4th IEEE International Conference on Computational Advanced in Bio and Medical Sciences, Miami Beach, Florida, June, 2014

# SAIS-OPT: On the Characterization and Optimization of the SA-IS Algorithm for Suffix Array Construction

Nataliya Timoshevskaya, Wu-chun Feng
Department of Computer Science, Virginia Tech
Blacksburg, Virginia, USA
{timnatvt, wfeng}@vt.edu

***Abstract*—The suffix array and Burrows-Wheeler Transform are critical index structures in next generation sequence analysis. The construction of such index structures for mammalian-sized genomes can take thousands of seconds (i.e. tens of minutes). Its construction is complicated by computational overheads that coming from irregular or complex memory-access patterns. This paper rigorously characterizes the execution profile of the SA-IS algorithm in order to guide its optimization. The resulting optimized SA-IS, which we refer to as *sais-opt*, outperforms previous implementations of SA-IS as well as "best in practice" algorithms, when applied to large DNA strings.**

***Keywords—suffix array; Burrows-Wheeler Transform; irregular memory access***

## I. Introduction

The rise in prominence of the Burrows-Wheeler Transform (BWT) coincided with the introduction and adoption of the Burrows-Wheeler Aligner (i.e., *bwa*) by the bioinformatics community. This aligner, the first BWT-based short-read alignment application [1], builds a BWT-based index of a reference genome to accelerate short-read mapping. The FM-index allows for searching in compressed BWT files without full decompression, which is particularly important in the "Big Data" era of next-generation sequencing [2]. Along with short- and long-read alignment, whole-genome comparison [3] and reference-assisted assembly programs [4] require BWT construction for the short and long strings. The performance of some *de novo* assemblers also depends on the time of "on-the-fly" BWT construction [5, 6, 7]. For all these applications the design of fast BWT construction algorithms and corresponding software is of great importance.

A suffix array (SA) is a sorted array of all suffixes of a given string. SA is a self-sufficient data structure in biological sequence analysis [2], but it also can be used for the construction of other complex index structures, such as the FM-index or the longest common prefixes array, including BWT. While building the BWT via SA requires several times more memory, than some direct approaches [8], it results in a significantly faster implementation than with the direct methods. The construction of the SA for a human genome, for example, requires approximately 15 GB of memory (DRAM), which is not an inordinate amount for modern scientific hardware. As such, the focus of this paper addresses the SA construction.

All suffix-array construction algorithms (SACAs) can be classified, with respect to their theoretical time complexity, as linear-time algorithms or super linear-time algorithms. In practice, some of the super linear-time algorithms deliver better performance than algorithms with linear-time complexity. While latter may not be as fast, they can guarantee stable performance for any inputs. The SA-IS algorithm, published by Nong et al. in 2008 [9], is currently the best-known linear-time algorithm. It is based on an induced sorting method and is used as the default algorithm for BWT construction in *bwa* for inputs of 2GB and less. Its elegance and modest space requirements, compared to other SACAs, led us to thoroughly analyze SA-IS in order to guide its optimization and benefit many genome analysis applications.

This paper makes the following contributions:

1. A rigorous characterization and analysis of different stages of the SA-IS algorithm, which revealed that the most time-consuming stages experience a highly irregular memory-access pattern.

2. *sais-opt* – a co-designed optimization of the SA-IS algorithm, which achieves a 27% improvement in performance (on average) on tested data, when compared to the already highly tuned implementation of SA-IS in *bwa*.

3. An experimental evaluation of *sais-opt* and six other SACAs using a set of standard benchmarks along with a set of large-sized DNA sequences (1-3 GB). The results of the evaluation show that on tested data *sais-opt* almost always delivers the best performance among the linear-time algorithms and is also capable of competing with the "best in practice" SACA applications, like *libdivsufsort*, and outperforms them on the aforementioned large-sized DNA sequences.

## II. Related Work

An extensive survey and classification of suffix sorting algorithms published before 2008 can be found in [10]. The three most popular linear-time algorithms [9, 11, 12] share a recursive approach and induced sorting ideology. In spite of the theoretical linear-time complexity, in practice, these algorithms perform worse than some of their super linear-time

This work was supported in part by NSF IIS-1247693 via the NSF BIGDATA program.

978-1-4673-1321-6/12/$31.00 ©2014 IEEE

counterparts, i.e., O($n$log$n$). On the other hand, super linear-time algorithms can behave very poorly on some inputs.

An evaluation of many algorithms against different benchmarks has been done by Yuta Mori (http://code.google.com/p/libdivsufsort/) – the author of the *libdivsufsort* library and the *bwa-is* and *sais-lite* implementations of the SA-IS algorithm. Along with *libdivsufsort,* the *msufsort* and *anchor* programs realize a mixture of different suffix-sorting methods. We picked *libdivsufsort* to be a representative of these so-called "best in practice" algorithms. Two other fast algorithms BPR v.2.0 [13] and radixSA [14] rely on enhanced radix sorting and are evaluated in the results section.

