@caos.uab.es (Antonio Espinosa), juancarlos.moure@uab.es (Juan Carlos Moure), porfidio.hernandez@uab.es (Porfidio Hernández)

Data structure	L [bp]	Search operations	Search [s]	Memory usage [MB]
Suffix tree	20	$9{,}87\times10^{18}$	169189,4	48665,12

Table 1: Search of Maximal Unique Matches between a reference genome (2960,21Mbp) and query genome (2716,96Mbp)

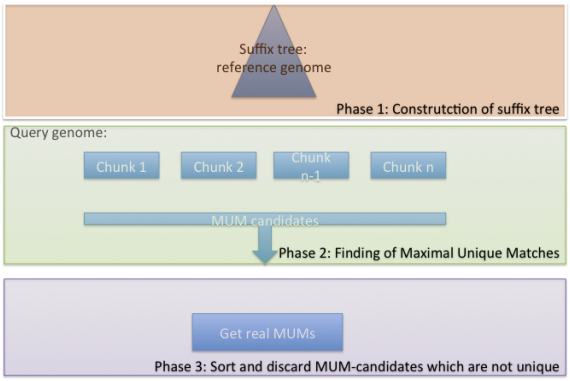

Figure 1: Data-level parallelism in multi-core architectures for whole genome alignment.

1.1. The MUM: an heuristic approach

Although a pair of conserved genes rarely contain the same entire sequence, they share a lot of short common substrings and some of them are indeed unique to this pair of genes. For example the following two sequences, R and Q:

R=ac ga ctc a gctac t ggtcagctatt acttaccgc$
Q=ac tt ctc t gctac ggtcagctatt c acttaccgc$

It is clear that sequences R and Q have many common substrings, they are: ac, ctc, gctac, ggtcagctatt, acttaccgc.
Among those five common substrings, ac is the only substring that is not unique. It occurs more than once in both sequences. You can also observe that actually a, c, t, and g are common substrings of R and Q. However, they are not maximal, i.e. they are contained in at least one longer common substrings. We are only interested in those that are of maximal length.
Our aim is to search for all these short common substrings. Given genomes R and Q, we need to find all common substrings which are unique and of maximal length. Each of such common substrings is known as Maximum Unique Match (MUM). For almost every conserved gene pairs, there exist at least one MUM which is unique to them.
For example, assuming d = 3, sequences R and Q in the previous example has four MUMs: ctc, gctac, ggtcagctatt, acttaccgc. Substring ac is not an MUM because its length is smaller than the value of d and it is not unique to both sequences.

R=ac ga ctc a gctac t ggtcagctatt acttaccgc$
Q=ac tt ctc t gctac ggtcagctatt c acttaccgc$

The concept of MUM is important in whole genome alignment because a significantly long MUM is very likely to be part of the global alignment.

1.1.1. Finding MUMs in a suffix tree

The key idea in this method is to build a suffix tree for genome R, a data structure which allows finding, all distinct subsequences in a given genome.

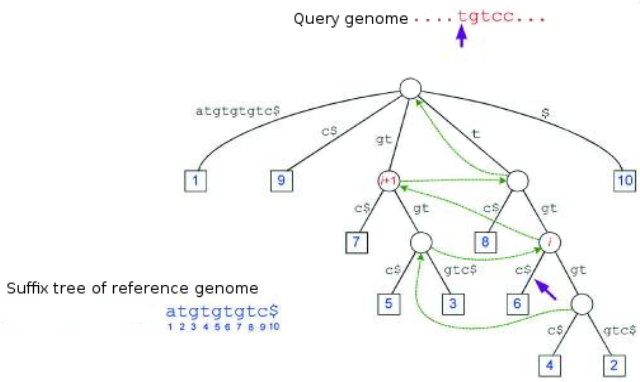

Figure 2: Suffix tree and its use to find matches between a reference and query genome [2].

A suffix tree is a data structure which indexes a whole string in every suffix. It allows finding all matches between a reference genome (suffix tree) and a query genome. It can also check for unique matches in the reference very easily just to check if the match ends in a leaf node and by checking the character immediately preceding the start of this match it is possible to determine whether it is a maximal match. To understand the efficient search of matches in a suffix tree, time complexity is $O(m)$ where m is the length of query genome, it is used a suffix link. A link points from a source node s in suffix tree to another destination node d in suffix tree if the string label from the root node to node d is equal to the label from root node to s with the first character removed. In Figure 2 is shown the suffix tree, green lines are suffix links and boxes are start position of every suffix. For instance, the string label of node i in Figure 2 is tgt and next suffix $i + 1$ is gt. So that is exactly the tree position corresponding to the next position in the query genome.

