Supplementary Material

1 Introduction

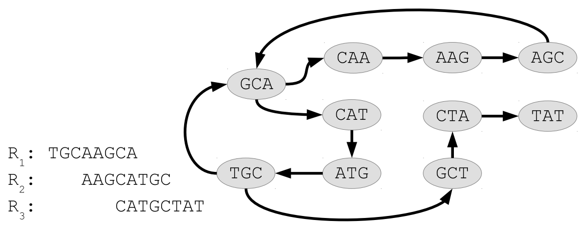

Fig. 1: Example reads and the corresponding de Bruijn graph for $k = 3$. The edge labels, indicating 4-mers with overlap 2, have been left out for better readability.

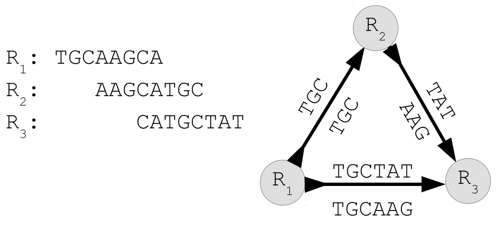

Fig. 2: Example reads and their corresponding string graph. Note that the overlap represented by edge $R_1 \rightarrow R_3$ can also be inferred by the path $R_1 \rightarrow R_2 \rightarrow R_3$.

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 aligment [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

	A	C	G	$	G	T	A	$
	1	2	3	4	5	6	7	8
8								$
4				$	G	T	A	$
7							A	$
1	A	C	G	$	G	T	A	$
2		C	G	$	G	T	A	$
3			G	$	G	T	A	$
5					G	T	A	$
6						T	A	$
SA	8	4	7	1	2	3	5	6

TABLE 1: The suffix array of the two concatenated reads "ACG$" and "GTA$".

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 $\approx 50\%$ 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 $P$th 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

	A	C	G	$	G	T	A	$
	1	2	3	4	5	6	7	8
SA	8	4	7	1	2	3	5	6
B	A	G	T	$	A	C	$	G
$Occ_B(\$)$	0	0	0	1	1	1	2	2
$Occ_B(A)$	1	1	1	1	2	2	2	2
$Occ_B(C)$	0	0	0	0	0	1	1	1
$Occ_B(G)$	0	1	1	1	1	1	1	2
$Occ_B(T)$	0	0	0	0	0	1	1	1
$C(A)$	2							
$C(C)$	4							
$C(G)$	5							
$C(T)$	7							

TABLE 2: The suffix array, BWT and FM-index of the two concatenated reads "ACG$" and "GTA$".

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 $R$.
$L \leftarrow \emptyset$; $i \leftarrow l - 1$;
Initialize $[u, v]$ as the interval of $R[l]$;
while $v \leq u$ and $i \geq 0$ do
    if $l - i + 1 \geq \tau$ then
        $[u_\$, v_\$] \leftarrow$ updateBwd($[u, v], \$$);
        if $u_\$ \leq v_\$$ then
            $L \leftarrow L \cup [u_\$, v_\$]$;
    $[u, v] \leftarrow$ updateBwd($[u, v], R[i]$);
    $i \leftarrow i - 1$;
```

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 $v \in \{u \mid u$ is BRANCH vertex$\}$ in parallel do
    $v.P \leftarrow \emptyset$;
    foreach neighbor $w$ of $v$ do
        $p.end \leftarrow w$; $p.len \leftarrow 1$;
        $p.dir \leftarrow$ typeof($v \rightarrow w$);
        $\{len', end'\} \leftarrow$ extend_path($p$);
        $p.end \leftarrow end'$; $p.len \leftarrow 1en'$;
        $p.updated \leftarrow$ true;
        $v.P \leftarrow v.P \cup p$;
```

6 Experiments

Settings: The Pasqual code has been compiled with GCC using -O3 optimization. Additional compilation flags include OpenMP and SSE4 support. The

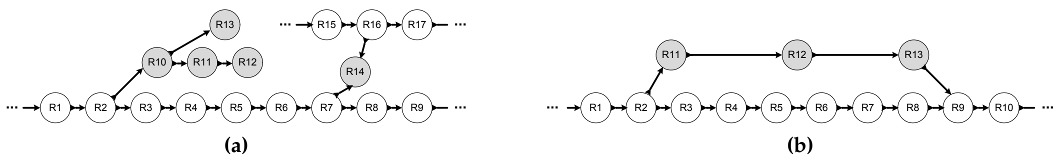

Fig. 3: Special graph structures caused by sequencing errors. (a) *tip*, (b) *bubble*. The vertices of erroneous reads are shown in grey.

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).

	Organism/ Genome length	Data set	Coverage	Read length (bp)	No. of reads	Total length (bp)
(a)	*Human genome (chr22)* / 33.5Mbp	chr22_c30_l35	30	35	29,909,610	1,046,836,350
		chr22_c30_l100	30	100	10,468,362	1,046,836,200
		chr22_c50_l35	50	35	49,849,350	1,744,727,250
		chr22_c50_l100	50	100	17,447,272	1,744,727,200
	Zebrafish (chr6) / 61Mbp	d_rerio_c30_l100	30	100	18,417,873	1,841,787,300
		d_rerio_c30_l200	30	200	9,208,935	1,841,787,000
		d_rerio_c50_l100	50	100	30,696,456	3,069,645,600
		d_rerio_c50_l200	50	200	15,348,228	3,069,645,600

	Organism/ Genome length	Accession No.	Read length (bp)	No. of reads	No. of bases
(b)	*Escherichia coli str. K-12* / 4.6M	SRX000429	36	20,816,448	749.4M
	S. typhimurium T000240 / 4.9M	DRX000261	80	13,460,777	1.1G
	Caenorhabditis elegans / 100M	SRX026594	100	67,617,092	6.8G

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 τ used. That is because the string graph with smaller τ 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: $S \leftarrow \{v \mid v$ is BRANCH vertex$\}$;
forall $v \in V$, $v.changed \leftarrow false$;
$finished$ = 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
                $\{len', end'\} \leftarrow$ 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
        $v.changed$ = 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 τ, 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 $l = 200$, $k \in \{153, 155, 157, 159, 161, 165\}$.