Existing SA-IS algorithms seek to improve either space or performance characteristics. For example, SAR-IS [15] and SACA-K [16] reduce space consumption, while *bwa-is* and *sais-lite* significantly improve both, when compared to the original implementation of SA-IS [9]. Our work addresses the memory-bounded character of SA-IS algorithm. An abundance of memory access operations, often in an irregular manner, became a real curse when the algorithm is applied to the large-sized inputs. While keeping an original workflow of *SA-IS*, we made several algorithmic changes of its main stages by structuring and manipulating data with the goal to reduce the number of memory access operations. We used the *bwa-is* implementation of SA-IS as our baseline and significantly improved its performance characteristics.

The rest of the paper is organized as follows. Section III provides background on the induced sorting technique and the SA-IS algorithm and then discusses some specific details of the SA-IS implementation in *bwa*. Next, in section IV, we introduce our optimized version of SA-IS. Finally, section V presents our experimental results, followed by conclusion and discussion of future work.

## III. SA-IS ALGORITHM

### A. *Induced sorting technique and SA-IS algorithm*

Let $T = (t_0, t_1, \ldots, t_n)$ be a string of $n$ characters over the alphabet $\Sigma\cup\{\$\}$, represented as an array indexed by $[0\ldots n]$, where $t_0, t_1, \ldots, t_{n-1} \in \Sigma$ and $t_n = \$$ is a unique character that is lexicographically smallest. \$ is used only for description purposes. Denote $T_i = (t_i, t_1, \ldots, t_n)$ the $i$-th suffix of $T$ and $T_{ij} = (t_i, t_{i+1}, \ldots, t_j)$ the substring of $T$. The *suffix array* SA($T$) is the permutation of the integers 0, …, $n$–1, such that $T_{\mathrm{SA}(T)[\mathrm{i-1}]} < T_{\mathrm{SA}(T)[\mathrm{i}]}$, where "<" means lexicographical order.

Below, there is a high-level description of SA-IS algorithm.

1. Reduce input string $T$ to a shorter string $T$.'
   a. Sort all LMS-substrings of string $T$ in the lexicographical order and compute their ranks.
   b. Build $T$' by replacing all LMS-substrings in $T$ with their ranks. Treat ranks as characters of $T$'.
2. If all characters in $T$' are different, construct SA($T$') directly; else apply SA-IS to construct SA($T$').
3. Induce the order of all suffixes in $T$ from the revealed from SA($T$') order of LMS-suffixes.

The algorithm is based on three main ideas: 1) induced sorting, which induces the order of unsorted suffixes from a set of already sorted key suffixes (applied on step 3); 2) recursive reduction to a shorter string (steps 1, 2); and 3) applying the same induced sorting technique in the reduction step (step 1a). Below we describe them briefly.

First, in SA, all suffixes are partitioned to different buckets according to their first character. The idea of induced sorting comes from the recursive definition for lexicographical order of two suffixes: $T_i < T_j$, if (1) $t_i < t_j$ or (2) $t_i = t_j$ and $T_{i+1} < T_{j+1}$. The first condition regulates the choice of the bucket, and the second one enables the identification of the order of the suffixes within the bucket. The information about the order of $T_{j+1}$ is used to induce the order of the suffix $T_{j.}$

Algorithms that use induced sorting differ from each other depending on how they define and sort key suffixes. The work of Ko and Aluru [12], which preceded the work of the SA-IS algorithm, distinguishes suffixes of type S and L. If $Ti < T_{i+1}$, then $T_i$ has type S (states for Smaller), and L (states for Larger) otherwise. The sentinel character \$ is always a S-suffix. If the relative lexicographic order of all S-suffixes (L-suffixes) is known, then the order of all L-suffixes (S-suffixes) can be induced iteratively from the smallest to largest (largest to smallest). The above terminology was then used in turn by SA-IS algorithm.

Ko and Aluru proposed to put into the key set only the *LMS-suffixes* (the Left Most S-type suffixes), i.e., *S*-suffixes with the left neighbor being a L-suffix. If the relative lexicographic order of all LMS-suffixes is known, then the order of all L-suffixes can be induced iteratively smallest to largest. After that, the order of all S-suffixes can also be induced iteratively largest to smallest.