In Figure 2 the match gtc which starts at $i + 1$ in the query genome and node 7 in the suffix tree is not maximal on the left because the preceeding character in both strings is a t.

By construction, the location of a match in the suffix tree represents a substring of the subject sequence which maximaly matches a prefix of `query suffix`. Thus it is only necessary to verify that, the match is long enough, that it is unique in the subject sequence and that the match is also left maximal. This is done as follows:

1. Does `match` represent a substring of length at least `minimum match length`?
2. Does `match` correspond to a leaf edge? Then then the string represented by the match is unique in the subject sequence.
3. Is the match left maximal? This is true if one of the following conditions hold:
   - The suffix of the query currently considered is the first suffix, or
   - The string represented by `match` is a prefix of the subject string, or
   - The characters immediately to the left of the matching strings in the subject sequence and the query sequence are different

If all conditions 1-3 are true, then this MUM is stored in a list of MUM-candidates as it is shown in Figure 3. If a match is found but it doesn't reach a leaf node then this match is dropped as it is shown in Figure 3. A MUM-candidate is saved in an array of type MUM-candidate which has the following information:

- Position in reference sequence.
- Position in query genome.
- Length of match.

The current serial search method is summarized in Algorithm 1, the first suffix is **always** searched from the root node in Suffix Tree and rest of suffixes are searched with suffix links from the node in Suffix Tree from previous search.

MUMs can cover 100% of the known conserved gene pairs. Moreover, finding all MUMs can be done in almost

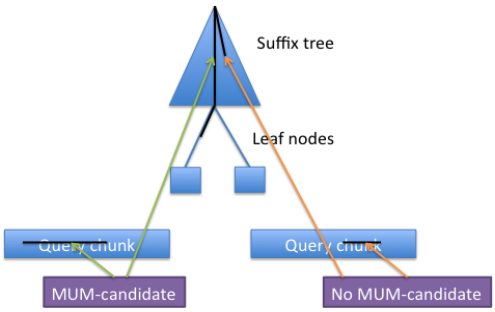

Figure 3: Finding MUMs in a suffix tree. Blue box represents a leaf node and it saves the start position of its suffix.

Algorithm 1 MUM search of a query and reference genome.

```
1: for all suffixes in query genome do
2:     Check if current match is a MUM-candidate.
3:     if current node in Suffix Tree is the root node then
4:         Start traversal from root node and from the beginning of current suffix.
5:     else if current node in Suffix Tree is an intermediate node then
6:         Search for suffix link to jump to another node in Suffix Tree.
7:         Continue search of match with rest of suffix from the node which suffix link points out.
```

linear time because of the use of suffix links without this artifact we should start every traversal in Suffix Tree from root node adding more execution time because we scan a longer string. Since a Suffix Tree can be queried several times, it is very likely to have more than one query at the same time. So far, Algorithm 1 is a serial execution because it cannot check for more than one suffix at the same time. This problem of searching MUM has a time complexity of $O(m + k)$ where m is the length of the query genome and k is the number of maximal unique matches of some minimum length. This problem is a very high intensive computing task, for every substring in the query genome the search for a maximum unique match has to be performed.

2. Related work

Search of Maximal Unique Matches to do Whole Genome Alignment was proposed in [3]. There have been some previous work in the parallelization of search of matches in genomic data, like [4, 5, 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, there is only one approach in [8] but without access to source code to check theirs implementation and it is more focused with GPU and CPU hybrid architectures and suffix array.

3. Parallel algorithm

The availability of multi-core architecture makes possible to execute the same task (MUM search) with a different kind of data (chunks of query genome).
The sequential version of the MUM searh trades high computation time when executed in a single machine.
Previously the algorithm for sequence alignment was described in detail. Now our own proposal of a parallelization of MUM search within multi-core architecture is explained.
There are two resources to improve in this algorithm:

- Memory usage.
- 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 in advance to genome alignment.
Our 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.
2. Parallel execution of the task of Algorithm 1 for every chunk then every thread has its own list of MUM-candidates.
3. Get the final list of MUMs from every MUM-candidate list of all threads.

