BIOINFORMATICS ORIGINAL PAPER

Vol. 26 no. 19 2010, pages 2368–2374
doi:10.1093/bioinformatics/btq433

Genome analysis

Advance Access publication July 27, 2010

RepMaestro: scalable repeat detection on disk-based genome sequences

Nikolas Askitis* and Ranjan Sinha*

Department of Computer Science and Software Engineering, University of Melbourne, Australia

Associate Editor: Dmitrij Frishman

ABSTRACT

Motivation: We investigate the problem of exact repeat detection on large genomic sequences. Most existing approaches based on suffix trees and suffix arrays (SAs) are limited either to small sequences or those that are memory resident. We introduce RepMaestro, a software that adapts existing in-memory-enhanced SA algorithms to enable them to scale efficiently to large sequences that are disk resident. Supermaximal repeats, maximal unique matches (MuMs) and pairwise branching tandem repeats have been used to demonstrate the practicality of our approach; the first such study to use an enhanced SA to detect these repeats in large genome sequences.

Results: The detection of supermaximal repeats was observed to be up to two times faster than Vmatch, but more importantly, was shown to scale efficiently to large genome sequences that Vmatch could not process due to memory constraints (4 GB). Similar results were observed for the detection of MuMs, with RepMaestro shown to scale well and also perform up to six times faster than Vmatch. For tandem repeats, RepMaestro was found to be slower but could nonetheless scale to large disk-resident sequences. These results are a significant advance in the quest of scalable repeat detection.

Software availability: RepMaestro is available at http://www.naskitis.com

Contact: askitisn@gmail.com; sinhar@unimelb.edu.au

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

Received on March 16, 2010; revised on July 19, 2010; accepted on July 23, 2010

1 INTRODUCTION

The analysis of whole genome sequences is of critical importance in bioinformatics. A significant problem in such sequence analysis is the presence of repetitive structures, the analysis of which plays a crucial role in the study and comparison of complete genomes (Abouelhoda *et al.*, 2006). These repetitive regions occur in biological strings such as in DNA, RNA and protein, and have been shown to play a functional and evolutionary role (Gusfield, 1997; Jurka and Batzer, 1996; McConkey, 1993; Smith, 1998; Watson *et al.*, 1987). The vast majority of repeats occur in larger organisms such as plants and animals belonging to the eukaryote kingdom and comparatively fewer repeats occur in prokaryote organisms such as bacteria. Significantly, families of repeats account for almost 50% of the human genome (Gusfield, 1997) and comprise nearly 11% of the mustard weed genome and 7% of the worm genome (Abouelhoda *et al.*, 2004, 2006).

Surprisingly, only a very small fraction (2–3%) of the human genome sequence contains the protein-coding regions. A large fraction of the genome contain repetitive elements, the functioning of which are yet to be known. For example, long and short interspersed elements account for 21% and 13% of the genome, respectively, while repetitive elements such as micro- and mini-satellites could amount to 15% of the genome. This indicates that repetitive sequences exist in abundance in the genome and several of them have been shown to result in abnormalities and diseases.

During the analysis of genomes, the basic tasks include locating and characterizing repetitive sequences in the genome and finding similar repeats between two genomes. Broadly, there are two types of repeats: exact and approximate. While exact repeats are a smaller fraction of the repeats having biological interest, they form the ‘core blocks of approximate repeats’ (Abouelhoda *et al.*, 2004, 2006; Leung *et al.*, 1991). Consequently, we restrict our discussion to the detection of exact repeats and do not consider approximate or inexact repeats. Nonetheless, adapting our software to detect approximate repeats presents a promising avenue for future research.

Repeats play a crucial role in furthering our understanding, but the rate at which new genome sequences are being uncovered has been increasing rapidly. The genome sequencing project has been generating sequence data at an exponential rate (Benson *et al.*, 2007) and the size of GenBank has been doubling approximately every 18 months. Hence, with the growth of genomic collections as well as increasing query lengths (Williams and Zobel, 2002), it is crucial that the efficiency of sequence analysis remains practical. An example of the impact of this exponential data growth is with the popular local alignment search tool BLAST (Altschul *et al.*, 1997), which has been getting slower by $\sim$64% each year (Cameron *et al.*, 2004) inspite of hardware improvements. Thus, we continue to need not only faster and more efficient approaches for the detection of repeats but also ones that are scalable.

Two key data structures used for such tasks are the suffix tree and the suffix array (SA). These are helpful for the analysis of sequences that are large but otherwise static. There exists disk-based suffix tree (Phoophakdee and Zaki, 2007) and SA construction (Dementiev *et al.*, 2008) algorithms that can create a human genome scale structure in under 5 h. However, these data structures are seldom considered for large-scale genome analysis, where these indexes primarily reside on disk. This is primarily due to the poor locality of access inherent in these structures. Our intention is to show that the enhanced SA -based structure (Abouelhoda *et al.*, 2004, 2006;

*To whom correspondence should be addressed.

Moffat *et al.*, 2009; Sinha *et al.*, 2008) scales very well to the task of repeat detection of large genomes. Furthermore, it has been shown that a SA equipped with additional information can be used to solve all problems for which a suffix tree has been traditionally used (Abouelhoda *et al.*, 2004), thus our techniques could be applied to enhance the scalability of a wide array of (bottom–up traversal) problems.