Second, to make induced sorting possible, the LMS-suffixes must be sorted first. A recursive approach exploited by SA-IS for this task allows to keep its time complexity to be linear. The approach incorporates a "ranking" technique – replacement of substrings of initial string, which assigns to them names or ranks. For each LMS-suffix $T_i$ we define the *LMS-substring* $T_{ij} = (t_i, \ldots t_j)$, where $T_j$ is the next LMS-suffix. The sentinel character \$ itself presents the LMS-substring. Each non-sentinel LMS-substring has at least 3 characters and overlaps with the next LMS-substring by one character. For the sake of brevity, we refer to the first character in LMS-substring as the *LMS-character* and the index of the LMS-character in $T$ as the *LMS-index*. For the LMS-substring $G$, $rank(G) = k$, if exactly $k$ unique LMS-substrings are lexicographically smaller than $G$. Equal LMS-substrings have the same rank. If all LMS-substrings are different, the order of all LMS-suffixes is defined. In the opposite case, by replacing each LMS-substring in the initial string $T$ with its rank, a new string $T$' over alphabet $\{0,\ldots, m\text{-}1\}$ is constructed. Here $m$ is the number of different LMS-substrings in $T$ and $m < n/2$. The suffix array for string $T$' gives the linear order of all suffixes in $T$' and therefore its corresponding LMS-substrings.

Third, SA-IS applies the same induced sorting technique in the reduction step, as proposed by Nong et al. [9], in order to sort the LMS-substrings.

### B. *Implementation of SA-IS in bwa*

*The bwa-is* algorithm is used in *bwa* for the construction of BWT for files of size $n$ < 2G, which requires 5.37$n$ bytes of memory. For larger files for the sake of reducing memory footprint, *bwa* constructs BWT with much more slower *bwtsw* program based on work [8]**.** But today, having 16 or more gigabytes of RAM is not unusual, and that is enough to build BWT for the human genome size sequences via SA.

In contrast to the original implementation of SA-IS, presented in [9], *bwa-is* avoids the use of extra memory to store type of each suffix and locations of all LMS-suffixes (up to $n$/2). The identification of type of suffixes (L, S, or LMS) must be done "on-the-fly" and requires extra time. Nevertheless, experimental testing reveals, that owing to thorough engineering code design, *bwa-is* has 1.5 times better performance compare to the original SA-IS. Because of the popularity of *bwa* and improved space and time characteristics of *bwa-is* we chose it as a framework for our optimizations.

The *bwa-is* consists of eight main stages (Fig. 1). On Fig.2 we show the time distribution between these stages, obtained experimentally as an average across several strings with different sizes and alphabets.

```
bwa-is (T)
1. LMS-indexes search and placement
   1) compute buckets;
   2) find and place all LMS-indexes at the end of corresponding buckets
2. Sorting of LMS-substrings*
3. Gathering of LMS-indexes*: gather LMS-indexes, which are spread across SA, to the beginning part of SA.
4. T' Construction:
   1) Find length of each LMS-substring and store it in a special position associated with its index
   2) Ranking*: find rank of each LMS-substring
   3) New string construction:
      If all LMS-substrings are different, go to step 7;
      else construct string T' by replacing LMS-substrings of T with their ranks.
5. Recursion*: recursively build SA(T')
6. Unranking: replace characters in T' with the corresponding LMS-indexes and place latter at the beginning part of SA preserving their order from SA(T').
7. Placement: place LMS-indexes in the corresponding buckets, preserving their order.
8. Induced sorting*: induce the order of all suffixes from the ordered LMS-suffixes.
```

Fig.1. Eight main stages of *bwa-is*, stages marked with * are optimized in *sais-opt*

## IV. OPTIMIZATIONS

The main target of our work is the construction of SA for long strings. The *bwa-is* exhibits a very irregular memory access pattern, which critically affects its performance on large-sized inputs. For example, the common set of operations used on the Induced sorting stage (stage 8) is similar to the next line of C code:

```
j=SA[i]; SA[B[T[j-1]] = j-1;
```

where B is a bucket array. The input string *T* and the suffix array SA can be very large arrays, e.g. 3 and 12 gigabytes respectively for the human genome. The diagram on Fig.3 instantiates that the growth of string size leads to the increase of cache-misses rate up to 50-60% level, and after a certain point the program behaves equally "bad" with respect to the number of cache-misses. The better performance can be achieved by reducing the number of memory access operations, especially in an irregular manner. Below we describe the applied optimizations in details.

#### 1) *Optimized sorting of LMS-substrings and gathering*

The novelty of SA-IS is the ability to use the induced sorting, not only to induce the order of all suffixes from the sorted LMS-suffixes (stage 8), but also to sort LMS-substrings (stage 2). *bwa-is* literally applies the same function for both stages. We improved the performance of stages 2 and 3 by taking into account that on stage 2 only the LMS-indexes need to be sorted. The goal of stage 3 is to gather all sorted LMS-indexes in the beginning part of SA to release space for further computations. If the general induced sorting procedure was applied, the LMS-indexes would be spread across SA among the S- and L-indexes. On stage 3, to distinguish the LMS-indexes from the others, *bwa-is* has to scan each element in SA and read and compare corresponding characters from string *T*. Moreover, for each index with the left neighbor of L-type, the comparison of more than two characters can be required. Because of an irregular access to string T for the "on-the-fly" type identification, this simple stage takes about 7% of the overall time of *bwa-is* (Fig. 2).

