1 INTRODUCTION

Fig. 1: Example reads and the corresponding de Bruijn graph for $k = 3$. The edge labels, indicating 4-mers with overlap 2, have been left out for better readability.

Fig. 2: Example reads and their corresponding string graph. Note that the overlap represented by edge $R_1 \rightarrow R_3$ can also be inferred by the path $R_1 \rightarrow R_2 \rightarrow R_3$.

**Contribution of PASQUAL Software System Architecture:** The PASQUAL software system architecture can be divided into three main steps.

1) (Section 3 of the main file)
   - Remove duplicate reads. Perform approximate membership queries with a Bloom filter [2]. (Currently, this step is carried out sequentially due to its insignificant costs.)
   - Build the suffix array in parallel. To this end we adapt the LS algorithm [9] to our needs.
   - Transform the suffix array into a compressed FM-index, trading running time for reduced memory consumption and improving on Simpson and Durbin [14] in terms of memory consumption due to the compression. Parallelism is used in several other steps throughout PASQUAL for acceleration in order to account for the running time investment incurred by compression.

2) (Section 4 of the main file)
   - Use the compressed FM-index to build the string graph in parallel. In addition to parallelism, the approach by Simpson and Durbin [14] is improved in this paper by two further techniques for overlap search and transitive edge removal.

3) (Section 5 of the main file)
   - Simplify the string graph by removing anomalous structures from it in parallel. In addition to the previously known structures tip and bubble, we identify two new structures called bridge and bubble combo. We introduce parallel algorithms to remove all four structures and present pseudocode for three of them.
   - Perform a parallel graph traversal to extract the contigs. Use an unaggressive approach that extracts only unambiguous simple paths.

2 PRELIMINARIES

2.1 Biological Background and Notation

2.2 Related Work

Related techniques for speeding up certain parts of the assembly process have appeared recently in works by Kundeti et al. [8] and by Dinh and Rajasekaran [4]. The first one proposes an improved construction algorithm for de Bruijn graphs, also for parallel and external memory settings. The authors show their approach to have better performance than an earlier approach of Jackson and Aluru [5] and Velvet. A direct translation of the results to overlap or string graphs is not obvious. The second paper [4] discusses memory-efficient sequential algorithms for constructing exact-match overlap graphs. A transfer to the parallel setting is still open.

The suffix tree is a data structure used for text indexing. It is employed in related biological applications such as sequence alignment [3] and DNA clustering [6]. In practice suffix arrays, which we use in PASQUAL, are generally preferred over suffix trees due to their smaller memory consumption and often better performance [12].
3 Parallel Index Construction

3.1 Removing Duplicate Reads

A Bloom filter is a memory-efficient data structure that supports constant-time set membership queries at the cost of a small false positive rate. Each query only requires the evaluation of a small number of hash functions, which usually takes constant time. Reads occurring only once have a small probability (in our setting typically less than 0.1%) of being removed from the data set by mistake. We perform this preprocessing step sequentially as it does not contribute significantly to the running time. If required, Bloom filters offer various possibilities for parallelization. For example, assuming support for atomic operations, each of the hash functions of one query can be evaluated concurrently.

3.2 Parallel Suffix Array Construction

<table>
<thead>
<tr>
<th>A</th>
<th>C</th>
<th>G</th>
<th>$</th>
<th>G</th>
<th>T</th>
<th>A</th>
<th>$</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>5</td>
<td>6</td>
<td>7</td>
<td>8</td>
</tr>
</tbody>
</table>

TABLE 1: The suffix array of the two concatenated reads “ACG$” and “GTAS$”.

Unfortunately, none of the suffix array construction algorithms we found in the literature satisfies all our requirements: Both fast and very memory-efficient in practice, as well as easy to parallelize. As an example, an implementation of the MP algorithm [10] and the library libdivsufsort [11] are among the fastest tools in practice, however, both of them require an inherently sequential in-order scan of the array. Kulla and Sanders [7] and Blelloch and Shun [1] parallelize the DC3 algorithm. Yet, DC3 requires ≈50% more running time and memory than LS [12] p. 7.

Algorithm 1: Parallel Suffix Array Sorting (I, P, M)

begin
  Initial sort: Radix sort (in parallel) the suffixes according to the lexicographic order of the initial I characters.
  Parallel LS sort: For each bucket yielded from initial sort, perform LS sort phase by phase. Buckets of each phase are sorted concurrently.
  Final sort: After the Pth phase of parallel LS sort, buckets are sorted recursively by TSQS. When the length of a bucket is less than M, use insertion sort to complete.

3.3 Parallel Compressed Index Construction

4 String Graph Construction

4.1 Overlap Search

Algorithm 2: Overlap search for a read R with length l adapted from [14].

Output: L, suffix intervals of the reads sharing overlaps with R.