Fundamental problems in bioinformatics (Gusfield, 1997; Smyth, 2003) include the detection of supermaximal repeats, maximal repeats, maximal unique matches (MuMs), maximal multiple exact matches, tandem repeats, exact pattern matching and maximal exact matches; each of which has numerous applications in comparative genomics, homology searching and finding repeat sequences. For example, tandem repeats are the markers of choice in genomic research and find applications in determining parentage, genealogical DNA tests, genotyping pathogens, morphological evolution and in human identification in forensic cases. Furthermore, variations in tandem repeats are an indication of diseases.

Existing genomic search tools (Chain *et al.*, 2003) can be divided into two major groups: exhaustive and index based. The exhaustive search tools like BLAST incorporate several heuristics to compare the query to each sequence in the database, but there are growing concerns regarding its scalability. The index-based tools such as CAFE (Williams and Zobel, 2002), Vmatch (Kurtz, 2008) and MUMmer (Kurtz *et al.*, 2004) preprocess the collection to create an index. CAFE adapts the inverted index structure for fast and scalable homology search but it remains unclear if it could be adapted to solve the other core bioinformatics problems. Vmatch, a state-of-the-art pattern recognition software, is based on an enhanced SA structure (Abouelhoda *et al.*, 2004) and solves several of the core bioinformatics problems but it is restricted to sequences and indexes being memory-resident. REPuter (Kurtz *et al.*, 2001), a precursor to Vmatch, is based on the suffix tree representation and consequently has both speed and scalability concerns. Lian *et al.* (2008) introduced a technique that builds an index on top of an external suffix tree to detect supermaximal repeats on large genome sequences. With a minimum repeat length of 10, their implementation is 2× faster than Vmatch, but its large index size is prohibitive. Thus, a primary drawback in existing systems is that the sequences and associated structures are required to be memory-resident for efficiency. The Tallymer software (Kurtz *et al.*, 2008) addresses this concern for the task of k-mer counting and for indexing large sequence sets, by utilizing a disk-resident-enhanced SA. Tallymer offers linear processing time and space consumption, but it is unclear as to whether the software can operate with a bound on memory usage (i.e. whether or not it is scalable), as a consequence of its linear space requirement. Moreover, the authors do not investigate or propose software solutions for finding repeats such as MuMs, in a scalable manner. Consequently, there exists inherent constraints in existing software tools that we seek to improve upon.

We have, therefore, developed a pattern recognition software called RepMaestro that offers scalable and memory-conscious algorithms for finding supermaximal repeats, MuMs and tandem repeats in large genome sequences on disk. Our approach, similar to the workings of Vmatch, employs an enhanced SA that is engineered to be accessed and maintained from disk. We demonstrate through careful experimental evaluations that our approach is highly scalable and in many cases, faster than Vmatch and MUMmer.

2 METHODS AND ALGORITHMS

2.1 Enhanced SA/SLB

Since we must process a SA and its corresponding longest common prefix (LCP) array and Burrow–Wheeler transform (BWT) array, we employ an enhanced SA (Abouelhoda *et al.*, 2006; Homann *et al.*, 2009) or SLB (abbreviation for **S**A, **L**CP and **B**WT) array to improve access locality of large disk-resident genome sequences. To create an SLB array, we copy each SA value (4 bytes), its corresponding LCP (4 bytes) and BWT value (1 byte) into a 9-byte SLB cell (assuming a 32-bit address space; otherwise its 17 bytes). Thus, SLB[i] represents SA[i],LCP[i],BWT[i], where $i=0<i<n$ (n being the number of entries).

2.2 Supermaximal repeats

A *repeat* is a substring occurring in at least two different positions in any given sequence. A *maximal repeat* satisfies the repeat conditions but cannot be extended either to the left or right to make a longer repeat. Thus, a maximal repeated pair in a sequence S is a pair of identical substrings S_1 and S_2 in S such that the character to the immediate left or right of S_1 is different from the character to the immediate left or right of S_2, respectively. That is, extending S_1 and S_2 in either direction would destroy the equality of the two strings. A *supermaximal repeat* is also a maximal repeat except that it does not occur as a substring of any other maximal repeat.

Our interest is to develop an algorithm to detect supermaximal repeats efficiently on large disk-based genome sequences. We direct the reader to Abouelhoda *et al.* (2006) for further information regarding maximal and supermaximal repeats. According to Abouelhoda *et al.* (2006), the detection of supermaximal repeats requires finding all L-intervals and *local maximas*. To find these, we need to only process the LCP array. An L-interval represents a region i to j (where $0 \leq i < j \leq n$) in the LCP array that shares a LCP length of L. A more formal definition is provided in Supplementary Material.

A local maxima is defined as an L-interval i to j, where all LCP values between positions $(i+1)$ and j are equal to L. Specifically, lcp[k]$=L$ for all $(i+1) \leq k \leq j$. A local maxima is recognized as a supermaximal repeat if and only if the BWT values between positions i and j are pairwise distinct. Once satisfied, positions i to j in the SA are supermaximal repeats.

