Search for Maximal Unique Matches in Multi-core architectures

Julio Cesar Garcia Vizcaino, Antonio Espinosa, Juan Carlos Moure, and Porfidio Hernandez

Universitat Autónoma de Barcelona (UAB), Barcelona, Spain

{jcgarcia,aespinosa}@caos.uab.cat
{juancarlos.moure,porfidio.hernandez}@uab.es

Abstract. Maximal Unique Matches (MUMs) are common substrings that match a Reference and a Query genome. They are exact, unique and maximal. The search for 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 evaluates a parallelization of the search for MUMs in genomic sequences within multi-core architectures. The Reference genome is indexed by using a Suffix Tree or Enhanced Suffix Array in main memory and then a parallelized algorithm searches for MUMs between Reference genome and Query genome. Query genome is read by several threads in chunks of fixed size. This approach is based on MUMmer, a genome alignment tool, which is able to find Maximal Unique Matches (MUMs). Results show a reduction in search time and less usage of memory.

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

1 Introduction

Modern sequencing and computational technologies and advances in bioinformatics has made whole genome sequencing possible. One resulting challenge is the fast alignment of whole genomes. Dynamic programming is too slow for aligning two large genomes, hundreds of Mbp. One very successful approach to perform a whole genome alignment is based on identifying "maximal unique matches". This heuristic is based on the assumption that one expects substrings occurring in two similar genomes. Maximal unique matches (MUMs) are almost surely part of a good alignment of the two sequences and so the whole genome alignment problem can be reduced to aligning the sequence in the gaps between the MUMs.

Definition 1. *Assume we are given 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.*

The problem of searching maximal unique matches for a minimum length between a Reference genome and a Query genome has been identified in several applications, one of them is MUMmer [3]. MUMmer's algorithm can perform searches for Maximal Unique Matches (MUMs), although with a high use of main memory to store the Reference genome and a null use of multi-core architectures.

We designed and tested a data-level parallelism to use in multi-core architectures with two different data structures: Suffix Tree and Enhanced Suffix Array.

The rest of this paper is organized as follows. Section 2 discusses the related work. Section 3 defines some definitions used in this paper. Section 4 shows our implementation of parallelization of search for MUMs. Section 5 discusses the results of our parallelization. Section 6 concludes the paper.

2 Related work

Search for Maximal Unique Matches to do Whole Genome Alignment was proposed in [2]. There have been some previous work in the parallelization of search of matches in genomic data, like [11, 9, 7], 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 [8]. The parallelization of searching for MUMs with a full-text index data structure is a research field not covered very deep, there is only one approach in [4] but without access to source code to check their implementation and it is more focused with GPU and CPU hybrid architectures and Suffix Array. There are other implementations in [12, 11, 6, 10] but they search for Maximal Exact Matches with threads and not Maximal Unique Matches . We propose the use of OpenMP to search for MUMs in multi-core architectures with two full-text index data structures: Suffix Tree and Enhanced Suffix Array.

3 Preliminaries

Let's assume the Reference genome $R[0, \ldots, n-1]$ 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 or an Enhanced Suffix Array.

Definition 2. *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* $R[i \ldots n]$. *[5].*

Definition 3. *An enhanced suffix array consists of four arrays (suffix, longest common prefix (LCP), child and inverse suffix arrays) that have parts of the information saved in a suffix tree. [1].*

We also define a suffix link, which an important trick to achieve linear complexity to search for MUMs in both Suffix Tree and Enhanced Suffix Array.

Definition 4. *A suffix link is a pointer from string $\overline{aw}$ to substring $\overline{w}$.*

For reasons of space we recommend the reader to check [1] for full description of Enhanced Suffix Array. A search for a MUM between a Reference genome, $|R| = n$, and a Query genome, $|Q| = m$, can be done in a Suffix Tree in $O(m)$ steps and in an Enhanced Suffix Array in $O(m)$ steps. Once we have defined the resources used to index a Reference genome, we need to answer the question: Where are the MUMs of R and Q of some minimum length L? We show the algorithms used to answer this question using a Suffix Tree, see Algorithm 1, and a Enhanced Suffix Array, see Algorithm 2.

