Design of an Efficient Out-of-Core Read Alignment Algorithm

Arun S. Konagurthu[1,2,⋆,⋆⋆], Lloyd Allison[1,⋆⋆], Thomas Conway[1,2], Bryan Beresford-Smith[1,2], and Justin Zobel[2,1]

[1] National ICT Australia (NICTA) Victoria Research Laboratory,
Department of Electronics and Electrical Engineering
The University of Melbourne, Parkville, Victoria 3010 Australia
[2] Department of Computer Science and Software Engineering,
The University of Melbourne, Parkville, Victoria 3010 Australia
{firstname.lastname}@nicta.com.au

Abstract. New genome sequencing technologies are poised to enter the sequencing landscape with significantly higher throughput of read data produced at unprecedented speeds and lower costs per run. However, current *in-memory* methods to align a set of reads to one or more reference genomes are ill-equipped to handle the expected growth of read-throughput from newer technologies.

This paper reports the design of a new out-of-core read mapping algorithm, `Syzygy`, which can scale to large volumes of read and genome data. The algorithm is designed to run in a constant, user-stipulated amount of main memory – small enough to fit on standard desktops – irrespective of the sizes of read and genome data. `Syzygy` achieves a superior spatial locality-of-reference that allows all large data structures used in the algorithm to be maintained on disk. We compare our prototype implementation with several popular read alignment programs. Our results demonstrate clearly that `Syzygy` can scale to very large read volumes while using only a fraction of memory in comparison, without sacrificing performance.

1 Introduction

The landmark publications of Margulies *et al.* [1] and Shendure *et al.* [2] in 2005 heralded a new era of non-Sanger based, massively parallel genome sequencing technologies. Today's major commercial *next-generation sequencing* (NGS) systems include Roche's (454) Genome Sequencer FLX, Illumina-Solexa's Genome Analyzer (GA) II, and Applied Biosystem's SOLiD. The volume of data generated from these new sequencers is already staggering. (For example, Illumina's latest GA IIe sequencer produces about 1.75×10^9 bases of read data in a day's run.) More recently, several new sequencing systems, such as Helicos' Genetic

⋆ Corresponding author.
⋆⋆ These authors contributed equally to this work.

Analysis system, Pacific Biosciences' Single Molecule Real Time (SMRT) system, Oxford Nanopores' Nanopore sequencer, and Visigen's Genetic sequencer, have been announced promising higher read throughput at faster speeds and significantly reduced costs per run. Some of these technologies are already in business.

Mapping (or aligning) a set of reads to a reference genome is a fundamental task in genome resequencing studies. Massive volumes of read data (growing faster than Moore's law) and very large genome sizes make the read mapping problem computationally very challenging. Neither the classical methods for pattern matching on strings [3–5] nor the methods from traditional sequence bioinformatics [6, 7] can cope with the large volumes of data from modern sequencers.

Since 2007, several methods catering specifically to NGS were published [8–19].[1] These methods can be broadly classified into four groups:

1. Methods which rely on hashing the read set (*e.g.* see [8–12]);
2. Methods which rely on hashing the reference (*e.g.* see [13–15]);
3. Methods based on advances in Stringology (*e.g.* see [16–18]);
4. A method [19] based on a *sort-and-join* approach.

Methods in categories 1 and 2 maintain a large hash index in main memory when performing read alignment. The growth of data (read or genome, depending on which set is maintained as a hash index) translates to increasing demands on main memory for these methods. Methods in category 3, especially those that use the Burrows-Wheeler index [20] of the reference sequence are comparatively memory efficient when aligning reads to a single genome [16, 17]. However if reads were to be mapped on more than one large genome (for example, multiple human genomes simultaneously), even these methods begin to have impractical memory demands. Slider [19] is currently a solitary method in category 4 which relies on a simple sort-and-join strategy. The program is slow and requires both a large amount of memory as well as disk space.

A common problem with the current programs is that they do *not* scale elegantly to handle very large data volumes due to impractical memory requirements or very long run times (in some cases, both). Moreover, the random nature of data accesses to the index structures maintained by the methods in the first three categories pose a major hurdle for their implementation out of core.

This paper describes the design of a new method, `Syzygy`, to efficiently align massive numbers of reads simultaneously against multiple genomes. The design allows the program to run in a fixed, user-stipulated amount of memory, small enough to be deployed even on standard computers. Broadly, our method is based on a *sort-and-join* strategy similar to the one used by Slider [19]. However the details of our algorithm and its implementation is radically different, especially in the way it handles the *approximate* read mapping problem. `Syzygy` reorganizes the read mapping problem to achieve a superior *spatial locality* of accesses to various data structures maintained by the method. This reorganization results

[1] A comprehensive list of NGS read mapping tools is maintained by Heng Li at `http://lh3lh3.users.sourceforge.net/NGSalign.shtml`

in accesses to all data structures being predominantly *linear*, facilitating an *out-of-core* implementation among other performance optimizations; all large data structures in the algorithm are maintained on disk, requiring only a small *in-memory* working set to proceed with the alignment.