We have modified stage 2, so that, when it is done, all the elements in SA, except the LMS-indexes, have zero values.

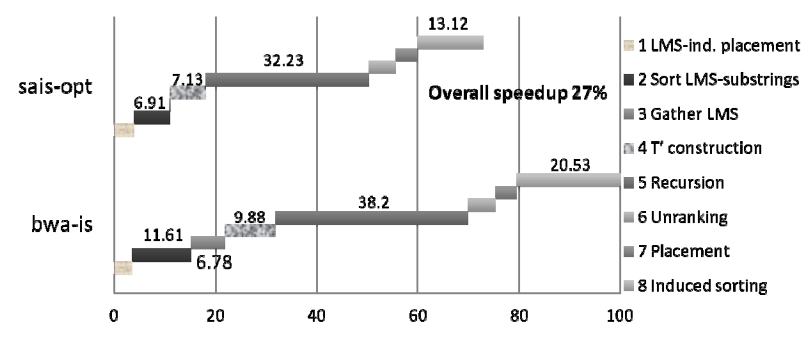

Fig.2. Average impact of each step (% of overall *bwa-is* time).

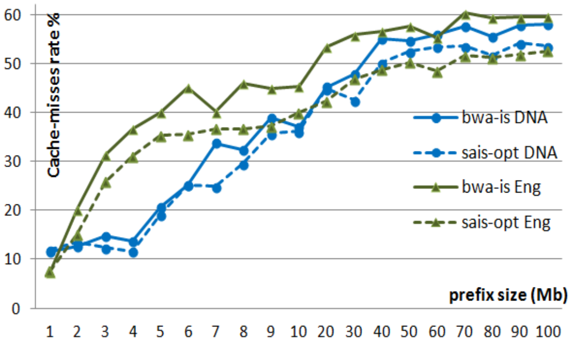

Fig.3. Rate of cache-misses for different size prefixes

After that, the gathering stage can be implemented without consulting string *T*. This modification is used on each level of recursion, which results in a 6% speedup of stage 5 (see Fig.2). In contrast, the next optimization can be applied only on the first level of the recursion, when an input string is a string over a small alphabet, e.g. ASCII codes.

The *Sorting of LMS-substrings* stage consists of: (1) inducing the order of L-indexes from the order of LMS-indexes by forward scan of SA, and (2) inducing the order of S-indexes from L-indexes by backwards scan. Each bucket in SA is naturally partitioned on L- and S-segments, which contained only L- or S-indexes respectively, because any L-suffix is smaller than any S-suffix started with the same character. For backwards scan we further partition S-segment into LMS- and non-LMS-parts. It is possible to do because the order of LMS-indexes among non-LMS-indexes is out of our interest in stage 2. Moreover, for bucket *c*, the L-segment and non-LMS-part of the S-segment are further split into subbuckets (*c*', *c*), where *c* = $t_j$ and *c*' = $t_{j\text{-}1}$ for some index *j* (Fig. 4). Recall that the position in SA for index *j*-1 (child-index) will be induced from index *j* (parent-index). This two-level partitioned bucket structure allows: (1) scanning only those parent-indexes, which induce child-indexes of S-type, i.e. for each bucket we can skip LMS-part and all (L-child, L-parent)-subbuckets; and (2) grouping LMS-indexes, preserving their order in the LMS-part. Stage 2 benefits from (1) and stage 3 – from (2). The execution time for the gathering stage on the first iteration was decreased on tested data from 6.78% to 0.03% (Fig. 2).

### 2) *Optimized ranking of LMS-substrings*

We denote $T^*_j$ as a LMS-substring that starts with an index *j* in *T*, and *rank*($T^*_j$) as a number of the *unique* LMS-substrings which are smaller than $T^*_j$. Because all LMS-indexes are stored in SA in the ascending order of the corresponding LMS-substrings, $\mathrm{rank}(T^*_{\mathrm{SA}[i]}) = \mathrm{rank}(T^*_{\mathrm{SA}[i\text{-}1]})$ if $T^*_{\mathrm{SA}[i]} = T^*_{\mathrm{SA}[i\text{-}1]}$, and otherwise $\mathrm{rank}(T^*_{\mathrm{SA}[i]}) = \mathrm{rank}(T^*_{\mathrm{SA}[i\text{-}1]}) + 1$. The naïve approach would be to compare each pair of substrings $T^*_{\mathrm{SA}[i]}$ and $T^*_{\mathrm{SA}[i\text{-}1]}$ character by character. It demands querying distant locations in a memory, which leads to numerous cache-misses. To overcome this problem *bwa-is,* first, compute the length of each LMS-substring. After that, an expensive explicit comparison takes place only for substrings with equal lengths.