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
 1 begin
 2     ST ← buildST(R)
 3     foreach position i ∈ Q do
 4         length ← TraverseSuffixTree(Q[i],ST)
 5         if isLeafNode(Q[i]) then
               /* Leaf saves the position of a suffix in ST.          */
 6             if R[leaf − 1] ≠ Q[i − 1] and length ≥ L then
 7                 MUMcands ← saveMUMcand(R_leaf,i,length)
     /* Get unique MUMs from list of MUM-candidates.                  */
 8     MUMs ← cleanMUMcand(MUMcands)
 9 end
```

4 Implementation

The search for MUMs between a Reference and Query genome has an important feature which may help us to parallelize these algorithms: we can check more than one suffix of Query genome at the same time in Reference genome. Using a full-text index data structure to search for MUMs, we find MUM candidates while we stream the Query genome against the data structure. A MUM candidate is a MUM which is not unique in Query genome but only in Reference genome. To get a MUM we require to reach the end of Query genome to discard those MUM candidates which are not unique in both Reference and Query genome.

Algorithm 2: Search for MUMs in an Enhanced Suffix Array.

```
   input  : R, Q, L
   output: List of MUMs of length ≥ L, with start position in R and Q and length
 1 begin
 2     ESA ← buildESA
 3     while Q [i ] < end do
 4         traverseESA(Q, Q [i], interval, Q [i ].length())
 5         if interval.depth ≤ 1 then
 6             interval.depth = 0; interval.start = 0; interval.end = n-1; i ++;
               continue;
 7         if interval.size() == 1 and interval.depth ≥ L then
 8             if leftMaximal(Q, Q [i ], SA [interval.start]) then
 9                 saveMUMcand(SA [interval.start],i,interval.depth)
10         repeat
11             interval.depth = interval.depth-1
12             interval.start = ISA [SA [interval.start] + 1]
13             interval.end = ISA [SA [interval.end] + 1]; i ++; if interval.depth ==
               0 or suffixlink(inter) == false then
14                 interval.depth = 0; interval.start = 0; interval.end = n-1; break;
15         until interval.depth > 0 and interval.size()
16     MUMs ← cleanMUMcand(MUMcands)
17 end
```

Our approach, see Figure 1, is divided in three phases: Creation of Suffix Tree or Enhanced Suffix Array; Splitting query genome data (chunks) according to the number of available cores using 1 thread per core and parallel execution of the search for MUMs for every chunk, then every thread has its own list of MUM candidates; and Get MUMs from list of MUM candidates of all threads. A data-level parallelism to search for MUMs requires to split the Query genome in chunks and stream each chunk against the full-text index data structure. This search is independent of each other chunk. The final stage to filter MUMs must be performed with a full Query genome and with chunks of Query genome is more obvious too. This approach is suitable to be performed in multi-core architectures because chunks can be assigned to each core and search for MUMs in the full-text index data structure which is stored in main memory. So that a very good choice of parallelization involves a data partition technique.