2 Algorithm

2.1 Definitions

DNA sequence: A DNA sequence of length n is a string $S = (s_1 \cdots s_n)$ containing n 'bases', where each base is from the alphabet of DNA nucleotides, $\aleph = \{A, C, G, T\}$.
Reference genome set: Let $\mathcal{G} = \{\mathcal{G}^1 \cdots \mathcal{G}^N\}$ be a set containing N reference genomes where any genome $\mathcal{G}^i = (g_1^i \cdots g_{n_i}^i)$ (assume) is a DNA sequence containing n_i bases.
Read set: Let $\mathcal{R} = \{r^1 \cdots r^m\}$ be a set containing m sequence reads, where any read $r^i = (r_1^i \cdots r_L^i) \in \mathcal{R}$ of length L is a short DNA sequence.
k-mer: Given any sequence $S = (s_1 \cdots s_l)$ of length l and a constant $k \leq l$, a k-mer of S defines another sequence $\mathcal{K} = (s_i \cdots s_{i+k-1}), 1 \leq i \leq l - k + 1$ which is a substring of S.
Reverse complement: A reverse complement of a DNA sequence S, denoted by $\overline{S}$, is a sequence of bases which reverses S and replaces each base with its Watson-Crick conjugate (A with T, G with C, and vice versa).
Key: A `key(`S`)` denotes an integral hash value of a sequence S using some key-generation function which transforms strings uniquely to integers. A straight-forward key generation function of DNA sequences over the alphabet $\aleph$ is the integral value as a result of representing the sequence using a 2 bits-per-base encoding. (For example, $\{00, 01, 10, 11\}$ for $\{A, C, G, T\}$.)

2.2 The Basic Sort-and-Join Method

We build our exposition by introducing first the basic sort-and-join method to align reads simultaneously to a set of genomes. In the basic approach we use the ideal scenario where reads R are matched *exactly* (that is, without errors) with the genome set $\mathcal{G}$. (To make the exposition clearer, in the entire paper we assume that all reads in the set $\mathcal{R}$ are of a fixed length L. We note, however, that it is straightforward to generalize our algorithm to variable length reads.) The basic algorithm involves three simple steps:

Step 1: Reference list generation. A reference list defines a sorted list of records G corresponding to every L-mer in $\mathcal{G}$. Each L-mer, $\mathcal{L} = (g_j^i \cdots g_{j+L-1}^i), 1 \leq i \leq N, 1 \leq j \leq n_i - L + 1$, contributes the fields (h, p) to form a record in G, where $h =$ `key(`$\mathcal{L}$`)` is the key of $\mathcal{L}$, and $p = [i, j]$ is the positional coordinate (sequence number and offset in sequence) of $\mathcal{L}$ in $\mathcal{G}$. The records in list G are sorted on the field h.

Step 2: Read list generation. For each read $r = (r_1 \cdots r_L) \in \mathcal{R}$, construct a read list $R = \{(h, r)\}$ where $h =$`key(r)`. R is also sorted on the field h.
Step 3: Join list generation. A join list $J = G \bowtie R$ is derived by joining $G = \{(h, p)\}$ and $R = \{(h, r)\}$ on the field h, resulting in $J = \{(p, r)\}$. Each record in this list gives a position p of the *exact* occurrence of a read r.

Observe the predominantly linear nature of data accesses in this method. In Steps 1 and 2, genome and reference data are read sequentially while creating G and R respectively. Again, the generation of the join list J requires all sequential accesses through the sorted lists G and R, giving the list of matches of reads against the genome(s). We note here that sorting of small keys (of fixed size) is *near-linear* [21, 22].

Below, we use the framework of the basic sort-and-join strategy to address the problem of approximate matching of reads to multiple genomes.