After profiling the ranking stage on several inputs, some interesting statistics were revealed. On the first recursive iteration, the number of LMS-substrings $T^*_{\mathrm{SA}[i]}$ with the same length as $T^*_{\mathrm{SA}[i\text{-}1]}$ comprised about 99% of the number of all LMS-substrings, and only 1% of LMS-substrings with equal lengths were actually different. So, in at least 98% of cases, *bwa-is* explores string *T* to prove an equality of equal LMS-substrings. Another interesting observation is that for profiled examples, 75% of all LMS-substrings have length ≤ 10.

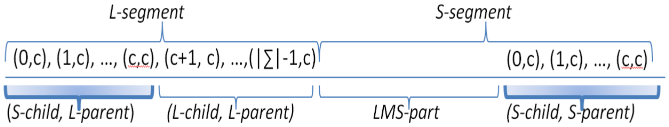

Fig.4. The structure of bucket *c* in stage 2. Only elements from highlighted parts need to be processed to induce the order of S-indexes.

We propose to compute for the LMS-substrings with length ≤ [31/([log(2|Σ|-1)]] their signatures – dense binary codes. Our target is to fit the signature into 31 bits of a 32 bits word to achieve a fast comparison and to avoid the use of extra memory. For example, for strings over Σ = {A, C, G, T, N} signatures will be built for each LMS-substring with length ≤ 10. The first bit is used to distinguish if either the signature or the length of the LMS-substring is stored. Only in the last case the explicit comparison with the previous LMS-substring may take place, otherwise strings are compared by signatures. With the described approach ranking stage (4.2) runs twice as fast and reduces the overall time of *bwa-is* by another 3%.

### 3) *Optimized Induced sorting*

L-suffix $T_i$ is called *RML-suffix* (Right Most L-suffix) if $T_{i+1}$ has type S, more precisely, $T_{i+1}$ is LMS-suffix.

*Lemma 1.* If RML-suffix $T_i$ and non-RML L-suffix $T_j$ start with the same pair of characters, i.e. $t_i$=$t_j$ and $t_{i+1}$= $t_{j+1}$, then $T_i > T_j$.

*Proof.* From the definition of RML-suffixes $T_{i+1}$ must have type S and $T_{j+1}$ must have type L. Hence, $T_{i+1} > T_{j+1}$, because $t_{i+1} = t_{j+1}$. At last, $t_i = t_j$ and $T_{i+1} > T_{j+1}$ implies that $T_i > T_j$. (Notice, a similar lemma for LMS- and S-suffixes would be incorrect.)

Let's consider stages 7 and 8 from Fig. 3 in more details.

7. Placement: *place* sorted *LMS-suffixes* at the ends of the corresponding buckets, preserving their order.

8(a). *Place* L-suffixes in the order induced from the order of LMS-suffixes;

8(b). *Place* S-suffixes, including *LMS-suffixes*, in the order induced from the order of L-suffixes.

Here we observe the initial placement of LMS-suffixes on the stage 7 and the secondary placement to new positions of the same suffixes on the step 8b.

We have modified stage 7, so instead of the LMS-suffix $T_j$ the RML-suffix $T_{j\text{-}1}$ is placed in SA. In other words we induce the order of RML-suffixes "on-the-fly" without expensive intermediate writing and successive reading of LMS-indexes to and from SA. To make it possible each bucket *c* must be partitioned into subbuckets (c, c'), where c = $t_j$ and c' = $t_{j+1}$ for some index *j*, and RML-suffixes, according to Lemma 1, have to be placed at the ends of the corresponding subbuckets. This partitioning requires us to keep |Σ|*|Σ-1|/2 pointers. To conform to the memory limitations of *bwa-is* we apply this optimization only on the first recursive level achieving 25-30% speedup for stage 8, which brings about 7% speedup of the entire program.

Table 1 summarizes required types of memory access operations for each stage of *bwa-is* and *sais-opt* on the first recursive iteration. For simplicity "reading" is used for two types of operations: (1) read and (2) read and update the same location. "Consecutive" reading or writing stands for operations that applied to long consecutive intervals in memory in opposite to "irregular" or "random" access. Although the consecutive reading (writing) is faster than irregular, it is still very time consuming, because it spans gigabytes of DRAM without significant data reuse in a cache memory. The diagram on Fig.3 demonstrates drop in the rate of cache-misses when run *sais-opt* compare to *bwa-is*.