We need an efficient method of finding supermaximal repeats on a genome sequence (an SLB) that is stored on disk. Abouelhoda *et al.* (2006) provide a simple linear-time algorithm that involves finding all local maxima in a sequence S, to compute supermaximal repeats. We provide the algorithm in Supplementary Material. This is an elegant solution but one that may not be particularly beneficial to a practitioner that wishes to implement it on disk. Specifically, how do we find all local maximas on disk efficiently? Moreover, why do we need to find all local maximas before we proceed to find supermaximal repeats? We, therefore, propose an algorithm that is engineered to eliminate random access to disk and also make efficient use of CPU cache, by finding all supermaximal repeats through a single scan of an SLB array.

Given an SLB array, we scan it once from left-to-right, which is both I/O optimal and CPU-cache efficient. While we are scanning, our goal is to detect specific LCP and BWT patterns that signify the presence of a supermaximal repeat. We look for *hat-like* patterns in LCP values: the second LCP must be larger than the first, thereupon the LCP values must be equal to the second LCP followed by a fall (or equivalently, when the SLB is exhausted). Once we have established the existence of this hat-like pattern, the BWT values are checked to confirm that they are distinct (this is accomplished on-the-fly). The requirements are: a pre-built SLB of S and optionally its text file, which is needed to print out the supermaximal repeats. We summarize our algorithm below.

(1) Copy the current contents from SLB[i] (init. $i=0$) into an empty (reuseable) buffer and maintain a counter C (init. 1) of its entries. Clear a (reusable) BWT bit-map that has a 1-to-1 mapping with the

ASCII-7 table. Set the bit that maps to the current BWT (from SLB[i]). Label the current LCP value as `previous LCP`.

(2) Fetch the next entry from the SLB ($i=i+1$ then SLB[i]). Its LCP value represents the `current LCP`. Set the bit that maps to the current BWT. If the corresponding position is already set, raise a flag to indicate. Increment C by 1 if and only if the previous step was not Step 1.

(3) If `current LCP` > `previous LCP` and $C=1$, append the contents of SLB[i] to the buffer. Update the `previous LCP` label then go back to Step 2 unless the BWT flag was raised in the previous step, in which case, go to Step 1.

(4) If `current LCP` = `previous LCP` and $C=1$, then this is not an L-interval. Go to Step 1.

(5) If `current LCP` > `previous LCP` and $C\geq 2$, then it can not be a local maxima. Go to Step 1.

(6) If `current LCP` = `previous LCP` and $C\geq 2$, append the contents of SLB[i] to the buffer. Update the `previous LCP` label then go back to Step 2 unless the BWT flag was raised in the previous step, in which case, go to Step 1.

(7) If `current LCP` < `previous LCP` (or SLB is exhausted), iterate through each entry in the buffer and print out the corresponding SA value that represents a location of a supermaximal repeat. Finish by printing the maximum LCP encountered during the iteration—the length of the repeats. Go back to Step 1 and repeat until SLB is exhausted.

2.3 MuMs

MuMs are important in computational biology for large-scale genome comparisons and sequence alignment problems. A MuM is defined as: given two sequences $S1$ and $S2$, a MuM is a sequence that occurs exactly once in $S1$ and once in $S2$, and is not contained in any longer such sequence. Global alignment algorithms (Hon and Sadakane, 2002) take $O(mn)$ time, given input sequences of length m (for $S1$) and n (for $S2$), respectively. Unfortunately, large values of m and n above 1 million characters makes it impractical. To scale to larger sequences, Delcher *et al.* (2002) introduced a heuristic that if a long sequence occurs exactly once in the two strings, then this substring has a high chance of occurring in the global alignment. This reduced the problem to finding the MuM, aligning the MuMs and then aligning the local gaps between successive MuMs. Significantly, these phases require only $O(n)$ time and is faster than conventional algorithms.

Consequently, several tools exist that seek to align two similar genomes. During this alignment, the first task is to find the MuM of the two genome sequences. While there exists approaches using suffix trees that can compute MuMs in $O(n)$ time and space, where $n=|S1\#S2|$ and # is a symbol not occurring in either of $S1$ and $S2$, the SA approach offers greater space efficiency and locality of access. Hence, we have used the ESA approach (Abouelhoda *et al.*, 2004; Sinha *et al.*, 2008) to engineer an algorithm that can scale to large disk-resident genome sequences. We provide an existing linear time MuM algorithm (Abouelhoda *et al.*, 2006) in Supplementary Material, which we adapted for use on disk.

Similar to the supermaximal repeat algorithm described earlier, the MuM algorithm requires us to find all local maximas prior to processing, which is not an efficient task on disk. We, therefore, present our modified MuM algorithm below that can find MuMs in $S1\#S2$ with single scan of its SLB array. Our algorithm can not compute matches in the reverse strand (reverse complemented matches), though it can be readily adapted for this task at the expense of consuming more disk space for storing the necessary indexes (we provide further details in Suppleменtary Material).

During the scan, we look for local maximas with *spike-like* patterns in LCP values. First, we find a local maxima (of only two entries) by tracking a consecutive rise-fall pattern of LCP values. Once identified, the corresponding BWT values are compared. If they are distinct and the two corresponding SA values do not appear in the same file ($S1$ or $S2$), then the SA values represent the location of a MuM.

