BIOINFORMATICS ORIGINAL PAPER

Vol. 31 no. 4 2015, pages 509–514
doi:10.1093/bioinformatics/btu687

Genome analysis

Advance Access publication October 17, 2014

E-MEM: efficient computation of maximal exact matches for very large genomes

Nilesh Khiste and Lucian Ilie*

Department of Computer Science, University of Western Ontario, London, Ontario, N6A 5B7, Canada

Associate Editor: John Hancock

ABSTRACT

Motivation: Alignment of similar whole genomes is often performed using anchors given by the maximal exact matches (MEMs) between their sequences. In spite of significant amount of research on this problem, the computation of MEMs for large genomes remains a challenging problem. The leading current algorithms employ full text indexes, the sparse suffix array giving the best results. Still, their memory requirements are high, the parallelization is not very efficient, and they cannot handle very large genomes.

Results: We present a new algorithm, efficient computation of MEMs (E-MEM) that does not use full text indexes. Our algorithm uses much less space and is highly amenable to parallelization. It can compute all MEMs of minimum length 100 between the whole human and mouse genomes on a 12 core machine in 10 min and 2 GB of memory; the required memory can be as low as 600 MB. It can run efficiently genomes of any size. Extensive testing and comparison with currently best algorithms is provided.

Availability and implementation: The source code of E-MEM is freely available at: http://www.csd.uwo.ca/~ilie/E-MEM/

Contact: ilie@csd.uwo.ca

Supplementary information: Supplementary data are available at *Bioinformatics* online.

Received on July 21, 2014; revised on September 25, 2014; accepted on October 14, 2014

1 INTRODUCTION

Maximal exact matches (MEMs) are exact matches between two sequences that cannot be extended either way without introducing mismatches. MEM computation is a fundamental problem in stringology (Gusfield, 1997) and has important applications in sequence alignment. Closely related genomes are often aligned by using local similarities as anchors (Brudno *et al*., 2003; Deogun *et al*., 2004; Höhl *et al*., 2002; Kent, 2002; Menconi *et al*., 2013; Ohlebusch and Abouelhoda, 2006; Schwartz *et al*., 2000) and sufficiently long MEMs have been quite successfully used in this respect (Abouelhoda *et al*., 2004; Bray and Pachter, 2004; Choi *et al*., 2005; Delcher *et al*., 1999, 2002; Kurtz *et al*., 2004).

A theoretically optimal solution, in linear time and space, for the MEM computation problem is easily obtained using suffix trees (Weiner, 1973). However, suffix trees require large memory and practical implementations use highly engineered suffix trees (Kurtz, 1999). Suffix arrays were introduced by Manber and Myers (1993) as a space-efficient alternative to suffix trees and have replaced them in most applications. Enhanced suffix arrays were shown to solve all problems suffix trees could solve with the same theoretical complexity (Abouelhoda *et al*., 2004). Suffix arrays still use a significant amount of memory, especially with the additional tables required to match the complexity of suffix trees (Abouelhoda *et al*., 2004). When whole genomes are aligned, the memory required by the computation of all MEMs may become prohibitively high.

The large popularity of whole-genome alignment programs, most notably that of the MUMmer software (Kurtz *et al*., 2004), attracted a lot of attention to the MEM computation problem, with the purpose of enabling the alignment of larger genomes within reasonable amount of memory. For example, one of the most reliable such programs, Vmatch (Abouelhoda *et al*., 2004), uses enhanced suffix arrays and its memory usage is very high.

The idea of sparseness has been already used for suffix trees (Kärkkäinen and Ukkonen, 1996) and it has been successfully employed for suffix arrays by Khan *et al*. (2009) in their sparseMEM program. Their approach relies on indexing only every kth suffix of the given genome sequence (k is called *sparseness factor*) and is able to find MEMs faster and using less memory than previous approaches. It can serve as a drop-in replacement for the MUMmer3 software package (Kurtz *et al*., 2004). The approach of sparseMEM has been enhanced by Vyverman *et al*. (2013) with a sparse child array for large sparseness factors and implemented in their essaMEM software. Our tests show that essaMEM is currently the best program for MEM computation in large genomes.