TABLE I. SUMMARY OF OPERATIONS ON EACH STAGE FROM FIG. 1 FOR BWA-IS AND SAIS-OPT

| stage | *cr*T[a] | *cr*SA | *ir*T | *ir*SA | *cw*SA | *iw*SA |
|---|---|---|---|---|---|---|
| 1 | *b*/*o* | | | | | *b*/*o* |
| 2 | | *b*/*o** | *b*/*o* | | | *b*/*o* |
| 3 | | *b*/*o** | *b* | | *b*/*o* | |
| 4.1 | *b*/*o* | | | | | *b*/*o* |
| 4.2 | | *b*/*o* | *b*/*o** | *b*/*o* | | |
| 4.3 | | *b*/*o* | | | *b*/*o* | |
| 6 | *b*/*o* | *b*/*o* | | *b*/*o* | *b*/*o* | |
| 7 | | *b*/*o* | *b*/*o* | | *b* | *o* |
| 8 | | *b*/*o** | *b*/*o** | | | *b*/*o** |

[a] *cr*(*w*)T(SA) – consecutive read (write) from/to T(SA) , *ir*(*w*)T(SA) – irregular read (write) from/to T(SA); *b* – operations present in *bwa-is*, *o* - operations present in *sais-opt*; * – number of operations of this type is reduced in *sais-opt* compare to *bwa-is*

### 4) *sais-opt32*

The sizes of many mammalian genomes, including human genome, fall into an interval of $2^{31}$ to $2^{32}$ base pairs [17]. As well as *bwa-is* our program *sais-opt* is operational only for strings of size < $2^{31}$, so any index *j* for a string of this size can be stored in 31 bits, while one bit of 32 bits word is reserved for the type of child-index. The use of 40 bits structure for this type of strings would require 5*n* bytes instead of 4*n* to store SA and it also would slowdown the entire application. Instead, we have designed 32-bit program *sais-opt32* that exploits all 32 bits to store the indexes. Because of *sais-opt32* can't benefit from the preliminary type identification it experiences about 5% slowdown compare to *sais-opt* when applied to the acceptable for *sais-opt* inputs, but in return it allows us to build SA for mammalian size genomes within the same memory frames.

## V. RESULTS AND DISCUSSION

We employed proposed in section IV optimizations in our C programs *sais-opt* and *sais-opt32*, which have to be chosen, depending on the input size. Six other SACA applications (table. 2) were used in our experimental performance evaluation, including three super-linear time algorithms: *libdivsufsort* as the golden standard, *radixSA* [14] as the most recently published, *bpr2* [13], which is claimed in [14] to be the fastest before *radixSA*; and three different modifications of SA-IS algorithm: *saca-k* [16], *bwa-is* and *sais-lite*.

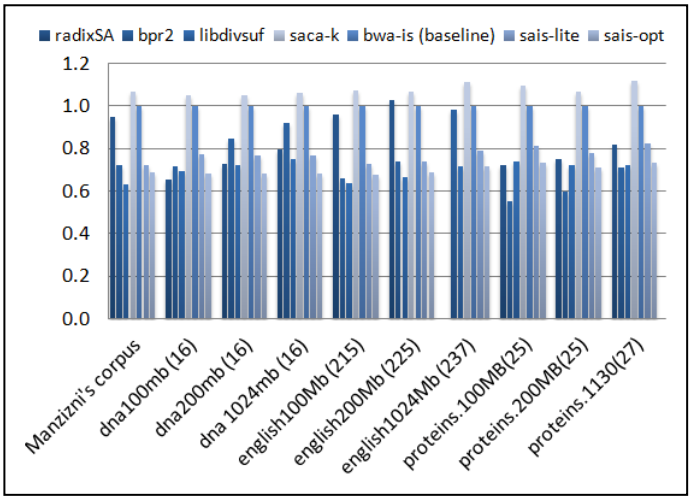

Fig. 5. Performance comparison of the leading SACAs on the set of standard test data: Manzini's and Pizza&Chili corpora.

TABLE II. SACA APPLICATIONS USED IN EXPERIMENTS

| *program* | *source* |
|---|---|
| *radixSA* | http://www.engr.uconn.edu/~man09004/radixSA.zip |
| *bpr2 (2.0.0)* | http://bibiserv.techfak.uni-bielefeld.de/download/tools/bpr.html |
| *libdivsufsort (2.0.1)* | http://code.google.com/p/libdivsufsort/ |
| *saca-k* | https://code.google.com/p/ge-nong/ |
| *bwa-is* | Code extracted from *is.c* file of *bwa-0.7.5a*<br>http://sourceforge.net/projects/bio-bwa/files/ |
| *sais-lite (2.4.1)* | https://sites.google.com/site/yuta256/sais/ |

We do not consider any other linear-time SACAs except of SA-IS modifications, as previous studies proved the priority of SA-IS over them [9]. According to another recently published in [18] evaluation, *sais-lite* shows the best performance results over several other programs, including *saca-k*.

All experiments were held on a single core and one (closest to the core) memory node, if acceptable, of NUMA machine with 2 AMD Opteron Processors (6272) containing 16 cores (2100 MHz with 2MB L1-cache) and two 16G of RAM memory nodes each. For *radixSA* and *bpr2* additional memory nodes were used when necessary. The programs *bpr2, radixSA, libdivsufsort, saca-k* and *sais-lite* were compiled using the default *makefile* provided in their source packages, and *bwa-is* and *sais-opt* were compiled by *gcc* compiler with "-O3" optimization option. The average time of three runs for each input was used in the evaluation.