2.3 Sort-and-Join Method for Mapping with Errors

In practice, a large number of reads will not map *exactly* to reference sequence(s) due to the presence of sequencing errors in the reads as well as other natural genomic variations between the sample and the reference draft assembly. Therefore it becomes necessary to map the reads to the reference genomes allowing a certain number of errors or mismatches. A common strategy to handle approximate matches is based on a lossless k-mer filtering technique. This technique relies on the observation that two sequences of length L which are at a Hamming distance of *at most* δ should share *at least* one k-mer of size $k = \lfloor \frac{L}{\delta+1} \rfloor$. Indeed this observation generalizes to Levenshtein distances between two strings. Our method for approximate matching uses this observation. Below we describe the extension of the basic sort-and-join strategy to map the reads in the presence of errors under a threshold of Hamming distance δ:

Step 1: Reference list generation. Build a sorted reference list G such that the tuples correspond to k-mers (instead of L-mers, previously) such that $k = \lfloor \frac{L}{(\delta+1)} \rfloor$.
Step 2: Read list generation. Each read is partitioned into fixed size (non-overlapping) tiles of length $k = \lfloor \frac{L}{\delta+1} \rfloor$. The pigeonhole principle suggests that if a read matches under a threshold of δ at some position in the reference, then there must be *at least* one of the $\delta + 1$ read tiles that must match *exactly* to a corresponding k-mer in the reference. Note that due to the chemistry involved in the sequencing process each read should also be examined for a match against the reference using its reverse complement.

Each read, therefore, contributes $2 \times (\delta + 1)$ non-overlapping (tiled) k-mers (in both forward and reverse complement directions), where $k = \lfloor \frac{L}{(\delta+1)} \rfloor$. Specifically, a read $r^i = (r^i_1 \cdots r^i_L)$ and its corresponding reverse complement $\overline{r^i} = (\overline{r^i}_1 \cdots \overline{r^i}_L)$ contributes to these non-overlapping k-mer tiles:$\{(r^i_1 \cdots r^i_k),\ (r^i_{k+1} \cdots r^i_{2k}),\ \cdots,\ (r^i_{\delta\times k+1} \cdots r^i_{(\delta+1)\times k})\}$ and $\Big\{(\overline{r^i}_1 \cdots \overline{r^i}_k),\ (\overline{r^i}_{k+1} \cdots \overline{r^i}_{2k}),\ \cdots,\ (\overline{r^i}_{\delta\times k+1} \cdots \overline{r^i}_{(\delta+1)\times k})\Big\}$.

Hence each tile of a read now contributes to the following fields: (h, r, o, s), where h is the key of a tile, r the read from which it came, o the offset of the tile from the start of the read, and s the 'sense' (forward or reverse complement) of the tile. For example, the k-mer tile $\mathcal{K} = ({r^i}_{k+1} \cdots {r^i}_{2k})$ and its corresponding reverse complement $\overline{\mathcal{K}} = (\overline{r^i}_{k+1} \cdots \overline{r^i}_{2k})$ contribute to R: $\big(\texttt{key}(\mathcal{K}), r^i, (k+1), \rightarrow\big)$, and $\Big(\texttt{key}(\overline{\mathcal{K}}), \overline{r^i}, (k+1), \leftarrow\Big)$, where '$\rightarrow$' and '$\leftarrow$' indicate matching in the *sense* (forward) and *antisense* (reverse complement) directions of the k-mer tiles respectively. Finally, the list $R \equiv \{(h, r, o, s)\}$ is sorted on the field h.

Step 3: Join list generation. Join the sorted lists $G = (h, p)$ and $R = (h, r, o, s)$, to give a new list $J = G \bowtie R \equiv \{(p', r, s)\}$, where $p' = p - o$ is the adjusted positional coordinate on the genome set which allows different tiles of the same read to coalesce back together. (See Step 4.) For a given read if more than one tile matches exactly at some position in the genome, the adjustment p' ensures that they point to the same starting position on the reference. This provides the necessary efficiency in the post-processing step below.

Step 4: Verification and post-process. The list J is sorted on the fields p', r, and s in that order. Let a *context* in J define a set of items in J that have the same (adjusted) position, read, and sense tuples. Sorting J on (p', r, s) ensures that tiles which share the same context will group together. Each context, containing one or more tiles, identifies a unique position $p' \in \mathcal{G}$, read r, and the directionality of the match s. While traversing linearly in the newly sorted list J, for each unique context, just one tile is enough to *verify* the Hamming distance of the L-mer in $\mathcal{G}$ starting at position p' with respect to the read r, in the direction specified by s. The result of the verification is a list of mappings of the set $\mathcal{R}$ on $\mathcal{G}$ under the approximate Hamming distance threshold δ.

We note that the extension of the sort-and-join strategy to approximate matching still retains its sequential data access characteristic which is important for any out-of-core implementation. In step 4, each Hamming distance verification of the join record requires the extraction of a read-length sized substring from the reference (string) data set which is then compared with the corresponding read available in the join record. Since the join list is sorted primarily on the (adjusted) position in the reference, accesses to the reference string(s) are sequential, giving the crucial advantage of spatial locality of reference.