Compressed indexes (Navarro and Mäkinen, 2007) have been used as well. Ohlebusch *et al*. (2010) developed backwardMEM that uses a backward search method over a compressed suffix array. Recently, Fernandes and Freitas (2013) employed in slaMEM a new sampled representation of the longest common prefix (LCP) array that works with the backward search method of the FM-Index (Ferragina and Manzini, 2000).

In spite of these advances, MEM computation remains a challenging problem for large genomes. The memory requirements of the current approaches remain high and very large genomes cannot be effectively handled. We present a new algorithm, E-MEM that targets large genome sequences. E-MEM does not use full text indexes. Instead, hash tables are efficiently used in combination with several ideas to speed up the search. Our algorithm uses much less space and is highly amenable to parallelization. For example, it can compute all MEMs of

*To whom correspondence should be addressed.

minimum length 100 between the whole human and mouse genomes on a 12-core machine in 10 min and 2 GB of memory; the required memory can be as low as 600 MB. It can run efficiently on genomes of any size.

Extensive testing and comparison with currently best algorithms is provided. We have used for comparison traditional datasets, such as whole human versus mouse genomes and whole human versus chimp genomes, but also introduced a new test where two species of wheat, *Triticum aestivum* and *Triticum durum* are used. It turns out that only E-MEM and Vmatch could handle these genomes. However, Vmatch requires 57 GB whereas E-MEM can use less than 1 GB while being also faster.

Our E-MEM software is implemented in C++ and OpenMP, is freely available, and can be used as a stand-alone program or as a drop-in replacement for the MUMmer3 software package (Kurtz *et al.*, 2004).

2 METHODS

Assume we have two genomes, the reference R, of length $|R|=n$, and the query Q, of length $|Q|=m$. The character of R at position i is $R[i]$ and the substring of R starting at position i and ending at j is denoted $R[i..j]$; we have also that $R=R[1..n]$. A k-mer is a string of length k. A *maximal exact match (MEM)* between R and Q is a match that cannot be extended on either side. Formally, it is a quadruple (b_r, e_r, b_q, e_q) such that $R[b_r..e_r]=Q[b_q..e_q]$ and $R[b_r-1]\neq Q[b_q-1]$, $R[e_r+1]\neq Q[e_q+1]$, if defined; e_r-b_r+1 is the *length* of the MEM.

The *MEM finding problem* is: given two sequences R and Q and an integer L, find all MEMs of length at least L between R and Q.

2.1 Hashing the reference

The starting idea of our approach is that, for any MEM (b_r, e_r, b_q, e_q) of length L or more between R and Q, there must be a k-mer in R starting at a position that is a multiple of $L-k+1$ that is completely contained in the MEM, that is, $b_r \le j(L-k+1) \le e_r-k+1$, for some $j\ge 1$. That means, it is sufficient to index all k-mers in the reference that start at positions that are multiples of $L-k+1$, which significantly reduces the required memory. However, all k-mers of the query genome have to be processed and this must be done very efficiently.

Each genome is encoded using two bits per nucleotide and then stored as an array of unsigned 64-bit integers, that is, as blocks of 32 nucleotides. We use k-mers of size 28 or less for technical reasons described below. Each k-mer starting at a position that is a multiple of $L-k+1$ is computed by a bit-and operation between a mask of $2k$ 1's and the appropriate elements of the array storing R. All k-mers are hashed using double hashing. In the hash table, each entry stores the value of the k-mer and a list of all positions where the k-mer occurs in R. All numbers are unsigned 64-bit integers, thus able to handle genomes of any size.

2.2 Searching the query

As mentioned above, all k-mers in Q need to be considered. However, they are not indexed but simply computed from Q and searched for in the hash table we built for R. Every time a k-mer of Q is found in the hash table, all positions at which it occurs in R are investigated for possible extension. If the extension has length L or more, then it is reported as a MEM. This way all MEMs are found.

The idea is now clear but a straightforward implementation is very slow. First, we need a fast way to compute the k-mers of Q. This is done by bit operations; we slide a 64-bit window across Q and each k-mer is computed by a bit-and between the window and a mask of $2k$ 1's. The window is then updated by shifting its content two bits. Every four shifts, another byte is added, for that reason the k-mer size has to be at most 28.