For our first test we chose inputs with moderate sizes from the standard data sets for SACAs evaluation: Manzini's Large Corpus (10 strings of sizes from 34MB to 116 MB) that can be downloaded at http://code.google.com/libdivsufsort, and DNA, proteins and English texts from Pizza&Chili Corpus at http://pizzachili.dcc.uchile.cl/texts.html. The second dataset includes large-sized DNA strings. All the strings[1], except for the human genome, were obtained from "Soft-masked" assembly sequences, downloaded from UCSC Genome Bioinformatics Site at http://genome.ucsc.edu. The files were stripped of all characters but {A, C, G, T, N} and normalized to upper-case. Already preprocessed in the similar way, sequence for the human genome was downloaded from http://tbingmann.de/2012/esais/ [19]. For some tests only 2000MB prefixes were considered to satisfy a 31 bits limitation for index representation.

On the diagrams depicted on Fig. 5 and Fig. 6, time is normalized with respect to the running time of *bwa-is*. For all moderate-sized strings from Manzini's Corpus the leadership belongs to *libdivsufsort*, with the exception of the shortest string chr22, for which the best result was shown by *bpr2*. But for the longer strings from the Pizza&Chili corpus *sais-opt* outperforms *libdivsufsort* for DNA strings and has comparable results for English texts and proteins. Although for protein sequences the best performance was delivered by *bpr2*, it's worth to mention, that this algorithm consumes almost three times more memory than *sais-opt* or *libdivsufsort*. Fig.6 presents comparison of performance time for large-sized DNA (856Mb to 2000Mb). For these inputs *sais-opt* shows 27-30% speedup over *bwa-is* and excels two best SACA programs

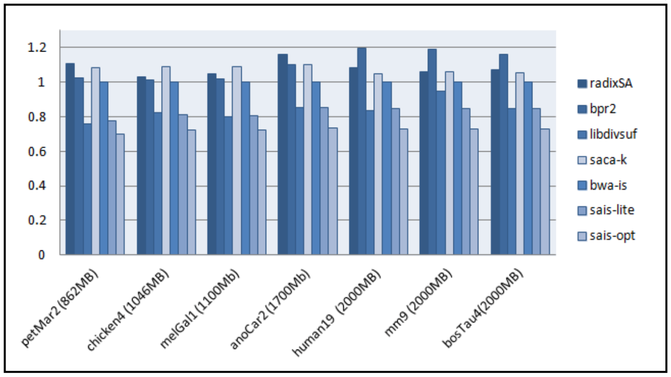

Fig.6. Performance comparison of different SACAs for large-sized DNA strings (|Σ|=5).

*libdivsufsort* and *sais-lite*. Fig. 7 displays time and memory consumptions required to build SA for tested DNA sequences with length > $2^{31}$. We were able to run successfully on these data only *bpr2*, *saca-k*, *sais-opt32* and *bwa-is32* – our 32 bits modification of *bwa-is*. For all tested inputs *sais-opt* requires less time and uses almost the same amount of memory as *saca-k*, with the difference less than one megabyte.

Our library also contains functions with the modified last stage of the *sais-opt* algorithm, providing an option to build BWT instead of suffix array[2].

## VI. Conclusion and Future Work

SA-IS algorithm exhaustively employs the fact that sorted strings are the suffixes of one string. This differentiates SA-IS from the algorithms which use regular string sorting methods and provides advantages when applying SA-IS to large-sized data. At the same time, the memory-bounded characteristic of SA-IS impairs its performance in practice and only proper implementation, co-designed with the underlying architectures, is able to mitigate this problem. The introduced *sais-opt* algorithm reduces the number of memory-access operations, resulting in a better performance compared to the previous implementations of SA-IS. To take advantage of modern multi- and many-core CPU architectures we plan to undertake some efforts to parallelize SA-IS algorithm. This task is quite challenging due to severe data dependencies and recursive nature of the algorithm. Our preliminary work shows that the induced sorting, although it seems to be inherently sequential, has some hidden parallelism.

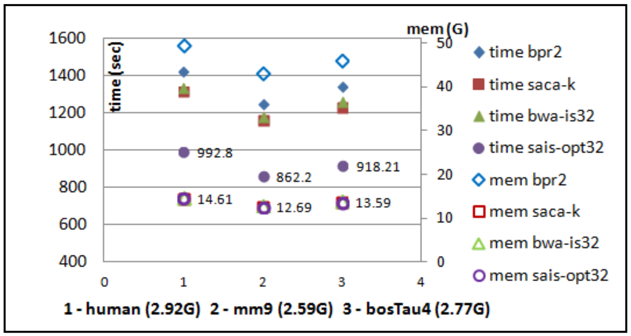

Fig.7. Performance time (sec) for DNA strings with length > $2^{31}$

## References

[1] Li H. and Durbin R.. "Fast and Accurate Short Read Alignment with Burrows-Wheeler Transform." *Bioinformatics (Oxford, England)* 25, no. 14 (July 15, 2009): 1754–1760.