3 Implementation of Syzygy

3.1 Encoding, Key Generation and Other Bitwise Tricks

A nucleotide sequence from the alphabet containing four bases $\{A, T, G, C\}$ is *packed* into an array of unsigned 64-bit integer data types, where each base is represented using a *2 bits-per-base* encoding. Specifically, Syzygy uses the $\{00, 01, 10, 11\}$ binary encoding for $\{A, C, G, T\}$ respectively. Each unsigned 64-bit word (or, plainly, *word*) can encode information of up to 32 bases of a sequence. For example, a DNA sequence of length 100 is packed into an array of 4 encoded words. (This requires an implicit convention to align strings to a consistent word boundary, necessary to decode strings whose length are not an integral multiple of 32. Assuming a *little-endian* architecture, in `Syzygy`, the start of strings are aligned to the *most-significant* word boundary). In the implementation of `Syzygy`, the key of a k-mer is simply the integer defined by its encoded word(s).

In practice, both the genome and read sequences come from an *extended* alphabet to account for ambiguous or unknown bases. On the genome side, we store the strings in the encoded form by converting each ambiguous base to a random unambiguous base in $\aleph$, while ignoring the contributions to the reference list G by the k-mers in the regions containing ambiguous bases. For the read set we ignore all sequences containing more than two ambiguous bases.

This encoding has some convenient advantages. Hamming distance and reverse complement generation can be performed cheaply using a few bitwise operations. The code for determining Hamming distance is shown in Fig. 1. (There are several bitwise tricks to facilitate fast population count using `popcount()` in Fig. 1. See [23] for a comprehensive summary on accelerated population counting. Alternatively, a rapid way to perform this operation would be to invoke a `POPCNT` instruction which comes as a part of the instruction set on most modern microprocessors.) Coincidentally, the specific encoding used in `Syzygy` also allows a fast computation of reverse complements of DNA sequencing which relies on the fact that, in our encoding, Watson-Crick conjugates have their bits flipped. (See Fig. 2 below.)

```
#define MASK 0x5555555555555555UL

uint8_t hammingDistance(
 uint64_t *w1,
 uint64_t *w2,
 size_t n
) {
   uint64_t tmp ;
   uint8_t  d = 0 ;

   for( size_t i = 0 ; i <  n ; i++ ) {
     /* XOR ith words */
     tmp = w1[i] ^ w2[i];

     /* convert to a popcount problem */
     tmp = (tmp & MASK) |
                (tmp>>1 & MASK) ;

     /* count set bits in tmp */
     d += popcount( tmp ) ;
   }
   return d ;
}
```

Fig. 1. Code for Hamming distance computation of two strings packed into an array of n words $w1$ and $w2$

3.2 Main List Data Structures

Reference list: Recall, the reference list G is a list of records of the form $\{(h, p)\}$. The field h, storing the key (the integer encoding) of a k-mer, can be denoted using an array of one or more words depending on the size of the k-mer. Field p on the other hand can be simply stored in a single word. In `Syzygy`, however, the reference list is generated using 32-mers from the reference genome set, requiring just one word each to store the fields h and p. (See also the special construction of the reference list, explained in section 3.3, to handle *'blowups'* in the join.) In addition, we observe that a 32-mer based reference list containing (h, p) tuples subsumes all reference lists corresponding to any ($k < 32$)-mers. This holds primarily due to the encoding `Syzygy` uses and the nature of its key generation function. A fully sorted 32-mer keys implies the sorted order of any of its prefixes. For example, a

16-mer reference list can be derived from a 32-mer list by trivially masking out the encoded bits corresponding to the trailing 16 bases. (Note however that some ($k < 32$)-mers in the end of each genome – or any discontinuous region – will be lost when sliding along the genome set with a window of size 32. But this is a minor issue which can be trivially remedied.) Using the above observation, a 32-mer reference list can be *preprocessed* and used for mapping under variable tile sizes, calculated based on the hamming distance threshold and the read length. This is especially meaningful because often it is the read set which changes while the set of reference genomes remains mostly static.

Read list: Reference list R is a list of records of the form $\{(h, r, o, s)\}$. To conserve space, we do not store the field h but, instead, it is calculated on the fly using the remaining fields: r, o and s. Field r is an array of words representing the encoded read. In addition to the encoded read data, it makes practical sense to carry along a read identifier. Read identifier (a number) and fields o and s are packed together into one word. We will call this word, *read metadata*.