(1) Store the contents of SLB[i] (init. $i=0$) into an empty reusable buffer and maintain a counter C (init. 1) of its entries. Label the current LCP value as the `previous LCP`.

(2) Fetch the next entry from the SLB ($i=i+1$ then SLB[i]). Its LCP value now represents the `current LCP`. Increment C by 1 if and only if the previous step was not Step 1.

(3) If `current LCP` > `previous LCP` and $C=1$, append the contents of SLB[i] to the buffer. Update the `previous LCP` label and back to Step 2.

(4) If `current LCP` = `previous LCP` and $C=1$, then this is not an L-interval. Go to Step 1.

(5) If `current LCP` > `previous LCP` and $C\geq 2$, then this can not be a local maxima. Go to Step 1.

(6) If `current LCP` = `previous LCP` and $C\geq 2$, append the contents of SLB[i] to the buffer. Update the `previous LCP` label and go back to Step 2.

(7) If `current LCP` < `previous LCP` (or SLB exhausted) and $C=2$ then check if the first and second BWT values in the buffer are distinct; check if the first SA value in the buffer $<p$ while the second $>p$ or visa-versa (p is the location in bytes of '#' in the text file). If both these conditions are met then the local maxima is a MuM.

(8) Go back to Step 1 and repeat until SLB is exhausted.

2.4 Pairwise branching tandem repeats

Pairwise branching tandem repeats (Abouelhoda *et al.*, 2006) (which we label as a *tandem* repeat for brevity) consist of two or more substrings comprised of two or more nucleotides in a DNA sequence that are identical and occur adjacent to each other. For example, in the sequence ATCGATCGGCAT, the substring 'ATCG' occurs twice and is thus a tandem repeat. The tandem repeats occur frequently in genomes; for instance, nearly 10% of the human genome is composed of such repeats.

Several programs to locate repeated strings in sequences exist. These include tools such as REPuter (Kurtz *et al.*, 2001), Tandem Repeats Finder (Benson, 1999), Tandem Repeats Analyzer (Bilgen *et al.*, 2004) and mreps (Kolpakov *et al.*, 2003). However, existing tools are limited in the number and size of sequences, the length of repeats that can be detected and range of DNA sequence formats supported. Due to its significant applications, it is important that a scalable algorithm is developed to process large disk-resident sequences. We present an existing in-memory algorithm for the computation of tandem repeats (Abouelhoda *et al.*, 2006) in Supplementary Material, which we base our following discussion on.

A key step of the algorithm requires us to find and process each L-interval in a sequence where L is $\geq$ an LCP value ($L>0$). Since we are no longer restricted to processing only local maximas, we require an algorithm that finds all L-intervals in an efficient manner, which on disk implies scanning the SLB from left-to-right. Fortunately, a simple linear stack-based algorithm exists that is well suited for our purpose (Abouelhoda *et al.*, 2006). When an element is popped from the stack (which represents an L-interval), it is fed to the tandem repeat algorithm shown in Supplementary Material. However, the tandem repeat algorithm presented cannot be mapped directly to disk without incurring substantial performance penalties.

We require an inverse SA also known as a *rank* array. Since the rank array is accessed frequently (once for every entry in the L-interval) and at random locations, maintaining it on disk is not a feasible solution. We, therefore, need to eliminate the use of the ranked array, if we are going to realize a viable solution on disk.

We eliminate the rank array by using the SLB array together with an optimized binary search routine. When we are given an L-interval $[i,\ldots,j]$

Table 1. The size (million base pairs approximately) of our human genome datasets with an alphabet size of 4, showing the chromosome sequence used, the largest L-interval (entries contained) with $L \geq 4$, and the number of supermaximal repeats and MuMs with $L \geq 30$

Size (Mbp)	SLB (s)	SLB (MB)	Chromosome sequence	Max. interval size, $L \geq 4$	No. repeats $L \geq 30$	
25.7	1.1	230.8	chrY	399 879	147 681	supermax
60.8	1.3	546.8	chrY,chr21	938 808	358 314	supermax
95.7	3.5	860.8	chrY,chr21,chr22	1 370 089	709 102	supermax
151.5	5.9	1363.1	chrY,chr21,chr22,chr19	2 144 861	1 598 991	supermax
211.0	14.1	1898.7	chrY,chr21,chr22,chr19,chr20	2 973 273	2 131 873	supermax
285.7	23.7	2570.6	chrY,chr21,chr22,chr19,chr20,chr18	4 148 856	2 674 267	supermax
363.5	37.4	3270.7	chrY,chr21,chr22,chr19,chr20,chr18,chr17	5 257 139	3 686 434	supermax
442.4	51.0	3980.7	chrY,chr21,chr22,chr19,chr20,chr18,chr17,chr16	6 361 943	4 613 547	supermax
60.8	1.6	546.8	chrY#chr21	–	64 960	MuM
115.3	5.4	1037.8	chr19#chr20	–	256 311	MuM
156.7	11.1	1410.1	chr16#chr17	–	352 467	MuM
183.9	11.4	1654.9	chr13#chr14	–	279 282	MuM
262.4	25.0	2361.9	chr10#chr11	–	451 068	MuM