The division of genome data was used using the paradigm of data-level parallelism which consists of a generation of chunks of a query sequence with a fixed size. The chunk size was computed with query genome length divided by number of available threads.
The main idea behind using a Maximal Unique Match (MUM): it is possible to cover a huge region of a genome when reference and query genomes are very closely related. However to get a MUM, it requires an important feature its uniqueness. Uniqueness can only be found when a whole genome is checked, see Figure 4. 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 4 is shown the consequences of using chunks for query genome. Meanwhile we may speedup 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 MUM.
In Figure 1 we can see phase 3 which involves to extract the list of MUM from the set of MUM-candidates of every

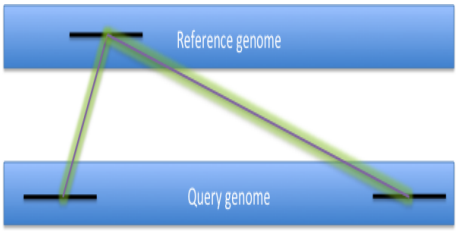

Figure 4: MUM in reference genome but not in query genome.

chunk in the query genome. In order to get the real MUMs we apply the Algorithm 2 to the set of MUM-candidates.

4. 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. 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 MUMs is a process which can be executed without any data dependency. However, when we split a query sequence the following problems arise:

- Chunk size is a performance factor.
- Different MUM-candidate in query sequence:

Algorithm 2 Get MUMs from MUM-candidates of a query genome and reference genome.

```
Set of MUM-candidates for all chunks.
Quicksort of MUM-candidates by position in Query genome.
for i = 0 a N do
    A[i].Good = TRUE
for i = 0 a i < N − 1 do
    if !A[i].Good then
        Continue.
    i_diag = A[i].Q − A[i].R
    i_fin = A[i].Q + A[i].Len
    for j = i + 1; j < N&&A[j].Q ≤ i_fin; j + + do
        if !A[j].Good then
            Continue.
        j_diag = A[j].Q − A[j].R
        if i_diag == j_diag then
            j_extent = A[j].Len + A[j].Q − A[i].Q
            if j_extent > A[i].Len then
                A[i].Len = j_extent
                i_end = A[i].Q + j_extent
            A[j].Good = FALSE
        else if A[i].R == A[j].R then
            olap = A[i].Q + A[i].Len − A[j].Q
            if A[i].Len < A[j].Len then
                if olap ≥ A[i].Len/2 then
                    A[i].Good = FALSE
                    Break.
            else if A[j].Len < A[i].Len then
                if olap ≥ A[j].Len/2 then
                    A[j].Good = FALSE
            else
                if olap ≥ A[i].Len/2 then
                    A[j].Tentative = TRUE
                    if A[i].Tentative then
                        A[i].Good = FALSE
                        Break.
        else if A[i].Q == A[j].Q then
            olap = A[i].R + A[i].Len − A[j].R
            if A[i].Len < A[j].Len then
                if olap ≥ A[i].Len/2 then
                    A[i].Good = FALSE
                    Break.
            else if A[j].Len < A[i].Len then
                if olap ≥ A[j].Len/2 then
                    A[j].Good = FALSE
            else
                if olap ≥ A[i].Len/2 then
                    A[j].Tentative = TRUE
                    if A[i].Tentative then
                        A[i].Good = FALSE
                        Break.
for i = j = 0; i < N; i + + do
    if A[i].Good then
        if i ≠ j then
            A[j] = A[i]
        j + +
N = j
for i = 0; I < N; i + + do
    A[i].Good = FALSE
```

- Additional MUMs: this is because we cannot know if the MUM-candidate is unique or not.
- Lost MUMs: we may skip MUMs because of the lack of control in pointers to query genome, that is every thread should point to the real value in the query genome.

The total number of iterations is the number of chunks created. Every iteration means the whole search of MUMs within a chunk, following the Algorithm 1, if the right end of chunk is not the end of the query sequence and there are still nucleotides to match then traversal of suffix tree until it occurs a mismatch or a MUM-candidate is found.
The key factors in this phase are:

- Number of chunks: every chunk is private for each thread.
- Size of chunks: delimiters for chunks are private for each thread.
- Number of threads.

Get real MUMs

This phase is applied with Algorithm 2 for the set of MUM-candidates. These are ordered with quicksort according to position in query sequence. Every thread has found a set of MUM-candidates from previous phase but all threads don't produce the MUM-candidates in order. That's why a quicksort is required.
After quicksort we need to get the real MUMs. A real MUM is unique in the whole reference and query genome. Those MUM-candidates which are overlapped by bigger MUMs are discarded too, see Figure **??** where MUM-candidate in orange colour is discarded because another MUM-candidate has a larger length and they have same match in the reference genome.