Syzygy computes an appropriate tile size depending on the read length L and the parameters for approximate matching δ as $\min\{\lfloor L/(\delta+1)\rfloor, 32\}$.

```
#define MASK1 0x3333333333333333UL
#define MASK2 0x0F0F0F0F0F0F0F0FUL
#define MASK3 0x00FF00FF00FF00FFUL
#define MASK4 0x0000FFFF0000FFFFUL

void reverseComplement(
 uint64_t *w,
 size_t n
) {
   uint64_t tmp ;

   /* First, reverse complement one word at a time */
   for( size_t i = 0 ; i < n ; i++ ) {
     /* A base complement trick on whole word */
     w[i] = ~w[i] ;

     /* Reverse base (NOT bits) order in each word*/
     w[i] = ((w[i]>>2)&MASK1) | ((w[i]&MASK1)<<2) ;
     w[i] = ((w[i]>>4)&MASK2) | ((w[i]&MASK2)<<4) ;
     w[i] = ((w[i]>>8)&MASK3) | ((w[i]&MASK3)<<8) ;
     w[i] = ((w[i]>>16)&MASK4) | ((w[i]&MASK4)<<16) ;
     w[i] = ((w[i]>>32)) | (w[i]<<32) ;
   }

   /* Next, reverse order of words in array  */
   for( size_t i = 0, j = n-1 ; i < j ; i++, j-- ) {
     tmp  = w[i] ; w[i] = w[j] ; w[j] = tmp ;
   }
  /* additional shift of bases across words may be
     needed to align to consistent word boundary */
}
```

Fig. 2. Code of reverse complement computation of a sequence w encoded into an array of n words

Join list: The resultant join list J consists of, for each item in the list, a word corresponding to a k-mer's (adjusted) position in the genome set, a word of the read metadata and remaining words corresponding to the encoded read.

Memory and disk usage: Syzygy runs within the user-defined amount of memory(≥ 32 MB). The program operates under the stipulated memory limit by maintaining all main data structures on disk rather than in memory. Each of the large data structures in the algorithm, G, R, and J is simply represented as one large linear array of words.

3.3 Managing "Blowups" in the Join List

A major challenge in this approach is to manage the size of the join list J. Consider a range of records in $R = \{(h, r, o, s)\}$ which share the same key h with another range of records in $G = \{(h, p)\}$. During the join operation, the join list will be populated with the *product*—Cartesian product of two sets containing items from the two ranges—of the items in these two ranges. For preponderant k-mers (such as poly-As in the Human genome which occur a very large number of times), this product would be staggering causing an unmanageable "blowup" in the size of the join.

`Syzygy` uses the following strategies to handle blowups. The program initially computes and stores (in memory) all k-mers (for a given tile size in the run) and their corresponding preponderances when they exceed a predefined threshold $(100, 000)$. Now, when generating the read list, tiles which can potentially cause a blowup are excluded from the read list at the time of its generation. A read is blacklisted (and written out to a blacklist file) if one or more of its tiles cause a blowup. For a given read it is however possible that only a few tiles are excluded while the remainder enter the read list, in which case the program does *not* guarantee to find all matches under the chosen distance parameters for that read. The program generates a 'greylist' of reads to inform the user of such cases.

We observe that the join list usually contains many *false-positives* which are removed at a later verification stage (in step 4). A vast portion of records in the join can be filtered out using a fast *early-verification* step. Recall (from section 3.1) that each record in the reference list has 32 bases. For a given tile size T, the join is performed by masking out unrelated $32 - T$ trailing bases. For a tile size of (say) 20, the bits corresponding to the 12-mer suffixes (that is, 24 bits) in all 32-mer keys from G are masked out. Since we are carrying in R the read data corresponding to all k-mer tiles, an early verification can be performed during the join on the remaining $32 - k$ bases between the read and the reference. If a tile fails the chosen distance threshold so would the read it belongs to. Hence such tiles are barred from entering into the join list.

While this procedure prunes the number of false positives drastically, it is possible, under some parameters, that the last tiles of the reads do not benefit from the early-verification. Take for example a read of length 50 mapped under an Hamming distance of 2. Each read will contain 3 tiles of length 16. While the first two tiles can be verified on 32-bases, the last tile can only be verified on the remaining two bases(that is 49th and 50th base in the read). Since we are running under a Hamming distance of 2 and there are only two additional bases to check, it is easy to see that all the last tiles of the read in such a run will pass unfiltered into the join list.

An alternative construction of Reference list used in `Syzygy` to aid early-verification of all tiles: A modification in the reference list generation will guarantee even the last tiles to benefit from early-verification. Recall that the reference list is constructed on 32-mers (and their positions) lexicographically sorted on the entire 32-mer keys. Modify the sort procedure by sorting the keys *only* on their 28-mer suffixes, instead of the complete 32-mers. To illustrate why this helps, we use the example in the above paragraph. The first 16-mer tile can now be verified on 28-bases (of which we know that 16-bases match exactly).