The # character is the divisor between two datasets. The SLB construction time (s) and disk space requirements (MB) are also shown for our $9n$ (default) SLB.

from the stack, we seek to the position i in the SLB array and buffer all SLB entries up to j inclusive. This step incurs a potential random disk access to the SLB, but this is still better than incurring several random disk accesses to a rank array. Now that we have buffered the entire L-interval in memory, we can access the SA value of each entry in the interval, one at a time, adding L to it (temporarily) and then searching all remaining SA values in the interval for a match. If found, then we know that the current SA value $+L$ exists within the interval i and j. However, searching each SA value n times, where n is the number of entries in the interval, is an n^2 problem. Hence, this is not an efficient solution.

Instead, once we buffer the L-interval in memory, we iterate through each entry and copy (append) its SA value to an array/buffer. The array is then sorted using quick sort. Once sorted, we conduct an optimized binary search as follows: iterate through each SA value in the sorted array and conduct a binary search looking for that element $+L$. If found, we know that the current SA value $+L$ exists within the interval i and j.

Otherwise, we access the next element in the L-interval, add L, and redo the binary search, repeating in this manner until we exhaust the array. On each repetition (including the first), we reduce the number of items searched by 1 (truncating the start of the array). This forms an *optimized* binary search thats allows large L-intervals to be processed more efficiently.

Another concern to performance is the access made to the text file, where two characters are fetched and compared. However, since access to the text is localized between $S[p+L]$ and $S[p+2L]$, only a single disk seek is required to fetch a strip of text that begins at $p+L$ and ends at $p+2L$ inclusive. If the size of this strip exceeds some threshold, say 128 MB, then we can revert back to conducting two random seeks. However, for our data, the largest text strip was found to be only a few kilobytes long. Alternatively, we can eliminate disk accesses altogether by buffering the entire text in memory, assuming we have enough space to do so.

The potential drawback of our adaptation to the existing tandem repeat algorithm is the need for sufficient memory to buffer an entire L-interval, and the corresponding text strip and/or the entire text file. Each SLB entry consumes 12 bytes of memory (9 bytes on disk but 12 bytes in-memory due to alignment constraints). Though, as we show in later experiments, even for large sequences, the amount of memory consumed in this manner is relatively small. Furthermore, an L-interval can be easily processed in smaller fragments, allowing one to impose a (reasonable) bound on space consumption. As such, our algorithm remains a scalable solution.

In contrast, our disk-based implementations of the supermaximal and MuM algorithms do not require us to buffer the entire L-interval, and are thus more memory-efficient. With these two algorithms, we only need to retain the current local maxima in memory and one that adheres to strict conditions that enforce a bound on its size.

3 EXPERIMENTAL METHODOLOGY

Our experiments were conducted on a 64-bit AMD Athlon II 240 processor running at 2.8 Ghz with 4 GB RAM and a 64-bit Linux operating system, which was kept under light load (no GUI, single user mode). We compiled our software using g++ (ver. 4.4.3) with all optimizations enabled (-O3). Further details are provided in Supplementary Material. The datasets used are shown in Table 1, derived from actual biological data, namely the complete *hg19* human genome.[1] We extracted a selection of chromosomes from *hg19*, which were pre-processed to ensure upper case letters (otherwise RepMaestro would consider $g \neq G$ for example) and also to (optionally) remove all ‘N’ characters, for brevity. RepMaestro assumes an alphabet size of 4 (5 including ‘N’); though not recommended, it can also accept larger alphabets (including all printable ASCII characters).

The Y chromosome formed our smallest dataset. We then concatenated it with an orderly sequence of chromosomes, creating eight datasets in total. By growing the datasets in this manner (as subsets), we can better illustrate the scalability of our algorithms in comparison to Vmatch. Similarly, we combined a selected pair of (pre-processed) chromosomes together to form a single sequence (five in total and increasing in size) separated by the unique # character, for use in MuM processing.

***Comparisons made*:** we compare the performance of RepMaestro (on supermaximal repeats, MuM and pairwise branching tandem repeats) against `Vmatch` by measuring the Wall time (s) using the Linux `time` command. We enabled the following options in Vmatch to reduce the output size and to allow for more comparable results: `-nodist -noscore -noidentity -noevalue -absolute`. Measurements were averaged over 10 runs—the SD of which was low. After each run, main memory was flushed with random data and the disks were unmounted then

[1]http://hgdownload.cse.ucsc.edu/goldenPath/hg19/bigZips/hg19.2bit.