Second, we need to check each extension very fast. Recall that the genome sequences are stored in arrays of 64 bit blocks. The extension is performed in such a way that two 64-bit blocks are compared using only very few bit operations.

Third, some MEMs may be rediscovered many times, thus decreasing the speed. In order to avoid MEM rediscovery, we use a data structure, CurrMEMs, that stores the current MEMs. Given the current position j in Q, the current MEMs are those whose query part covers j, that is, all MEMs (b_r, e_r, b_q, e_q) such that $b_q \le j \le e_q$. Any new MEM found is added to CurrMEMs. When the current position j moves past e_q, the corresponding MEM is removed from CurrMEMs.

MEM rediscovery verification is done as follows. Any time a k-mer w at position j in Q is found in the hash table of R, then, for each position ℓ in the list of w in the hash table, we check first whether the k-mer pair $(j, j+k-1)$, $(\ell, \ell+k-1)$ lies inside an already discovered MEM. For each current MEM, (b_r, e_r, b_q, e_q), we need only check whether $j-b_q=\ell-b_r$. In practice, the number of current MEMs is never very high and thus the algorithm advances very fast through Q.

2.3 Unknown bases

Genome sequences may contain unknown bases, denoted by N. Our two-bit encoding does not allow storing N's so we replace them with random bases. Therefore, we need to eliminate any accidental matches between randomly replaced N's and real bases. Storing the positions of N's would be too space consuming, so we store them as blocks of N's, recording only their beginning and end. During the MEMs discovery process, we check whether any intersection with a block of N's occurred in R or Q. For Q this is very simple, since we know, for the current position in Q, where the closest blocks of N's on both sides are. For R, we need to verify, for each MEM found, that no N position has been included.

2.4 Reducing the memory further

So far the algorithm works very well, with little space and time. However, we need to store both sequences R and Q to enable comparison, which gives a lower bound for the required peak memory. In order to avoid this problem and reduce the memory further, we split each of R and Q sequences into $D\ge 1$ subsequences each; D is called *division factor*. However, we cannot simply cut them into D equal subsequences since this may prevent us from finding some MEMs straddling the borders. The D sequences must overlap. For R (similarly for Q), each subsequence has length $\ell=\frac{1}{D}(|R|+(D-1)L)$ and the overlap between consecutive subsequences is $L-1$. This way, any MEM must have at least L bases within a single subsequence and will be discovered.

Since we build hash tables on the reference, as explained above, we consider the subsequences of R in order, one at the time, build its hash table, then, for each subsequence of Q, find all MEMs shared by the two subsequences. The time increases since we process each subsequence of Q multiple times, but the space decreases very much with D.

A different problem that appears now is that we may discover only parts of the MEMs straddling borders. That is, some of the matches that reach the borders of the subsequences may not be maximal. This happens because we load into memory only one subsequence from each genome, and hence no match can be extended past the end of either subsequence. Matches reaching the ends of the subsequences are stored together in a data structure, MEMext, and processed at the end. If some of these are parts of the same, larger, MEM, then they are merged accordingly.

2.5 Parallelism

The above algorithm can be easily parallelized on any number of cores. Each subsequence of the query genome is divided equally into a number

of pieces equal to the number of processors. Each processor will then execute the algorithm we have described independently of the others. MEM's straddling these new borders may be discovered more than once and duplicates are removed at the end. Also, each thread has its own CurrMEMs data structure that it has to compute from scratch. Nevertheless, a speed up of up to six times faster is obtained on the 12-core machine that we used for testing.

2.6 Very large number of MEMs

For large genomes that are highly similar, the number of MEMs can be very large. For example, for human and chimp, the number of MEMs is well over a 100 million. Keeping all these MEMs in memory at the same time will reverse all the memory reductions we obtained so far, not to mention a significant amount of time necessary to sort them. To avoid these problems, we store the MEMs in files, sorted according to their starting positions in the query genome. At the end, each file is sorted, duplicates are removed, and MEMs are output in the right order.

2.7 Pseudocode

We bring together the ideas explained above into the pseudocode of the E-MEM algorithm. The main steps are identified. The implementation details have been explained above.