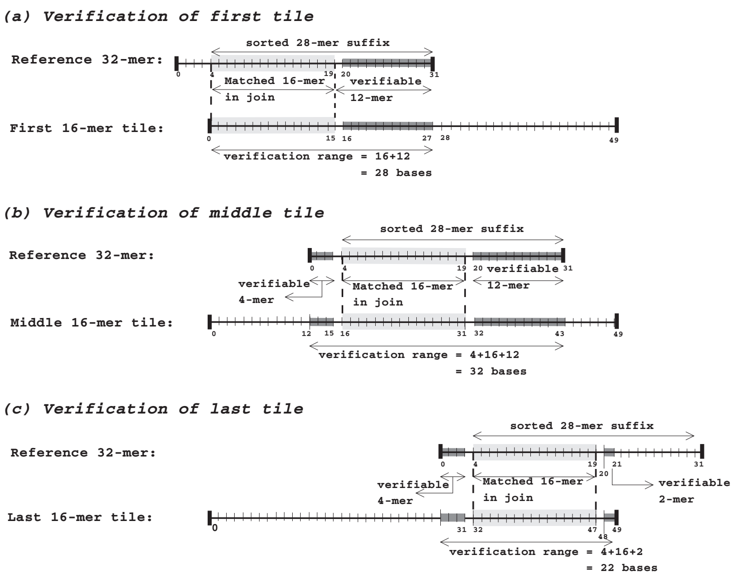

Fig. 3. An illustration of early-verification. (Assume: read length = 50; Hamming distance = 2; Tile size = 16. The reference list G is constructed on 32-mers sorted on their 28-mer suffixes.) **(a) Verification of all first tiles:** When the key of a first tile in a read matches a 16-mer key from the reference, bases at the positions [0-15] of the read are equivalent to those at the positions [4-19] of the reference. Early-verification can be performed on bases at the positions [16-27] of the read with those in positions [20-31] of the reference. **(b) Verification of all middle tiles:** When the key of a middle tile in a read matches a 16-mer key from the reference, bases at the positions [16-31] of the read are equivalent to those at the positions [4-19] in the reference. Early-verification can be performed on bases at the positions [12-15] and [32-43] of the read with those at the positions [0-3] and [20-31] of the reference respectively. **(c) Verification of all last tiles:** When the key of a last tile in some read matches with a 16-mer key from the reference, bases at the positions [32-47] in the read are equivalent to bases at the positions [4-19] in the reference. Early-verification can be performed on bases at the positions [28-31] and [48-49] of the read with those in the positions [0-3] and [20-21] of the reference respectively.

(See Fig. 3(a).) The middle tile can be verified on all 32-positions including the 4 base prefix that was left out from sorting. (See Fig. 3(b).) Importantly, the last tile can now be verified on the remaining 2 bases (49th and 50th) in the read, plus on the 4 bases preceding the last tile. (See Fig. 3(c).) In summary, early-verification filters out a significant portion of false-positives from entering into the join list, thereby severely constraining the final size of the join list. This naturally translates to a significant boost in the program's run time since it is implemented out of core.

3.4 Overlapped I/O

Syzygy uses a system of memory-mapped files and buffered writing for all I/O involving its data structures that are maintained on disk.

Memory mapping is an efficient alternative to standard file I/O on Unix-based systems where the kernel provides an interface to seamlessly map portions of very large files into memory. POSIX.1 standardizes the system call `mmap()` which most current Unix-based distributions implement. In addition, the user can advise the kernel in advance on how a memory-mapped block can be utilized. Linux provides, for example, `madvise()` system call for this purpose. Where the kernel recognizes (or is advised) that the accesses to memory mapped blocks are sequential, it automatically optimizes the read-performance by *caching* asynchronously pre-fetched pages of data through aggressive read-ahead from the point where data is being processed in the memory-mapped block.

Syzygy uses `write()` defined in POSIX.1 to write data to disk. The behaviour of write() is optimized by Linux. Write calls return immediately because the kernel copies the data into its buffers, batching many writes together and deferring the *writeback* to disk at a later time to be written asynchronously.

3.5 Sorting

Syzygy implements an efficient *external* sorting that is capable of sorting a very large number of records (and their payloads) on disk. Broadly, the sorting is performed in two steps. In the first step the collection is partitioned into several consecutive blocks or *runs*, where each run is fully sorted in memory. The size of each partition depends on the user-defined memory limit. Syzygy implements a variant of *least significant digit* (LSD) radix sort to sort the runs in memory. We note that the radix sort in general has a complexity of $O(kn)$, where k is the number of radixes in the sort-key while n is the size of the run. Our implementation uses byte-size radices. Sorting a sort-key of size one 64-bit word requires 8 linear passes (using byte-size radices) on the run. Our implementation additionally benefits from several algorithmic and hardware-directed optimizations derived from the works of [21] and [22]. The second step involves a merge step which merges partially sorted runs of a collection on disk into a fully sorted collection.