[2] Vyverman Michaël, Bernard De Baets, Veerle Fack, and Peter Dawyndt. "Prospects and Limitations of Full-Text Index Structures in Genome Analysis." *Nucleic Acids Research*, May 13, 2012, gks408.

[3] Lippert RA. "Space-Efficient Whole Genome Comparisons with Burrows-Wheeler Transforms." *Journal of Computational Biology: A Journal of Computational Molecular Cell Biology* 12, no. 4 (May 2005): 407–415.

[4] Kim Jaebum, Denis M. Larkin, Qingle Cai, Asan, Yongfen Zhang, Ri-Li Ge, et al. *"Reference-Assisted Chromosome Assembly." Proceedings of the National Academy of Sciences*, January 10, 2013.

[5] Jared T. Simpson and Richard Durbin, "Efficient *de novo assembly of large genomes using compres*sed data structures." *Genome Research, 2012.* □

[6] Liu Xing, Pushkar R. Pande, Henning Meyerhenke, and David A. Bader. "PASQUAL: Parallel Techniques for Next Generation Genome Sequence Assembly." *IEEE Transactions on Parallel and Distributed Systems* 24, no. 5 (May 2013): 977–986.

[7] Schatz Michael, Arthur L Delcher, and Steven L Salzberg. "Assembly of Large Genomes Using Second-Generation Sequencing." *Genome Research* 20, no. 9 (September 2010): 1165–1173.

[8] Hon Wing-Kai, Tak-Wah Lam, Kunihiko Sadakane, Wing-Kin Sung, and Siu-Ming Yiu. "A Space and Time Efficient Algorithm for Constructing Compressed Suffix Arrays." *Algorithmica* 48, no. 1 (May 1, 2007): 23–36.

[9] Nong Ge, Sen Zhang, and Wai Hong Chan. "Two Efficient Algorithms for Linear Time Suffix Array Construction." *IEEE Transactions on Computers* 60, no. 10 (October 2011): 1471–1484.

[10] Puglisi Simon J., W. F. Smyth, and Andrew H. Turpin. "A Taxonomy of Suffix Array Construction Algorithms." *ACM Comput. Surv.* 39, no. 2 (July 2007).

[11] Kärkkäinen J., P. Sanders, and S. Burkhardt. "Linear Work Suffix Array Construction." *J. ACM* 53, no. 6 (November 2006): 918–936.

[12] Ko Pang and Srinivas Aluru. "Space Efficient Linear Time Construction of Suffix Arrays." In *Journal of Discrete Algorithms*, 200–210. Springer, 2003.

[13] K.-B. Schurmann, J. Stoye. "An incomplex algorithm for fast suffix array construction." Softw: Pract. Exper. 37 (3) (2007) 309 -329

[14] Rajasekaran Sanguthevar and Marius Nicolae. "An Elegant Algorithm for the Construction of Suffix Arrays." *arXiv:1307.1417 [cs]*, (July 2013). http://arxiv.org/abs/1307.1417.

[15] Rajesh Yelchuri, Nagamalleswara Rao.N. "Space Efficient Suffix Array Construction using Induced Sorting LMS Substrings." International Journal of Information Sciences and Techniques. Vol.3, No.4, July 2013

[16] Nong Ge. "Practical Linear-Time O(1)-Workspace Suffix Sorting for Constant Alphabets." *ACM Trans. Inf. Syst.* 31, No. 3, August 2013

[17] Redi C A, and E Capanna. "Genome Size Evolution: Sizing Mammalian Genomes." Cytogenetic and Genome Research 137, no. 2–4 (2012): 97–112.

[18] Shrestha Anish Man Singh, Martin C. Frith, and Paul Horton. "A Bioinformatician's Guide to the Forefront of Suffix Array Construction Algorithms." Briefings in Bioinformatics, January 10, 2014.

[19] Bingmann T., Fischer J, Osipov V., Inducing Suffix and LCP Arrays in External Memory Workshop on Algorithm Engineering and Experiments (ALENEX 2013): 88-102

[1] petMar2 (862Mb) - Lamprey (WUGSC 7.0), 2010; chicken4 (1046Mb) - Chicken (ICGSC Gallus_gallus-4.0), 2011; melGal1(1100Mb) - TGC Turkey_2.01, 2009; anoCar2 (1700Mb) - Lizard (Broad AnoCar2.0), 2010; mm9 (2651Mb) - mouse (NCBI3); bosTau4 (2838Mb) - Cow (Baylor 4.0); human (2992Mb) - human-hg19.dna.3137161264

[2] The corresponding software is undergoing intellectual property review at Virginia Tech.