Memory usage is improved, within this approach, with multiple reads to the full-text index data structure, instead of one read at a time with a serial execution.We take advantage of multi-core architectures with our approach. Since a SPMD paradigm is used to solve the MUM-problem, we must tackle the common issues which arise when using multi-core architectures: cache issues, memory bandwidth and multithreading.

The division of Query genome uses the paradigm of data-level parallelism which consists of a generation of chunks of a Query genome with a fixed size.

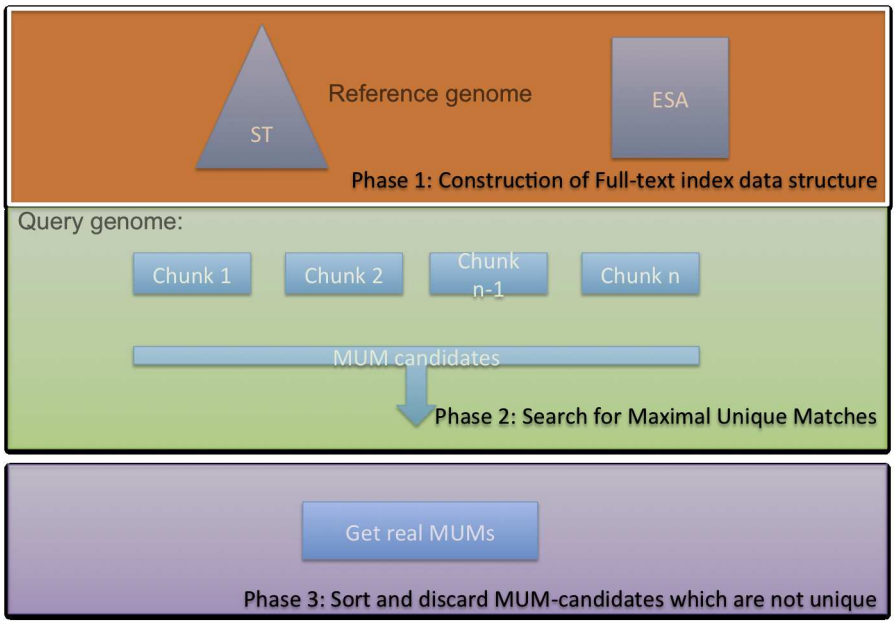

Fig. 1. Approach to search for MUMs in multi-core architectures with Suffix Tree or Enhanced Suffix Array.

The chunk size was computed with Query genome length divided by number of available threads. Chunk size is a performance factor because we need to have a balanced workload among threads.

Phase 1 is the construction of the data structure: Suffix Tree or Enhanced Suffix Array; this phase is executed in a serial way. Phase 2 requires the parallel execution of the Algorithm (1 or 2) to search for MUMs, the first step in Phase 2 is the split of Query genome in as many chunks as available threads. The split defines the start and end position in Query genome which is held in main memory and shared by all threads. The chunk size is fixed according to the number of threads. The parallel search for MUMs gets a list of MUM-candidates which are ordered in each thread by its position in Query genome. Phase 3 is performed after all threads have finished its execution and it merges the lists of MUM-candidates and it is ordered by position in Query genome (this step is needed because the merge process may have unordered lists); this unique list is checked to discard those MUM-candidates which are not unique in the Query genome. This check is performed in a serial way.

We evaluated two different full-text index data structures: a Suffix Tree and an Enhanced Suffix Array. We applied our approach, Figure 1, by using OpenMP in the phase 2: search for MUMs with a static OpenMP schedule.

5 Results

To verify that our approach, can have a better performance to search for MUMs of a Reference and Query genome, in Table 1, we verify that output of MUMmer and our approach have the same MUMs. Tests were carried out in the following node: 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 main goal of these tests was to

Table 1. List of Genomes used in experiments.

	1	2	3
Reference	4.64 [Mbp]	169 [Mbp]	1031 [Mbp]
Query	5.5 [Mbp]	167 [Mbp]	1357 [Mbp]

check the performance to search for MUMs in multi-core architectures by using OpenMP (threads). This performance involves the usage of Memory and CPU (execution time to search for MUMs).
Times were collected during the execution of the experiments: construction time of data structure and execution time to search for MUMs with the function omp_get_wtime of OpenMP.

5.1 Construction of Full-text index data structure