E-MEM(R,Q,L)

input: two sequences R and Q and a minimum MEM length L
output: all MEMs of length at least L between R and Q

```
1.  choose a division factor D
2.  split R into R_1, R_2, ..., R_D
3.  split Q into Q_1, Q_2, ..., Q_D
4.  for i from 1 to D do
5.    encode R_i
6.    ℓ ← L − k + 1
7.    while (ℓ ≤ |R_i| − k + 1) do
8.      hash the k-mer at position ℓ of R_i
9.      ℓ ← ℓ + L − k + 1
10.   for j from 1 to D do
11.     encode Q_j
12.     for ℓ from 1 to |Q_j| − k + 1 do
13.       get the k-mer q at position ℓ in Q_j
14.       search for k-mers r = q in the hash table of R_i
15.       for each occurrence of r with extension ≥ L do
16.         check CurrMEMs
17.         if ((q, r) discovers a new MEM) then
18.           if (MEM at ends of R_i or Q_j) then
19.             add to MEMext
20.           else
21.             add to file (by start position in query)
22.         update CurrMEMs
23. process MEMext to extend MEMs
24. move MEMs from MEMext to appropriate files
25. for each file with MEMs do
26.     sort MEMs by position in Q
27.     remove duplicates
28. output MEMs
```

3 RESULTS

We have compared E-MEM with several programs, including the top ones: essaMEM (Vyverman *et al.*, 2013), slaMEM (Fernandes and Freitas, 2013), sparseMEM (Khan *et al.*, 2009), and Vmatch (Abouelhoda *et al.*, 2004). We have also tested backwardMEM (Ohlebusch *et al.*, 2010) and MUMmer (Kurtz *et al.*, 2004) but they could not run any of our large tests. The genomes involved in the tests are given in Table 1, where we give also their length and number of sequences included in each fasta file. Download information for each genome sequence as well as for the source code of the programs tested is given in the Supplementary Material. Note that the wheat genome has ~17 GB, however the available sequence has only 4.3 GB.

We performed three tests. The first two are classical problems of large genome alignment: human versus mouse, human versus chimp. We have added two larger genomes of two wheat species: common wheat versus durum wheat. The results are presented in Tables 2–4, respectively. In each table, we include the time and

Table 1. Genomes used for testing

Datasets	Size (Mbp)	Sequences
Homo sapiens (Human)	3137	93
Mus musculus (Mouse)	2731	66
Pan troglodytes (Chimp)	3218	24 132
Triticum aestivum (Common wheat)	4391	731 921
Triticum durum (Durum wheat)	3229	5 671 204

Table 2. *Homo sapiens* versus *Mus musculus*; MEMs of minimum length 100

Program		Time (s)		Memory (MB)	
		serial	12 cores	serial	12 cores
essaMEM	$K = 1$	—	—	—	—
	$K = 2$	4076	3107	19 065	19 697
	$K = 4$	8291	3174	11 264	11 896
	$K = 8$	9243	2069	7394	8282
	$K = 16$	4437	1385	5468	6394
	$K = 32$	6245	3520	4508	5396
	$K = 64$	9044	5119	4029	4917
slaMEM		62 099	—	3480	—
sparseMEM	$K = 1$	—	—	—	—
	$K = 2$	3845	2947	20 182	20 750
	$K = 4$	19 797	6833	12 426	13 250
	$K = 8$	58 325	10 217	8548	9327
	$K = 16$	50 703	9169	6609	7497
	$K = 32$	46 492	8853	5640	6528
	$K = 64$	41 396	9398	5155	6043
Vmatch		7370	—	39 377	—
E-MEM	$D = 1$	1792	324	3979	4705
	$D = 2$	2241	392	2138	2864
	$D = 3$	2864	505	1513	2239
	$D = 4$	3266	596	1211	1937
	$D = 5$	3900	699	1009	1735
	$D = 6$	4327	799	884	1610
	$D = 7$	4841	903	786	1512
	$D = 8$	5340	1048	722	1448
	$D = 9$	5855	1097	669	1395
	$D = 10$	6296	1209	623	1349

Note: A dash means the program could not run that test, that is, in serial mode, the output was incorrect or empty, whereas the parallel mode was not supported.