Table 2. The wall time in seconds required to find all supermaximal repeats with LCP$\geq$30 using RepMaestro supermax against Vmatch

Mbp	25.7	60.8	95.7	151.5	211.0	285.7	363.5	442.4
Vmatch	2.8	6.0	9.7	15.5	22.6	31.4	–	–
RepMaestro	2.3	4.8	7.5	11.9	16.2	22.1	27.9	33.8

Table 3. The wall time in seconds required to find all MuMs with an LCP$\geq$30

	chrY#chr21	chr19#chr20	chr16#chr17	chr13#chr14	chr10#chr11
Vmatch	30.7	52.3	74.0	77.9	125.2
MUMmer	39.5	78.4	110.8	134.5	197.7
RepMaestro	4.8	8.8	12.1	14.2	20.1

Datasets are increasing in size from left to right.

remounted to flush system buffers, to allow for fairer comparison between an in-memory solution (Vmatch) and our disk-based solution (RepMaestro). We conducted our experiments by directing all output to `/dev/null`. Our results are based on our 9*n* SLB; 17*n* SLB offers the same scalable traits, but as expected, requires approximately double the disk space and processing time.

Since RepMaestro is a 64-bit compatible software and considering that we used a 64-bit computing architecture, we relied on the 64-bit version of `Vmatch`; we also provide results of the 32-bit version in Supplementary Material. We verified the correctness of our software by cross checking with the results generated by Vmatch. In addition, we compare against `SeqAn`[2] and MUMmer but due to space constraints, we provide most of their results in Supplementary Material and focus on Vmatch (64 bit) below, since it is a state-of-the-art software that supports most repeats.

4 RESULTS AND DISCUSSION

4.1 Supermaximal repeats

Table 2 shows the time required to find all supermaximal repeats with an LCP$\geq$30 (i.e. with an LCP of 30, or alternatively, the minimum length of matches). We also provide results for an LCP$\geq$15 in Supplementary Material. RepMaestro is consistently faster than Vmatch and scales well with increasing sequence size. With a genome sequence size of 25.7 million base pairs (Mbp) for instance, RepMaestro is marginally faster than Vmatch. As we increase the size of the sequence to 285.7 Mbp—the operational limit for Vmatch on a 64-bit system with 4 GB of RAM—RepMaestro is $\sim$30% faster, but more importantly, it can continue to process 363.5 and 442.4 Mbp with a linear increase in processing time. SeqAn also displayed similar performance to Vmatch, being consistently slower and less scalable than RepMaestro and rapidly exhausting main memory. These results provide a strong indication that RepMaestro is indeed a more scalable solution for finding supermaximal repeats.

RepMaestro is a superior and scalable solution due to its single linear scan of the SLB array. This action results in optimal use of disk since no random accesses are made. In addition, this sequential processing also prompts efficient use of CPU cache, further boosting performance. Since the alphabet size is only 4 characters and that only one local maxima is buffered at any given time, memory consumption remains bounded. In this case, a local maxima that is a supermaximal repeat can have a cardinality of at most 4. As a result, the amount of memory consumed by RepMaestro for detecting supermaximal repeats is tiny. In practice, memory consumption will typically fluctuate by potentially several hundred megabytes, as a result of the SLB prefetch buffer, which is bounded and configurable in size (see Supplementary Material). Thus, RepMaestro, unlike Vmatch, does not require space in proportion the genome sequence size processed, making RepMaestro is an attractive solution for both high- and low-end computer systems.

4.2 MuMs

We now investigate the performance of Vmatch and RepMaestro in detecting MuMs using our five MuM datasets shown in Table 1. The time required to find all MuMs with an LCP$\geq$30 is shown in Table 3. We also provide results for an LCP$\geq$15 in Supplementary Material. As observed with our previous experiments involving supermaximal repeats, RepMaestro is faster and importantly, more scalable than Vmatch as a result of its single linear scan of an SLB array. Considering *chr10#chr11*, which is our largest MuM sequence (262.4 Mbp), RepMaestro is over an order of magnitude faster than both Vmatch and MUMmer, and is shown to be a scalable solution with linear processing time. In this experiment, Vmatch and MUMmer also incur a performance overhead as a result of the mandatory requirement (to the best of our knowledge) of first processing the required indexes in-memory prior to computing any MuMs. Thus, our results provide a strong indication that RepMaestro is indeed a superior and scalable solution to Vmatch for finding MuMs. As discussed with our supermaximal experiments, memory usage is small and bounded in size, but in practice, it can fluctuate by potentially several hundred megabytes as a result of SLB prefetch (see Supplementary Material).

4.3 Pairwise branching tandem repeats

We now investigate the performance of Vmatch and RepMaestro to detect pairwise branching tandem repeats or just tandem repeats (for

[2]www.seqan.de/.

Table 4. The time required in seconds to find tandem repeats as a function of the min repeat length and the sequence size

Mbp	25.7	60.8	95.7	151.5	211.0	285.7	363.5	442.4
Min LCP	VMatch							
4	4.9	11.2	18.2	30.5	43.3	57.4	–	–
8	4.2	10.9	17.6	29.2	41.5	55.0	–	–
16	3.3	7.3	11.8	19.9	27.8	36.2	–	–
32	2.8	6.0	9.8	16.0	22.3	30.2	–	–
Min LCP	RepMaestro *(with text buffering)*							
4	32.6	81.6	135.6	230.4	330.1	456.3	607.7	757.6
8	20.9	51.8	86.7	151.9	219.5	300.6	395.4	494.9
16	7.8	16.9	29.1	56.0	79.8	107.7	145.8	187.3
32	5.1	10.4	16.8	28.9	40.7	54.8	72.0	90.2
Min LCP	RepMaestro *(without text buffering)*							
4	38.7	95.8	158.3	266.7	381.0	530.4	688.4	939.7
8	30.6	78.3	125.6	207.7	303.7	420.7	540.0	846.1
16	17.7	43.1	75.1	140.2	195.4	265.1	342.1	445.0
32	11.4	25.2	42.0	69.1	99.9	126.9	164.2	210.7

brevity). The results are shown in Table 4. The genome sequences (increasing in size) represent the columns of the Table. We also vary the LCP from a minimum of 4 to a maximum of 32, represented as the rows. With an LCP ≥ 4, the locations of all tandem repeats that are at least 4 characters long are found.