5. Experiments and results

To verify that our approach, can have a better performance to search MUMs of a query and reference genome we check that output of MUMmer and our approach to check that they produce the same MUMs. 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
  - Likwid 3.0
- Genomes:
  - Reference: Human chromosome 21 single fasta file
  - Query: Mouse chromosome 16 single fasta file

The main objective of the test was to check the performance to search MUMs in multi-core architectures by using OpenMP (threads). The variables to control were:

1. Number of chunks for query genome.
2. Number of threads: from 1 to 24 threads with affinity.
3. OpenMP scheduling: static with chunk size of 1.

Moreover, the use of Memory Bandwidth was measured with the tool likwid[1], the use of Memory Bandwidth was checked only in the search of MUMs. The times were collected for the total execution time and the time spent in the parallel region of OpenMP. This parallel region is the algorithm to search MUMs.
One key aspect of the several tests deployed was to assure the execution of every thread in a single core, that is no other thread is competing for the same cache. Affinity is a requirement because we need to ensure the scalability of our proposal, without affinity even a number of threads below the number of cores might be in the same core. Affinity ensures us a proper use of performance in a multi-core architecture.
The construction of the suffix tree for the reference genome of length 35.45[Mbp] takes 20.66[s] to construct and it uses 608[MB] of memory. The minimum length for a MUM was fixed in all tests, 20[bp]. The query sequence of length 95.28[Mbp] was divided in chunks equal to the available threads, the total number of MUMs found were 102844.
Two times were collected during the execution of the experiments: total execution time and execution time to find mums.
Figure 5 shows the speedup for the parallelization of searching of MUMs. Moreover from this Figure 5 we can conclude that we got a near linear speedup with 12 threads, which is the maximum number of cores. Then there is a reduction in the linear speedup because we start using more than 1 thread per core and this causes a degradation in the phase of searching MUMs. Moreover, because the rest of the application has not be improved yet; that is phase 1 and 3 in Figure 1, the general speedup is not improved as it is shown in Figure 5.
Besides speedup metric, we measured the use of Memory bandwidth to check how the suffix tree is queried with

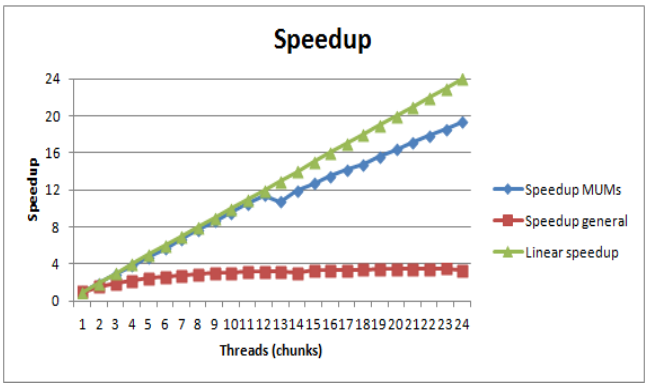

Figure 5: Speedup of our approach to search MUMs in multi-core architectures with OpenMP.

several threads. Figure 6 shows how much of Memory Bandwidth average is used in three different levels: L2 cache, L3 cache and Main memory. The maximum Memory Bandwidth was measured with benchmark stream 5.8 with a result of 20943[MB/s]. Within the first 6 threads the use of MEM BW is about the half of the maximum MEM BW with other threads, this is because we are only using one socket of our node. When we increase the number of threads up to 12 we reach the maximum use of MEM BW in our proposal. After 12 threads the use of MEM BW decreases and we can confirm with Figure 5 the lower speedup with more than one thread per core in our code.
CPI in Figure 7 was also measured as a metric to check if there are still improvements to do in our approach. As it is shown in Figure 7, we have a high with more than 12 threads and the consequence of this fact is a slower algorithm. High CPI is mainly caused by the traversal of Suffix Tree and the search for suffix link in Algorithm 1. In order to reduce the high CPI and increase the Memory Bandwidth and indeed improve the performance of our approach the following proposals may be used:

- Prefetching: this may work because we know in advance what information from suffix tree and query sequence we need in every traversal of suffix tree. If we can allow the use of some part of suffix tree be stored and read in cache by several threads which request that substree. However, because we use suffix links to jump to another part of suffix tree so that a new part of suffix tree would have to be read.

[1]Measure hardware performance counters on Intel and AMD processors.