Table 3. *Homo sapiens* versus *Pan troglodytes*; MEMs of minimum length 100

Program		Time (s)		Memory (MB)	
		serial	12 cores	serial	12 cores
essaMEM	$K = 1$	—	—	—	—
	$K = 2$	3452	2642	19 057	19 633
	$K = 4$	25 328	7887	11 384	12 088
	$K = 8$	24 220	4422	7386	8282
	$K = 16$	10 404	1850	5588	6356
	$K = 32$	12 381	3661	4628	5396
	$K = 64$	16 048	5275	4149	4917
sparseMEM	$K = 1$	—	—	—	—
	$K = 2$	10 169	2511	20 174	20 814
	$K = 4$	20 346	4898	12 606	13 250
	$K = 8$	57 390	9049	8540	9436
	$K = 16$	56 946	9413	6601	7535
	$K = 32$	58 210	9267	5632	6528
	$K = 64$	49 807	10 083	5147	6043
Vmatch		7696	—	39 725	—
E-MEM	$D = 1$	7611	1992	4123	4849
	$D = 2$	6780	1933	2210	2937
	$D = 3$	6838	2022	1562	2288
	$D = 4$	7056	2136	1247	1973
	$D = 5$	7772	2250	1039	1765
	$D = 6$	8077	2408	908	1634
	$D = 7$	8868	2487	807	1533
	$D = 8$	9018	2630	741	1467
	$D = 9$	9436	2707	686	1412
	$D = 10$	10 014	2795	638	1364

Note: A dash means the program could not run that test, that is, in serial mode, the output was incorrect or empty, whereas the parallel mode was not supported.

Table 4. *Triticum aestivum* versus *Triticum durum*; MEMs of minimum length 100

Program		Time (s)		Memory (MB)	
		serial	12 cores	serial	12 cores
E-MEM	$D = 1$	1073	352	5847	6573
	$D = 2$	1610	462	3105	3831
	$D = 3$	2388	611	2216	2942
	$D = 4$	2830	724	1700	2426
	$D = 5$	3472	849	1426	2152
	$D = 6$	4095	947	1237	1963
	$D = 7$	4690	1069	1108	1834
	$D = 8$	5175	1186	1008	1734
	$D = 9$	5664	1314	900	1626
	$D = 10$	6429	1413	926	1652
Vmatch		6932	—	56 987	—

Note: Only E-MEM and Vmatch could run this test.

memory required for each of the five programs tested, in both serial and parallel mode on 12 processors. The times given include all preprocessing and index construction. The minimum MEM length is 100 in all cases, as usually tested in the literature for these genome sizes. Since MUMmer3 (Kurtz *et al.*, 2004) computed MEMs of minimum length 300, we provide the results for this case in the Supplementary Material. We have verified that all programs produced the same MEMs in all tests.

Sparseness factors $K = 1, 2, 4, 8, 16, 32, 64$ have been tested for essaMEM and sparseMEM and division factors $D = 1, 2, \ldots, 10$ for E-MEM.

We have run all programs on the same DELL PowerEdge R620 computer with 12 cores Intel Xeon at 2.0 GHz and 256GB of RAM, running Linux Red Hat, CentOS 6.3. The latest versions of all programs were run; details are given in the Supplementary Material.

Since we are trying to reduce both time and memory, a trade off is obtained and the results are better seen when time is plotted against memory. We show these plots in Figures 1, 2 and 3. For the first two tests, we give two plots for clarity, one for the serial mode, the other for the parallel.

3.1 Human versus mouse

In our first test, the whole human and mouse genomes have 537 491 MEMs of minimum length 100. E-MEM produces much better results than the other programs; see Table 2.

In serial mode, the smallest memory obtained by the other programs was that of slaMEM, with 3.5 GB. However, slaMEM took 17 h to complete the test. Therefore, the best performance comes from essaMEM, which can use as low as 4 GB of memory, but complete the test in 2.5 h; essaMEM can complete the test in little over one hour but at the cost of increasing the memory to 19 GB. E-MEM can run within as little as 623 MB and complete in less than two hours or in half an hour and 4 GB of memory.