For all the cases considered, RepMaestro was slower than Vmatch. Unlike its application in supermaximal repeats and MuMs, when applied to tandem repeats, RepMaestro incurs a considerable amount of overhead even for small files. This is because not all tandem repeats can be found with a single scan of the SLB array. Since L-intervals can be subsets of a larger interval, we must rely on the linear stack-based algorithm described in Section 2.4 to efficiently provide us with L-intervals. We only consider L-intervals with a L-value (length of the smallest common prefix of all entries in the L-interval) $\geq$ the minimum LCP, which is either 4, 8, 16 or 32. Hence, the stack is small in size, but even with this pruning, we still need to buffer an entire (or ordered fragments of an) L-interval from disk. This can be an expensive ordeal relative to Vmatch, which maintains its enhanced SA in memory.

As shown in Table 1, some L-intervals particularly those with small L can contain many millions of entries. As a consequence, as we reduce the minimum LCP considered from 32 to 4, the number of L-intervals processed typically increase, since more genome sequences are likely to share shorter common prefixes. This unfortunately implies more disk I/O that will impact performance as reflected in Table 4.

Consider the results for our 285.7 Mbp dataset with RepMaestro that buffers the entire genome text in memory. The advantage of text file buffering is that it eliminates the random disk seek incurred when we fetch the pair of characters required by the tandem repeat algorithm (see Supplementary Material). With a minimum LCP of 32, RepMaestro requires ∼55 s. Although this is ∼44% slower than Vmatch, RepMaestro's performance is still reasonable considering that Vmatch cannot process larger sequences due to its high memory requirements. However, when we reduce the minimum LCP to 4, the time taken by RepMaestro increases by an order of magnitude—to ∼456 s. We observe similar results when we disable text file buffering, with the performance of RepMaestro degrading in all cases due to the additional disk costs associated with fetching the required characters from disk.

Although RepMaestro requires more time relative to Vmatch, the cost is linear relative to sequence size. This is also the case with Vmatch, but unfortunately, Vmatch cannot process genome sequences >285.7 Mbp unless sufficient memory is available—which is somewhat difficult to justify since processing a 363.5 Mbp sequence exhausts 4 GB of RAM. Processing the entire human genome (3 Gb) would, therefore, require many tens of gigabytes of RAM, which is currently uncommon in standard workstations. Thus, even though RepMaestro is slower, it compensates with scalability, offering an immediate and feasible solution for large-scale tandem repeat computations.

With respect to memory consumption, RepMaestro requires enough memory (assuming no interval fragmentation) to buffer the largest L-interval considered and preferably space for the entire genome sequence—along with space for the stack, implemented as a single resizable array. Given our 442.4 Mbp genome sequence and an LCP ≥ 4, the longest L-interval processed contained 6 361 943 entries, consuming about 77 MB on a 32-bit system and around 153 MB on a 64-bit system—a relatively tiny amount; and with a total of 303 831 513 L-intervals, the maximum stack size was only 249 entries (2988 bytes). Moreover, a large L-interval can be split into smaller fragments, allowing RepMaestro to operate with a reasonable bound (say, at least 512 MB) on memory consumption. In practice, however, as discussed with supermaximal repeats and MuM, memory consumption can fluctuate by several hundred megabytes as a result of SLB prefetch.

Furthermore, our results are based on conventional hard drives, which are still a common method of non-volatile storage. Solid state hard drives, however, are gaining popularity since they can offer faster and consistent transfer speeds and importantly, substantially reduced random seek costs. Our RepMaestro software is likely to further benefit from such a drive, allowing it to operate more efficiently, which is an interesting avenue for future research.

5 CONCLUSION

The analysis of repetitive sequences plays a key role in genome analysis. Such repetitive regions are abundant in genome sequences and these have been demonstrated to have several practical biological applications such as in the determination of parentage and in forensics. While there are several efficient algorithms available that enable the detection of repetitive sequences, these are mainly restricted to smaller memory-resident sequences and, therefore, can not scale well or operate on large sequences that are stored on disk. We have, therefore, introduced RepMaestro, a software that implements algorithms that allows us to compute supermaximal repeats, MuMs and tandem repeats of a large genome sequence stored on disk, in a scalable and memory-efficient manner. In most cases, we outperform a state-of-the-art memory-resident algorithm while simultaneously consuming significantly less (and bounded) memory. This allows RepMaestro to be an immediate solution for the problem of rapidly processing large genome sequences that are many times larger than the available main memory. RepMaestro is, however, dependant on third-party software(s) to generate the required indexes, and so, its applicability is subjected to their operational constraints (see Supplementary Material).