4 Results and Discussion

One tool from each of the three categories of read mapping programs was chosen to compare against our prototype implementation of Syzygy. Specifically, `maq` [10] from category 1, `soap(v.1)` [13] from category 2 and `bwa` [16] from category 3 were chosen for comparisons.

Reads used here belong to sample NA19240, sequenced using *Illumina* technology, downloaded from the *1000 Genome Project.*[2] Human genome (draft 18)

[2] `http://www.1000genomes.org/`

Table 1. Results of comparison of `Syzygy` with `maq`, `soap(v.1)` and `bwa` when mapping varying sizes of reads on the Human genome under a Hamming distance threshold of 2. All programs were run on a single core as a sequential program. '–' in various columns indicate that the program ran exceeded 48 hours and hence were aborted. Runs on `Syzygy` were performed with a 1GB memory limit. (`Syzygy` runs faster with a larger memory limit.)

Program	nReads	Time (wall)	Memory	Scratch
maq	1 million	1.9 hrs	1.1 GB	0
soap(v.1)	1 million	7.4 hrs	14.1 GB	0
bwa	1 million	0.4 hrs	2.3 GB	0
Syzygy	1 million	0.7 hrs	1 GB	5.7 GB
maq	10 million	21.6 hrs	5.7 GB	0
soap(v.1)	10 million	—	—	0
bwa	10 million	3.7 hrs	2.3 GB	0
Syzygy	10 million	3.4 hrs	1 GB	27 GB
maq	50 million	—	—	0
soap(v.1)	50 million	—	—	0
bwa	50 million	20.5 hrs	2.3 GB	0
Syzygy	50 million	8.2 hrs	1 GB	65.4 GB
maq	100 million	—	—	0
soap(v.1)	100 million	—	—	0
bwa	100 million	40.0 hrs	2.3 GB	0
Syzygy	100 million	13.9 hrs	1 GB	103 GB

was used as the reference genome. All experiments were carried out on a single node of a AMD Quad-core server with 32GB of main memory. The server is connected to a large array of Serially Connected SCSI (SAS) disks accessible via fast Ethernet (giving a upper limit of 125 MB/s on the disk bandwidth).[3]

Table 1 summarizes the results of comparison between various programs. From the table we can see that as the volume of reads increases, the run times of other programs grow drastically. `soap(v.1)` has the worst run time of all the methods, becoming impractical even at a read volume of 10 million. At a read volume of 50 million, `maq` becomes impractical. Only `bwa` scales to 50 million reads and beyond. `Syzygy` runs significantly faster than all other program, especially on very large read volumes. At the read volume of 100 million, `Syzygy` is $\sim$ 2.8x faster than `bwa`. The results clearly show that `Syzygy` scales elegantly compared to other methods.

Notice in Table 1 that the memory usages of various programs. `Syzygy` runs in the user-stipulated amount of memory (in this case 1GB) while the rest of the programs have a variable memory footprint depending on the size of various indexes they maintain. `soap(v.1)` uses an inverted hash table on the reference genome. For a human genome the size of this data structure is roughly 14GB

[3] The runs of `Syzygy` in Table 1 uses two independent disks to alternate reading the writing operations of the algorithm to reduce the I/O latency.

which has to be maintained in memory. `maq` uses an inverted index composed of pairs of k-mer tiles (that is, spaced tiles). However, since such an index prepared on large reference genomes is huge, `maq` chooses to prepare the inverted index on the read set. To further conserve space, `maq` indexes the read set in batches compromising on program speed. `bwa` which uses a Burrow-Wheeler index [20] has the most concise index structure among the tools. However, for the program to work efficiently, the entire index has to be maintained in memory. This comes in the way of mapping a set of reads on multiple (instead of one) genome.

`Syzygy` uses a large amount of scratch space on disk to allow the program to run in a fixed, user-defined amount of main memory. The last column of Table 1 gives the amount of scratch space used by the program for various runs. Note that, compared to main memory, disks are inexpensive and vastly bigger in sizes. (A standard 1TB disk costs only a couple of hundred Dollars.)