E-MEM has the best speed up in parallel, between 5 and 6 times on our 12-core machine. The speed up of essaMEM varies between 1.3 and 4.5 times. We note that sparseMEM has speed up comparable to that of E-MEM but the running time in serial mode is one order of magnitude higher. The best result from the competing programs comes again from essaMEM, that has been consistently outperforming the other ones: 23 min and 6 GB or 1 h and 5.4 GB. E-MEM can complete the test in 6 min and 4.7 GB or 10 minutes and less than 2 GB.

The time and space are plotted against each other in Figure 1. Programs closer to origin are faster and require less memory. The left plot is for serial mode and the right for parallel. The area is very large in the serial plot, due to the large memory requirements of Vmatch and long running times required by slaMEM and sparseMEM. Since the two programs do not run in parallel, the plot giving the time/space values for the parallel testing gives a more clear picture.

Only essaMEM, sparseMEM, and E-MEM could run in parallel. The memory required for the same sparseness factor by essaMEM is slightly smaller than that of sparseMEM while essaMEM is much faster than sparseMEM. All three programs trade time for space but the results of E-MEM are much closer to origin than the rest; see Figure 1.

3.2 Human versus chimp

For our second test, human versus chimp, there are 132 368 058 MEMs of minimum length 100 to be found. This slows down

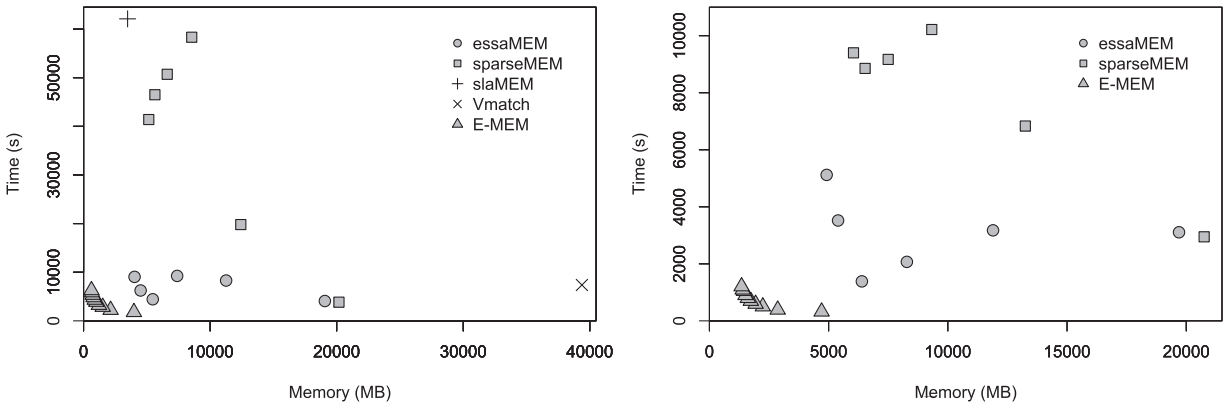

Fig. 1. *Homo sapiens* versus *Mus musculus*; MEMs of minimum length 100. The left plot is for serial mode, the right for parallel. Note the different scale of the plots

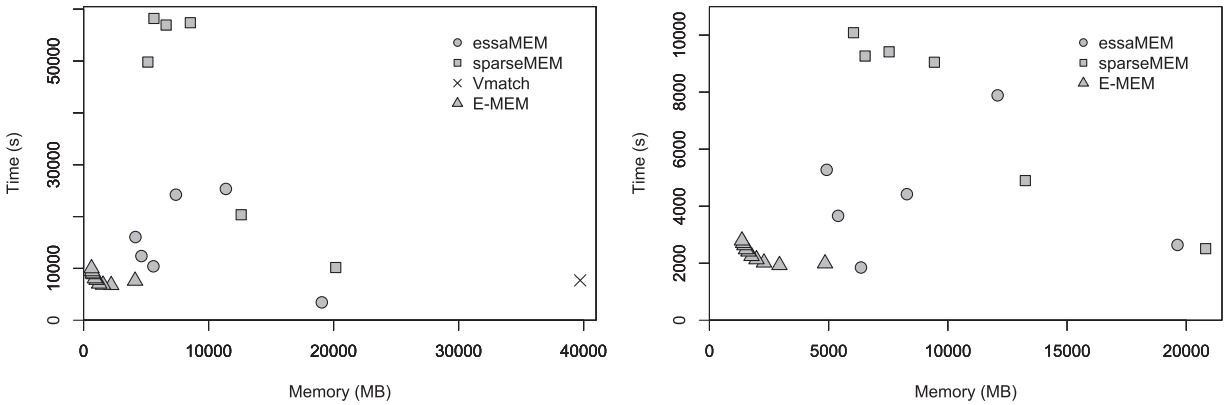

Fig. 2. *Homo sapiens* versus *Pan troglodytes*; MEMs of minimum length 100. The left plot is for serial mode, the right for parallel. Note the different scale of the plots