Data set	Tool	No. of contigs	Max length (bp)	Total length (bp)	N50 length	N50 contigs	Erron. contigs
	V			Segmentation fault			
d_rerio	E			Fails to assemble			
c30_	A			Execution hangs			
l100	S			Cannot handle $k > 127$			
	P	25,036	67,085	59,011,862	7,911	2,165	0

with PASQUAL (the other data sets are similar and therefore omitted to remove clutter). The values are almost linear when compared to one thread, sometimes slightly superlinear due to cache effects. Even when using hyperthreading, we obtain good scalability with 16 threads, as PASQUAL is 10-13 times faster than with one thread.

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

Data set	τ	Time in seconds: Index construction	Graph construction	Graph simplification
E. coli	21	17.98	4.85	10.24
	31	17.84	3.83	7.85
S. typhimurium	43	32.54	12.00	22.65
	53	32.77	7.89	10.27
C. elegans	61	470.46	111.92	186.23
	81	465.33	77.01	137.28

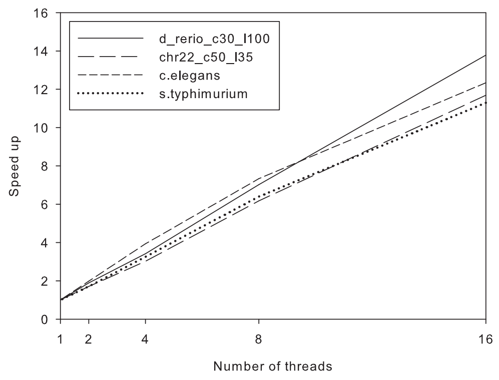

Fig. 4: Speedup achieved by PASQUAL on an eight-core Intel Xeon X5570 system.

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.

REFERENCES

[1] G. E. Blelloch and J. Shun, "A simple parallel cartesian tree algorithm and its application to suffix tree construction," in *Proc. Workshop on Algorithm Engineering and Experiments (ALENEX 2011)*. SIAM, 2011, pp. 48–58.

[2] B. H. Bloom, "Space/time trade-offs in hash coding with allowable errors," *Communications of the ACM*, vol. 13, pp. 422–426, 1970.

[3] A. L. Delcher, A. Phillippy, J. Carlton, and S. L. Salzberg, "Fast algorithms for large-scale genome alignment and comparison," *Nucleic Acids Research*, vol. 30, no. 11, pp. 2478–2483, 2002. [Online]. Available: http://nar.oxfordjournals.org/content/30/11/2478.abstract

[4] H. Dinh and S. Rajasekaran, "A memory-efficient data structure representing exact-match overlap graphs with application for next-generation DNA assembly," *Bioinformatics*, vol. 27, pp. 1901–1907, 2011.

[5] B. G. Jackson and S. Aluru, "Parallel construction of bidirected string graphs for genome assembly," in *Proceedings of the 2008 37th International Conference on Parallel Processing*. Washington, DC, USA: IEEE Computer Society, 2008, pp. 346–353.

[6] A. Kalyanaraman, S. Emrich, P. Schnable, and S. Aluru, "Assembling genomes on large-scale parallel computers," *Journal of Parallel and Distributed Computing*, vol. 67, no. 12, pp. 1240 – 1255, 2007, best Paper Awards: 20th International Parallel and Distributed Processing Symposium (IPDPS 2006).

[Online]. Available: http://www.sciencedirect.com/science/article/pii/S0743731507000810

[7] F. Kulla and P. Sanders, “Scalable parallel suffix array construction,” *Parallel Computing*, vol. 33, pp. 605–612, September 2007.

[8] V. Kundeti, S. Rajasekaran, H. Dinh, M. Vaughn, and V. Thapar, “Efficient parallel and out of core algorithms for constructing large bi-directed de bruijn graphs,” *BMC Bioinformatics*, vol. 11, no. 1, p. 560, 2010.

[9] N. J. Larsson and K. Sadakane, “Faster suffix sorting,” *Theoretical Computer Science*, vol. 387, pp. 258–272, November 2007.

[10] M. A. Maniscalco and S. J. Puglisi, “Faster lightweight suffix array construction,” in *Proceedings of the 17th Australasian Workshop on Combinatorial Algorithms (AWOCA’06)*, 2006, pp. 16–29.

[11] Y. Mori, “Divsufsort.” [Online]. Available: http://www.homepage3.nifty.com/wpage/software/libdivsufsort.html

[12] S. J. Puglisi, W. F. Smyth, and A. H. Turpin, “A taxonomy of suffix array construction algorithms,” *ACM Computing Surveys*, vol. 39, July 2007.

[13] S. L. Salzberg, A. M. Phillippy, A. Zimin, D. Puiu, T. Magoc, S. Koren, T. J. Treangen, M. C. Schatz, A. L. Delcher, M. Roberts, G. Marais, M. Pop, and J. A. Yorke, “Gage: A critical evaluation of genome assemblies and assembly algorithms,” *Genome Research*, 2011. [Online]. Available: http://genome.cshlp.org/content/early/2012/01/12/gr.131383.111.abstract

[14] J. Simpson and R. Durbin, “Efficient construction of an assembly string graph using the FM-index,” *Bioinformatics*, vol. 26, no. 12, p. i367, 2010.