5 Conclusion

The design of an efficient read mapping algorithm, `Syzygy`, has been described in this paper. The program reorganizes the read mapping problem to maximize spatial locality of reference of data accesses in the algorithm, a crucial ingredient for any performance optimization. This facilitates the program to maintain all its data structures on disk, and hence allowing `Syzygy` to run in any user-defined amount of memory. `Syzygy` scales elegantly to volumes of read and genome data unachievable by current read-mapping programs. Our future work will extend `Syzygy` to handle paired-end read mapping while generalizing the alorithm for mapping under an edit distance threshold. We are also working towards parallelizing `Syzygy` for symmetric as well as distributed memory multiprocessors. An academic version of the program will be available shortly from `http://www.csse.unimelb.edu.au/~arun/syzygy`.

Acknowledgement

National ICT Australia (NICTA) is funded by the Australian Government's Department of Communications, Information Technology and the Arts, the Australian Research Council through Backing Australia's Ability, and the ICT Centre of Excellence programs.

References

1. Margulies, M., Egholm, M., Altman, W., et al.: Genome sequencing in microfabricated high-density picolitre reactors. Nature 437, 376–380 (2005)
2. Shendure, J., Porreca, G.J., Reppas, N.B., Lin, X., Mccutcheon, J.P., Rosenbaum, A.M., Wang, M.D., Zhang, K., Mitra, R.D., Church, G.M.: Accurate multiplex polony sequencing of an evolved bacterial genome. Science 309, 1728–1732 (2005)
3. Boyer, R.S., Moore, J.S.: A fast string searching algorithm. Commun. ACM 20(10), 762–772 (1977)

4. Knuth Jr., D.E., Pratt, V.R.: Fast pattern matching in strings. SIAM Journal on Computing 6(2), 323–350 (1977)
5. Karp, R.M., Rabin, M.O.: Efficient randomized pattern-matching algorithms. IBM Journal of Research and Development 31(2), 249–260 (1987)
6. Altschul, S.F., Gish, W., Miller, W., Myers, E.W., Lipman, D.J.: Basic local alignment search tool. Journal of Molecular Biology 215, 403–410 (1990)
7. Kent, W.J.: BLAT–the blast-like alignment tool 12, 656–664 (April 2002)
8. Cox, A.J.: Ultra-high throughput alignment of short sequence tags (2007) (unpublished)
9. Rumble, S.M., Lacroute, P., Dalca, A.V., Fiume, M., Sidow, A., Brudno, M.: SHRiMP: accurate mapping of short color-space reads. PLoS Computational Biology 5 (May 2009)
10. Li, H., Ruan, J., Durbin, R.: Mapping short dna sequencing reads and calling variants using mapping quality scores. Genome Research (August 2008)
11. Lin, H., Zhang, Z., Zhang, M.Q., Ma, B., Li, M.: ZOOM! zillions of oligos mapped. Bioinformatics 24, 2431–2437 (2008)
12. Jiang, H., Wong, W.H.: SeqMap: mapping massive amount of oligonucleotides to the genome. Bioinformatics 24, 2395–2396 (2008)
13. Li, R., Li, Y., Kristiansen, K., Wang, J.: SOAP: short oligonucleotide alignment program. Bioinformatics 24, 713–714 (2008)
14. Eaves, H.L., Gao, Y.: MOM: maximum oligonucleotide mapping. Bioinformatics 25, 969–970 (2009)
15. Campagna, D., Albiero, A., Bilardi, A., Caniato, E., Forcato, C., Manavski, S., Vitulo, N., Valle, G.: PASS: a program to align short sequences. Bioinformatics 25, 967–968 (2009)
16. Li, H., Durbin, R.: Fast and accurate short read alignment with Burrows-Wheeler transform. Bioinformatics 25, 1754–1760 (2009)
17. Langmead, B., Trapnell, C., Pop, M., Salzberg, S.L.: Ultrafast and memory-efficient alignment of short dna sequences to the human genome. Genome Research 10 (March 2009)
18. `http://www.vmatch.de/`
19. Malhis, N., Butterfield, Y.S., Ester, M., Jones, S.J.: Slider–maximum use of probability information for alignment of short sequence reads and snp detection. Bioinformatics 25, 6–13 (2009)
20. Ferragina, P., Manzini, G.: Opportunistic data structures with applications. In: Proceedings of Foundations of Computer Science, pp. 390–398 (2000)
21. McIlroy, P.K., Bostic, K., Mcilroy, M.D.: Engineering radix sort. Computing Systems 6, 5–27 (1993)
22. Kärkkäinen, J., Rantala, T.: Engineering radix sort for strings. In: Amir, A., Turpin, A., Moffat, A. (eds.) SPIRE 2008. LNCS, vol. 5280, pp. 3–14. Springer, Heidelberg (2008)
23. The quest for an accelerated population count. In: Oram, A., Wilson, G. (eds.) Beautiful code, pp. 147–160. O' Reilly, Sebastopol (2007)