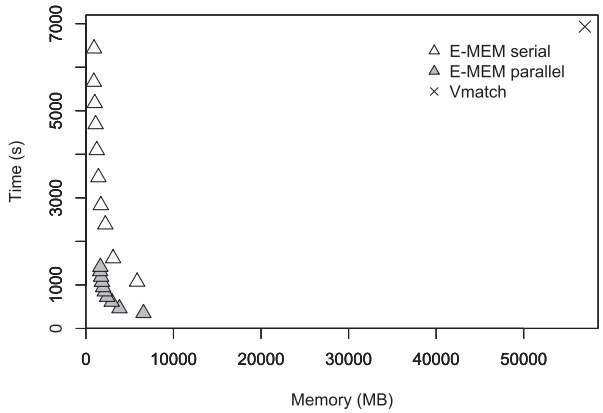

Fig. 3. *Triticum aestivum* versus *Triticum durum*; MEMs of minimum length 100. Both serial and parallel results of E-MEM are plotted

slightly the E-MEM program since all these MEMs need to be post processed. (The very large number of MEMs is also the reason why the program is slightly slower for $D = 1$ compared with $D = 2$.) The details are given in Table 3. The whole picture, given in Figure 2 is similar with the one for the first test, human versus mouse (Fig. 1) except that the difference between the running times is not as large.

3.3 Two wheat species

In our last test, we used two wheat genomes that are larger than the mammalian ones used in the previous two tests. Only E-MEM and Vmatch could run this test and the results are presented in Table 4 and plotted in Figure 3. Note that this figure is slightly different than Figures 1 and 2 in that both serial and parallel results are presented in the same plot. E-MEM could run this test efficiently; in serial mode it can use less than 1GB of memory and in parallel it requires only 16 min and 2GB of memory.

4 CONCLUSION

E-MEM provides a practical solution for finding MEMs between arbitrarily large genomes. It uses much less memory than the currently available programs. It is freely available and it can be used as a stand-alone program or as a drop-in replacement for the MUMmer3 software package (Kurtz *et al*., 2004).

MEMs are good anchors for closely related genomes. Otherwise, approximate matches are more suitable. The approach of E-MEM can be generalized to work with spaced seeds (Ma *et al*., 2002) in

order to search efficiently for approximate matches. Highly sensitive multiple spaced seeds (Li *et al*., 2004) of large weight are necessary and they can be designed using the approach of Ilie and Ilie (2007) by the SpEED program (Ilie *et al*., 2011). The exact matching procedure of index-based algorithms is not well suited for finding approximate matchings.

ACKNOWLEDGEMENTS

Performance evaluation has been performed using the facilities of the Shared Hierarchical Academic Research Computing Network (SHARCNET: www.sharcnet.ca) and Compute/Calcul Canada.

Funding: This work was supported by the Natural Sciences and Engineering Research Council of Canada (NSERC) Discovery Grant R3143A01 (L.I.).

Author contributions: L.I. designed the E-MEM algorithm and the efficient implementation and wrote the manuscript. N.K. implemented E-MEM, contributed to improving the design, installed the competing programs, downloaded the genome sequences, and performed all tests.

Conflict of interest: none declared.

REFERENCES

Abouelhoda,M.I. *et al*. (2004) Replacing suffix trees with enhanced suffix arrays. *J. Discrete Algorithms*, **2**, 53–86.

Bray,N. and Pachter,L. (2004) MAVID: constrained ancestral alignment of multiple sequences. *Genome Res*., **14**, 693–699.

Brudno,M. *et al*. (2003) Fast and sensitive multiple alignment of large genomic sequences. *BMC Bioinformatics*, **4**, 66.