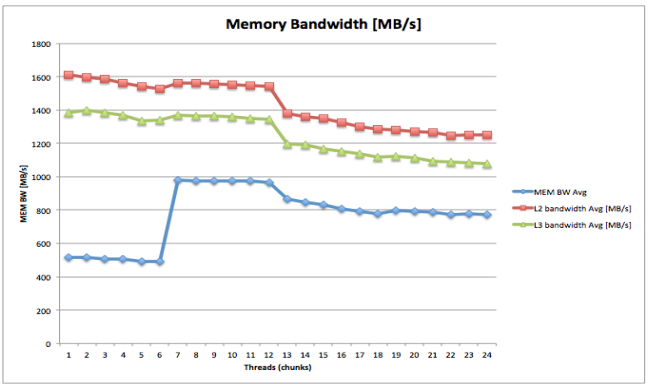

Figure 6: Use of memory bandwidth of our approach to search MUMs in multi-core architectures with OpenMP.

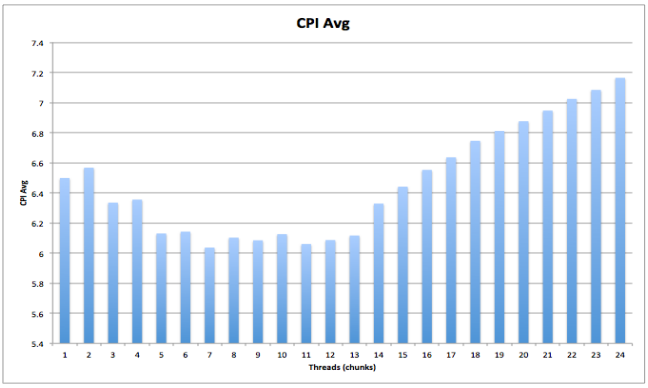

Figure 7: CPI of our approach to search MUMs in multi-core architectures with OpenMP.

- Explicit cache management: this is a obvious consequence of previous bullet because we may need to store part of the suffix tree in some cache-level of processor.
- An improved traversal of suffix tree would improve the overall performance of search of MUMs. So far, one thread traverses the suffix tree if we add multithread traversal of suffix tree to find next edge in the match path we would reduce the execution time.

6. Conclusions

This paper presents an evaluation of performance to search MUMs of a query and reference genome in multi-core architectures with OpenMP. The search of MUMs is performed in a suffix tree and the list of MUMs can help in the process of whole genome alignment. The results shows that the heaviest section of searching MUMs in a suffix tree is improved with the use of a multi-core architecture. Although we got a near linear speedup with only one thread per core, there is more improvements to do. These improvements involve a better use of resources (memory and CPU) when we have more than thread per core. In spite of suffix tree is a good data structure to string matching, when is used in multi-core architectures it shows a slow performance (high CPI) to search MUMs.
From the results obtained we conclude that bottleneck is in our data structure: suffix tree. We require to handle a strategy to traverse a suffix tree in multi-core architectures, this approach only queries the suffix tree for one string at a time. However, the traversal of suffix tree is still done in a serial way.

Acknowledgements

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] A. L. Delcher, S. L. Salzberg, A. M. Phillippy, Using MUMmer to identify similar regions in large sequence sets., Current protocols in bioinformatics editoral board Andreas D Baxevanis et al Chapter 10 (1934-340X (Electronic) LA - eng PT - Journal Article SB - IM) (2003) Unit 10.3.
URL http://www.ncbi.nlm.nih.gov/pubmed/18428693

[2] A. L. Delcher, A. Phillippy, J. Carlton, S. L. Salzberg, MUMmer: comparative applications Fast algorithms for large-scale genome alignment and comparison, Nucleic Acids Research 30 (2002) 2478–2483.
URL http://www.tigr.org/software/mummer/applications.html

[3] A. L. Delcher, S. Kasif, R. D. Fleischmann, J. Peterson, O. White, S. L. Salzberg, Alignment of whole genomes MUMMER, Nucl Acids Res 27 (11) (1999) 2369.

[4] M. Ouzhan Külekci, W.-K. Hon, R. Shah, J. Scott Vitter, B. Xu, Ψ-RA: a parallel sparse index for genomic read alignment., BMC genomics 12 Suppl 2 (2011) S7. doi:10.1186/1471-2164-12-S2-S7.
URL http://www.pubmedcentral.nih.gov/articlerender.fcgi?artid=3194238&tool=pmcentrez&rendertype=abstract

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

[6] C. S. Kouzinopoulos, P. D. Michailidis, K. G. Margaritis, Parallel Processing of Multiple Pattern Matching Algorithms for Biological Sequences : Methods and Performance Results.

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

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