L ← ∅; i ← 1 – l;
Initialize [u, v] as the interval of R[i];
while v ≤ u and i ≥ 0 do
  if i + 1 ≥ τ then
    [u, v] ← updateBwd([u, v], $);
    if u ≤ v then
      L ← L ∪ [u, v];
      [u, v] ← updateBwd([u, v], R[i]);
      i ← i – 1;
    i ← i – 1;

5 Parallel Graph Simplification and Contig Listing

5.1 Structures Caused by Sequencing Errors

5.2 Parallel Graph Simplification Algorithms

Algorithm 3: Initializing paths.

Input: G = (V, E)

Classify all vertices in parallel;
forall the v ∈ {u | u is BRANCH vertex} in parallel do
v.P ← ∅;
foreach neighbor w of v do
  p.end ← w; p.len ← 1;
  p.dir ← typeof(v → w);
  [len’, end’] ← extend_path(p);
  p.end ← end’; p.len ← len’;
  p.updated ← true;
  v.P ← v.P U p;

6 Experiments

Settings: The PASQUAL code has been compiled with GCC using -O3 optimization. Additional compilation flags include OpenMP and SSE4 support. The
TABLE 3: Simulated data sets generated from the genomes of (a) human and zebrafish and (b) data sets selected from NCBI short read archive (SRA).

<table>
<thead>
<tr>
<th>Organism/Genome length</th>
<th>Data set</th>
<th>Coverage</th>
<th>Read length (bp)</th>
<th>No. of reads</th>
<th>Total length (bp)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Human genome (chr22)</td>
<td>chr22_c30_135</td>
<td>30</td>
<td>35</td>
<td>29,909,610</td>
<td>1,046,836,350</td>
</tr>
<tr>
<td></td>
<td>chr22_c30_1100</td>
<td>30</td>
<td>100</td>
<td>10,468,362</td>
<td>1,046,836,200</td>
</tr>
<tr>
<td></td>
<td>chr22_c50_135</td>
<td>50</td>
<td>35</td>
<td>49,849,350</td>
<td>1,744,727,250</td>
</tr>
<tr>
<td></td>
<td>chr22_c50_1100</td>
<td>50</td>
<td>100</td>
<td>17,447,272</td>
<td>1,744,727,200</td>
</tr>
<tr>
<td>Zebrafish (chr6)</td>
<td>d_rerio_c30_1100</td>
<td>30</td>
<td>100</td>
<td>18,417,873</td>
<td>1,841,787,300</td>
</tr>
<tr>
<td></td>
<td>d_rerio_c30_1200</td>
<td>30</td>
<td>200</td>
<td>9,208,935</td>
<td>1,841,787,000</td>
</tr>
<tr>
<td></td>
<td>d_rerio_c50_1100</td>
<td>50</td>
<td>100</td>
<td>30,696,456</td>
<td>3,069,645,600</td>
</tr>
<tr>
<td></td>
<td>d_rerio_c50_1200</td>
<td>50</td>
<td>200</td>
<td>15,348,228</td>
<td>3,069,645,600</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Organism/Genome length</th>
<th>Accession No.</th>
<th>Read length (bp)</th>
<th>No. of reads</th>
<th>No. of bases</th>
</tr>
</thead>
<tbody>
<tr>
<td>Escherichia coli str. K-12</td>
<td>SRX000429</td>
<td>36</td>
<td>20,816,448</td>
<td>749.4M</td>
</tr>
<tr>
<td>S. typhimurium T000240</td>
<td>DRX000261</td>
<td>80</td>
<td>13,460,777</td>
<td>1.1G</td>
</tr>
<tr>
<td>Caenorhabditis elegans</td>
<td>SRX026594</td>
<td>100</td>
<td>67,617,092</td>
<td>6.8G</td>
</tr>
</tbody>
</table>

experiments are carried out on a single x86-64 server, a Dell PowerEdge R710 with 48 GB RAM. It has 2 quad-core Intel Xeon X5570 processors, 8MB cache per processor, Hyperthreading and Turbo Boost enabled. The installed GCC version is 4.5.0. For analyzing the scalability of our assembler, we perform additional experiments on an Intel server called mirasol with a total of 252 GB RAM. It has four Intel Xeon E7-8870 processors, each with 10 cores and Hyperthreading enabled, sharing 30 MB L3 cache. Here we use GCC 4.4.5.

Data Sets: The real data sets we use include Escherichia coli, S. typhimurium and Caenorhabditis elegans, corresponding to NCBI accession numbers SRX000429, DRX000261 and SRX026594, respectively. The data sets selected for experiments differ in read length and number of bases so as to provide diverse inputs to assemblers.