Choi,J.-H. *et al*. (2005) GAME: a simple and efficient whole genome alignment method using maximal exact match filtering. *Comput. Biol. Chem*, **29**, 244–253.

Delcher,A.L. *et al*. (1999) Alignment of whole genomes. *Nucleic Acids Res*, **27**, 2369–2376.

Delcher,A.L. *et al*. (2002) Fast algorithms for large-scale genome alignment and comparison. *Nucleic Acids Res*., **30**, 2478–2483.

Deogun,J.S. *et al*. (2004) Emagen: An efficient approach to multiple whole genome alignment. In: *Proceedings of the second conference on Asia-Pacific bioinformatics*. Vol. 29, Australian Computer Society, Inc, pp. 113–122.

Fernandes,F. and Freitas,A.T. (2014) slaMEM: efficient retrieval of maximal exact matches using a sampled LCP array. *Bioinformatics*, **30**, 464–471.

Ferragina,P. and Manzini,G. (2000) Opportunistic data structures with applications. In: *Foundations of Computer Science, 2000. Proceedings. 41st Annual Symposium on IEEE*. Redondo Beach, CA. pp. 390–398.

Gusfield,D. (1997) *Algorithms on strings, trees and sequences: computer science and computational biology*. Cambridge University Press, New York.

Höhl,M. *et al*. (2002) Efficient multiple genome alignment. *Bioinformatics*, **18** (**Suppl. 1**), S312–S320.

Ilie,L. and Ilie,S. (2007) Multiple spaced seeds for homology search. *Bioinformatics*, **23**, 2969–2977.

Ilie,L. *et al*. (2011) SpEED: fast computation of sensitive spaced seeds. *Bioinformatics*, **27**, 2433–2434.

Kärkkäinen,J. and Ukkonen,E. (1996) Sparse suffix trees. In: *Computing and Combinatorics*. Springer, Berlin Heidelberg, pp. 219–230.

Kent,W.J. (2002) Blatthe blast-like alignment tool. *Genome Res*., **12**, 656–664.

Khan,Z. *et al*. (2009) A practical algorithm for finding maximal exact matches in large sequence datasets using sparse suffix arrays. *Bioinformatics*, **25**, 1609–1616.

Kurtz,S. (1999) Reducing the space requirement of suffix trees. *Softw. Practice Exp*., **29**, 1149–71.

Kurtz,S. *et al*. (2004) Versatile and open software for comparing large genomes. *Genome Biol*., **5**, R12.

Li,M. *et al*. (2004) PatternHunter II: Highly sensitive and fast homology search. *J. Bioinformatics Comput. Biol*., **2**, 417–439.

Ma,B. *et al*. (2002) PatternHunter: faster and more sensitive homology search. *Bioinformatics*, **18**, 440–445.

Manber,U. and Myers,G. (1993) Suffix arrays: a new method for on-line string searches. *Siam J. Comput*., **22**, 935–948.

Menconi,G. *et al*. (2013) Mobilomics in saccharomyces cerevisiae strains. *BMC Bioinformatics*, **14**, 102.

Navarro,G. and Mäkinen,V. (2007) Compressed full-text indexes. *ACM Computing Surveys (CSUR)*, **39**, 2.

Ohlebusch,E. and Abouelhoda,M.I. (2006) Chaining algorithms and applications in comparative genomics. In: *Handbook of Computational Molecular Biology*. Chapman & Hall/CRC, Boca Raton FL.

Ohlebusch,E. *et al*. (2010) Computing matching statistics and maximal exact matches on compressed full-text indexes. In: *String Processing and Information Retrieval*. Springer, pp. 347–358.

Schwartz,S. *et al*. (2000) Pipmakera web server for aligning two genomic dna sequences. *Genome Res*., **10**, 577–586.

Vyverman,M. *et al*. (2013) essaMEM: finding maximal exact matches using enhanced sparse suffix arrays. *Bioinformatics*, **29**, 802–804.

Weiner,P. (1973) Linear pattern matching algorithms. In: *Switching and Automata Theory, 1973. SWAT'08. IEEE Conference Record of 14th Annual Symposium on IEEE*. pp. 1–11.