ACKNOWLEDGEMENTS

We thank Prof. Kurtz for providing Vmatch.

Funding: The initial 2007-2009 research was supported by the Australian Research Council Discovery Project [DP0771504] and the Uni. of Melb. Early Career Research Grant (00080) of R.S. The 2010 research and submission was supported by personal funds and resources.

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.

Abouelhoda,M.I. *et al.* (2006) *Enhanced Suffix Arrays and Applications*. CRC Press. Available at http://www.crcnetbase.com/doi/abs/10.1201/9781420036275.ch7.

Altschul,S.F. *et al.* (1997) Gapped BLAST and PSI-BLAST: a new generation of protein database search programs. *Nucleic Acids Res.*, **25**, 3389–3402.

Benson,D.A. *et al.* (2007) Genbank. *Nucleic Acids Res.*, **35**, D21–D25.

Benson,G. (1999) Tandem repeats finder: a program to analyze DNA sequences. *Nucleic Acids Res.*, **27**, 573–580.

Bilgen,M. *et al.* (2004) A software program combining sequence motif searches with keywords for finding repeats containing DNA sequences. *Bioinformatics*, **20**, 3379–3386.

Cameron,M. *et al.* (2004) Improved gapped alignment in BLAST. *IEEE/ACM Trans. Comput. Biol. Bioinform.*, **1**, 116–129.

Chain,P. *et al.* (2003) An applications-focused review of comparative genomics tools: capabilities, limitations and future challenges. *Brief. Bioinform.*, **4**, 105–123.

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

Dementiev,R. *et al.* (2008) Better external memory suffix array construction. *ACM J. Exp. Algorithmics*, **12**, 1–24.

Gusfield,D. (1997) *Algorithms on Strings, Trees, and Sequences: Computer Science and Computational Biology*. Cambridge University Press, Cambridge.

Homann,R. *et al.* (2009) mKESA: enhanced suffix array construction tool. *Bioinformatics*, **25**, 1084–1085.

Hon,W.-K. and Sadakane,K. (2002) Space-economical algorithms for finding maximal unique matches. In *Proceedings of the Annual Symposium on Combinatorial Pattern Matching*, volume 2373 of *Lecture Notes in Computer Science, Fukuoka, Japan*, Springer, London, UK, pp. 17–29.

Jurka,J. and Batzer,M.A. (1996) *Human Repetitive Elements, in Encyclopedia of Molecular Biology and Molecular Medicine*. Vol. 3. VCH Publishers, Weinham.

Kolpakov,R. *et al.* (2003) mreps: efficient and flexible detection of tandem repeats in DNA. *Nucleic Acids Res.*, **31**, 3672–3678.

Kurtz,S. (2008) Vmatch: large scale sequence analysis software. Available at http://www.vmatch.de (last accessed date September, 2009).

Kurtz,S. *et al.* (2001) Reputer: the manifold applications of repeat analysis on a genomic scale. *Nucleic Acids Res.*, **29**, 4633–4642.

Kurtz,S. *et al.* (2004) Versatile and open software for comparing large genomes. *Genome Biol.*, **5**, 1–9.

Kurtz,S. *et al.* (2008) A new method to compute k-mer frequencies and its application to annotate large repetitive plant genomes. *BMC Genomics*, **9**, 1–18.

Leung,M. *et al.* (1991) An efficient algorithm for identifying matches with errors in multiple long molecular sequences. *J. Mol. Biol.*, **221**, 1367–1378.

Lian,C.N. *et al.* (2008) Searching for supermaximal repeats in large DNA sequences. In *Proceedings of the 2nd International Conference on Bioinformatics Research and Development, Vienna, Austria*, Springer, Berlin, Heidelberg, pp. 87–101.

McConkey,E.H. (1993) *Human Genetics: The Molecular Revolution*. Jones and Bartlett Publishers, 40 Tall Pine Drive Sudbury, MA.

Moffat,A. *et al.* (2009) Reducing space requirements for disk resident suffix arrays. In *Proceedings of the Database Systems for Advanced Applications, Brisbane, Australia*, Springer, Berlin/Heidelberg, pp. 730–744.

Phoophakdee,B. and Zaki,M.J. (2007) Genome-scale disk-based suffix tree indexing. In *Proceedings of the ACM SIGMOD International Conference on Management of Data, Beijing, China*, ACM, New York, NY, USA, pp. 833–844.

Sinha,R. *et al.* (2008) Improving suffix array locality for fast pattern matching on disk. In *Proceedings of the ACM SIGMOD International Conference on Management of Data, Vancouver, Canada*, ACM, New York, NY, USA, pp. 661–672.

Smith,J.M. (1998) *Evolutionary Genetics*. Oxford University Press, NY.

Smyth,B. (2003) *Computing Patterns in Strings*. Addison-Wesley, Reading, MA.

Watson,J. *et al.* (1987) *Molecular Biology of the Gene*. Benjamin Cummings, Menlo Park, CA.

Williams,H.E. and Zobel,J. (2002) Indexing and retrieval for genomic databases. *IEEE Trans. Knowledge Data Eng.*, **14**, 63–78.