6.1 Solution Quality

Regarding the number of mis-assemblies, PASQUAL clearly fares best. The large number of erroneous contigs for the other assemblers also puts their other quality results described in the previous paragraph into perspective. Note that SOAPdenovo yields a particularly high number of mis-assemblies, also for the simpler genomes like E. coli and S. typhimurium. The problem is to a lesser extent also true for the other two de Bruijn graph based assemblers. Moreover, as can be seen in the last multirow of Table 1 in the main file, the large number of mis-assemblies becomes more pronounced for genomes of complex organisms such as C. elegans because de Bruijn graph based assemblers do not look explicitly for reads that overlap; the overlaps are implicit in their graph construction instead. Thus, the $k$-mer length plays a critical role in determining the quality of sequence assembly. As already argued in the introduction, the value of $k$ largely affects the error sensitivity of assembly when using Velvet or other tools based on the same approach.

Note that the mis-assembled contigs can be categorized further based on the reason why the error appears [13]. This is not pursued here as we believe such an analysis to be beyond the scope of this paper, which concentrates on computational techniques for parallel assembly.

6.2 Performance and Resource Consumption

Table 5 shows the running time of PASQUAL broken down by each of its three stages index construction, graph construction, and graph simplification. Index construction is the most expensive stage, almost taking half of the running time. However, indexing is a good investment since the FM-index is responsible for the efficiency of overlap search and graph construction. Another important observation from the table is that the running time of graph construction and graph simplification highly depends on the value of $\tau$ used. That is because the string graph with smaller $\tau$ usually has more transitive edges to remove and more complicated structures to simplify.

Fig. 4 shows the speedups obtained with increasing thread numbers for assembling some of the data sets...
Algorithm 4: Removing tips and bubbles.

Input: $G = (V, E)$, $\lambda, \mu$

Initialization: $S \leftarrow \{v \mid v \text{ is BRANCH vertex}\}$

forall $v \in V$, $v.changed \leftarrow false$

finished $\leftarrow false$

while not finished do

finished $\leftarrow true$;

/* 1. Path extension */

forall the $v \in S$ in parallel do

forall the $p \in v.P$ do

$w \leftarrow p.end$;

if $w.changed$ then

$\{\text{len'}}, \text{end'}\} \leftarrow \text{extend_path}(p)$;

$p$.len $\leftarrow$ len'; $p$.end $\leftarrow$ end';

$p$.updated $\leftarrow$ true;

/* 2. Find tips and bubbles */

forall the $v \in S$ in parallel do

$v.changed$ $\leftarrow$ false;

forall the $p \in v.P$ do

if $p$.updated and $p$.len $< \lambda$ then

if $p.end.type$ = ENDPOINT then

$p$ is a tip, remove $p$;

$v.changed$ $\leftarrow$ true;

if $p.end.type$ = BRANCH then

forall the $q \in v.P$ and $q \neq p$ and $q.dir = p.dir$ do

if $q.end = p.end$ with $q.len - p.len > \mu$ then

$p$ and $q$ form a bubble, remove $p$;

$v.changed$ $\leftarrow$ true;

$p$.updated $\leftarrow$ false;

if $v.changed$ then

finished $\leftarrow$ false;

relabel the type of $v$;

if $v.type$ $\neq$ BRANCH then

Remove $v$ from $S$;

TABLE 4: Additional assembly statistics: The solution quality values refer to the best assembly, which has been selected from a collection resulting from six different values for overlap parameter $\tau, k$ for each read length. For read length $l = 36$, $k \in \{21..31..2\}$, for $l = 80$, $k \in \{41..57..2\}$, for $l = 100$, $k \in \{53..63..2\}$, and for $l = 200$, $k \in \{153, 155, 157, 159, 161, 165\}$.

<table>
<thead>
<tr>
<th>Data set</th>
<th>No. of contigs</th>
<th>Max. length (bp)</th>
<th>Total N50 (bp)</th>
<th>N50 (%) length</th>
<th>N50 Error. contigs</th>
<th>contigs</th>
</tr>
</thead>
<tbody>
<tr>
<td>d_reirsi</td>
<td>V</td>
<td>Segmentation fault</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>C30</td>
<td>A</td>
<td>Fails to assemble</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1100</td>
<td>S</td>
<td>Cannot handle $k &gt; 127$</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Comparison to LEAP: Additional experiments with the recent sequential tool LEAP [4] on some of the depicted data sets show that PASQUAL with 16 threads is about three times faster. While LEAP is very memory-efficient, its solutions for our (error-free) simulated data sets are clearly inferior with thousands of mis-assembled contigs and a significantly smaller N50 length.

TABLE 5: Running time for various stages of PASQUAL using 40 cores with 80 OpenMP threads on mirasol.

<table>
<thead>
<tr>
<th>Data set</th>
<th>Time in seconds</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Index construction</td>
</tr>
<tr>
<td>E. coli</td>
<td>21</td>
</tr>
<tr>
<td>S. typhimurium</td>
<td>43</td>
</tr>
<tr>
<td>C. elegans</td>
<td>61</td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Fig. 4: Speedup achieved by PASQUAL on an eight-core Intel Xeon X5570 system.

REFERENCES