This paper compares the performance of two data structures to search for MUMs in multi-core architectures, two features for these structures are: construction time and memory footprint. Full-text indexes allow fast access to substrings of any length, but they have a great memory and construction cost. This cost is affected by the type and implementation of the full-text index data structure used. We measure the construction time for the set of genomes in Table1. Figure 2 shows that an ESA has a lower construction time for all the Reference genomes used. Since a full-text index data structure has to be build every time a search for MUMs is performed, a reduction in the construction phase in Figure 1 would allow us to improve the overall execution time. A future improvement would be to build the full-text index data structure in parallel or load from a file.

5.2 Search for MUMs in multi-core architectures

We focus in the experiment to search for MUMs of minimum length 20[bp]. Our goal is to get the best performance of a multi-core architecture while searching for MUMs with a Suffix Tree and Enhanced Suffix Array.

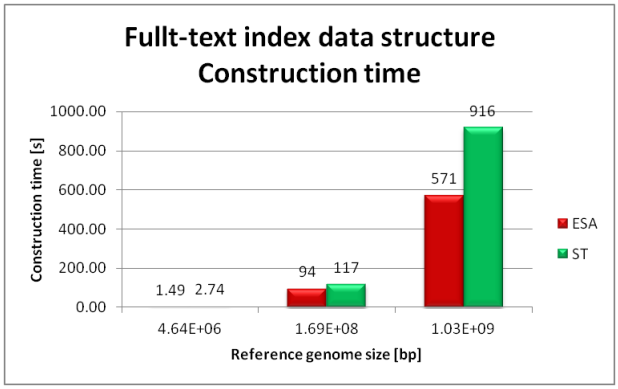

Fig. 2. Construction of Suffix Tree and Enhanced Suffix Array for several genomes sizes. Our approach stores the whole Reference genome indexed by a Suffix Tree (ST) or an Enhanced Suffix Array (ESA) in main memory.

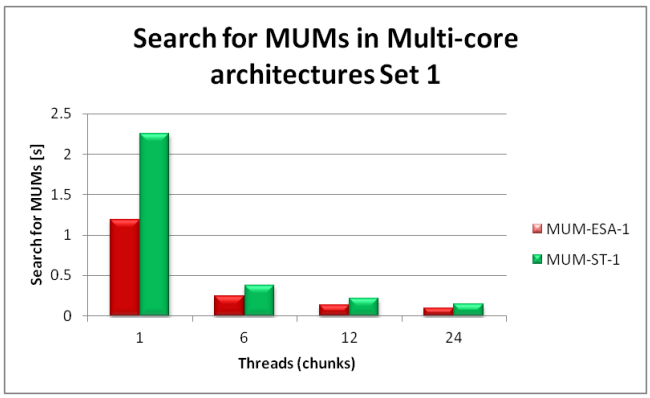

Fig. 3. Search for MUMs time for Ecolik12 and EcoliO157H7 in multi-core architectures with ST and ESA.

EcoliK12 vs. EcoliO157H7 The set 1 in Table 1 was used to search for MUMs in a ST and ESA. In Figure 3, we show that ESA has a better search time, see Figure 3. However, the space consumption shows that ESA has a better memory footprint, a reduction of 44% of memory space compare to Suffix Tree (159 MB). This first experiment shows that it is possible to search for MUMs with two different full-text index data structures with thea reduction in search time but with a lower memory consumption for ESA.

D.melanogaster vs. D.pseudoobscura The set 2 in Table 1 increases the genome size and we show in Figure 4 that ESA has a smaller search time. However, there is a problem with scalability between 12 and 24 threads. Our approach does not scale good with ESA. Although ST has a better speedup, we

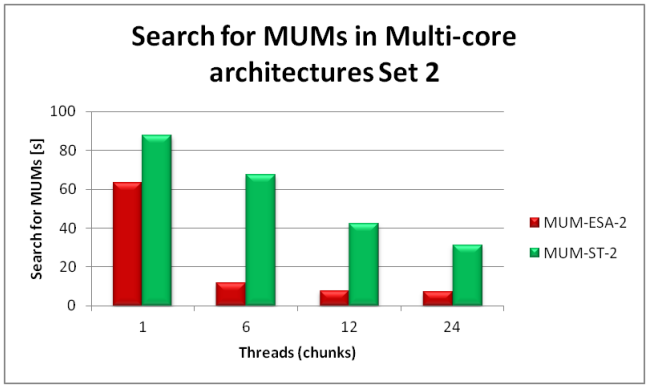

Fig. 4. Search for MUMs time for D.melanogaster and D.pseudoobscura in multi-core architectures with ST and ESA.

require to reduce the search time. On the other hand, the space consumption is almost constant in both data structures. However, again ESA has a better usage of memory. The reduction in memory footprint for ESA is near to the half of the ST.

Chicken vs. Zebrafish The set 3 in Table 1 have genomes of more than 1 [Gbp] and we show in Figure 5 that an execution with one thread has a better performance in search time in ESA over ST. In order to understand which full-

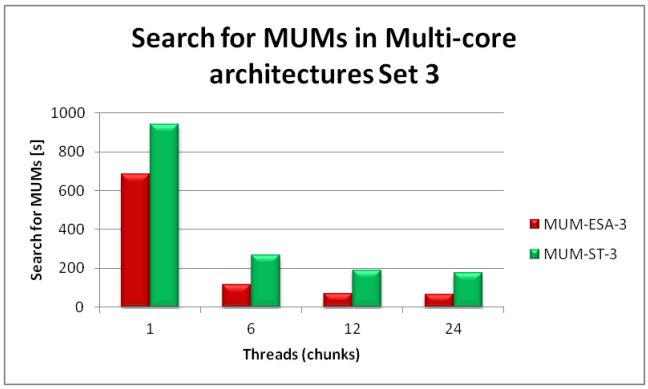

Fig. 5. Search for MUMs time for Chicken and Zebrafish in multi-core architectures with ST and ESA.

text index data structure (ST, ESA) use to search for MUMs we check the memory footprint in Figure 6. The ESA has a better use of memory with the same search time.

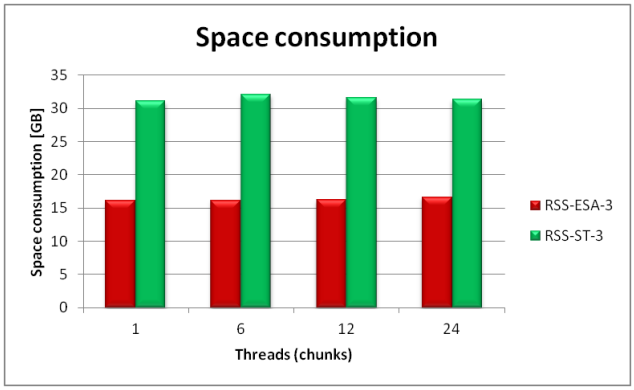

Fig. 6. RAM consumption to search for MUMs for Chicken and Zebrafish in multi-core architectures with ST and ESA.

6 Conclusions

This paper presents an evaluation of performance to search for MUMs between Reference genome and Query genome in multi-core architectures with OpenMP in a Suffix Tree and an Enhanced Suffix Array. The search for MUMs is performed in a Suffix Tree or an Enhanced Suffix Array and the list of MUMs can help in the process of Whole Genome Alignment. The results show that MUM-problem can be solved with a ST or ESA and with the use of a multi-core architecture we reduce the execution time. There is a reduction in memory usage with an Enhanced Suffix Array over a Suffix Tree. We have evaluated a different data structure which provides the same functionality of a Suffix Tree to search for MUMs, with less use of memory in multi-core architectures.

Improvements involve a better use of CPU when we have more than one thread per core. From the results obtained we conclude that an Enhanced Suffix Array is suitable to solve the MUM-problem. To improve the performance in CPU usage of our approach, the following proposals may be used: Prefetching: this may work because we know in advance what intervals to look for in ESA and Query genome; Data cache management: we may need to store the interval of ESA in some cache-level of processor. Since we need to get the interval of search by checking the first character of the suffix within the interval, we may use some SSE instructions to check more than one suffix at the same time.

7 Acknowledgments

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. M. I. Abouelhoda, S. Kurtz, and E. Ohlebusch. Replacing suffix trees with enhanced suffix arrays. *Journal of Discrete Algorithms*, 2(1):53–86, Mar. 2004.
2. A. L. Delcher, S. Kasif, R. D. Fleischmann, J. Peterson, O. White, and S. L. Salzberg. Alignment of whole genomes MUMMER. *Nucl Acids Res*, 27(11):2369, 1999.
3. A. L. Delcher, S. L. Salzberg, and 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):Unit 10.3, 2003.
4. G. Encarnac and N. Roma. Advantages and GPU Implementation of High-Performance Indexed DNA Search based on Suffix Arrays. *Architecture*, pages 49–55, 2011.
5. D. Gusfield. *Algorithms on strings, trees, and sequences: computer science and computational biology*. Cambridge University Press, 1997.
6. Z. Khan, J. S. Bloom, L. Kruglyak, and M. Singh. A practical algorithm for finding maximal exact matches in large sequence datasets using sparse suffix arrays. *Bioinformatics*, 25(13):1609–1616, July 2009.
7. 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.
8. X. Meng and V. Chaudhary. Exploiting multi-level parallelism for homology search using general purpose processors. *October*, 2005.
9. H. Mongelli. Efficient Two-Dimensional Parallel Pattern Matching with Scaling.
10. E. Ohlebusch, S. Gog, and A. Kügel. Computing matching statistics and maximal exact matches on compressed full-text indexes. In *SPIRE*, pages 347–358, 2010.
11. 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*, 12 Suppl 2:S7, Jan. 2011.
12. M. Vyverman, B. De Baets, V. Fack, and P. Dawyndt. essaMEM: finding maximal exact matches using enhanced sparse suffix arrays. *Bioinformatics (Oxford, England)*, pages 2–4, Feb. 2013.