Scalable Sequence Similarity Search and Join in Main Memory on Multi-Cores

Astrid Rheinländer and Ulf Leser

Humboldt-Universität zu Berlin, Department of Computer Science
Berlin, Germany

Abstract. Similarity-based queries play an important role in many large scale applications. In bioinformatics, DNA sequencing produces huge collections of strings, that need to be compared and merged. One strategy to speed up similarity-based queries is parallelization on clusters using MapReduce. However, distributing data over a cluster also incurs high cost. At the same time, modern hardware offers parallelization through multi-cores and can be equipped with large main memories at low cost. We present PeARL, a data structure and algorithms for similarity-based queries on many-core servers. PeARL indexes large string collections in compressed tries which are entirely held in main memory. Parallelization of searches and joins is performed using MapReduce as the underlying execution paradigm. We show that our data structure is capable of performing many real-world applications in sequence comparisons in main memory. Our evaluation reveals that PeARL reaches a significant performance gain compared to single-threaded solutions. However, the evaluation also shows that scalability should be further improved, e.g., by reducing sequential parts of the algorithms.

1 Introduction

Similarity-based searches and joins are important for many applications such as document clustering or near-duplicate and plagiarism detection [8, 9, 18]. In bioinformatics, similarity-based queries are used for sequence read alignment or for finding homologous sequences between different species. In recent years, much effort has been spent on developing tools to speed up similarity-based queries on sequences. Many prominent tools use sophisticated index structures and filter techniques that enable significant runtime improvements [3, 10, 11].

A challenge arises from the immense growth of sequence databases in the past few years. For example, the number of sequences stored in EMBL grows exponentially every year and sums up to more than 300 billion nucleotides as of May 2011. One strategy to deal with this huge amount of data is to divide it into smaller parts and perform analyses partition-wise in parallel. For this scenario, Google developed the programming paradigm MapReduce to enable a massively-parallel processing of huge data sets in large distributed systems of commodity hardware [5]. Recently, many open-source frameworks that build upon MapReduce have been developed [2, 4]. A main advantage of these tools is that they take many tasks off the user's shoulders, such as the actual process management in a distributed environment. At the downside, the main bottleneck of distributed MapReduce is network bandwidth and disk I/O. Therefore, another option is to design data structures and algorithms that adapt the MapReduce paradigm for many-core servers [13]. We argue that modern many-core servers, combined with the constantly falling prices for main memory, are perfectly suited to perform many real-world applications in sequence analysis. Such settings are much easier to maintain and do not suffer from bandwidth problems.

In this paper, we challenge the current opinion that problems in sequence analysis already have grown so big that distributed systems are the only solution. We present PEARL, a main-memory data structure and parallel algorithms for similarity-based search and join operations on sequence data. In particular, our data structure uses compressed tries. In tries, the complexity for exact searches only depends on string lengths and not on the number of stored strings [16]. This allows an efficient execution of exact searches even in large tries. In order to retain these advantages for similarity-based queries, we store additional information at each node that enable early pruning of whole subtries. Previously, we demonstrated that these strategies effectively speed up similarity-based queries in PETER [14], a disk-based index structure and predecessor of PeARL.

A crucial aspect in designing data structures for similarity based queries that interact with MapReduce is to support proper data partitioning. Specifically, we show how tries on top of large string collections can be compressed and partitioned for enabling in-memory MapReduce based search and join operations. To our knowledge, this is the first work that parallelizes similarity-based string searches and joins in tries. Our evaluation reveals that PeARL's similarity-based algorithms scale well.

The rest of this paper is organized as follows: Section 2 introduces basic concepts needed for the design of our data structure and algorithms. We describe design principles of PeARL and algorithms for similarity search and join, as well as our parallelization strategy in Sect. 3. We evaluate our tool in Sect. 4 and discuss related work in Sect. 5. Finally, we conclude our paper with an outlook to future work.

2 Preliminaries

Let Σ be an alphabet and let Σ^* be the set of all strings of any finite length over Σ. The length of strings $r, s \in \Sigma^*$ is denoted by $|r|$ ($|s|$, respectively). A substring $s[i \dots j]$ of s starts at position i and ends at position j, ($1 \leq i \leq j \leq |s|$). Any substring of length $q \in \mathbb{N}$ is called q-gram. Conceptually, we will ground our algorithms on operators for similarity search and similarity join, which are defined as follows:

Definition 1: (Similarity-based operators)
Let s be a string, R a bag of strings, d a distance function, and k a threshold. The similarity-based search operator is defined as $sim_{search}(s, R, K) = \{r | d(r, s) \leq$

$k, r \in R\}$. Similarly, for two bags of strings R, S, the similarity-based join operator is defined as $sim_{join}(R, S, k) = \{(r, s)|d(r, s) \leq k, r \in R, s \in S\}$. $\square$

Similarity-based queries must be grounded on a concrete similarity measure. In PeARL, we support Hamming and edit distance. We focus on edit distance based operations in this paper, but see [14] for the key ideas on Hamming distance-based queries. In general, the edit distance of r and s is computed in $O(|r| * |s|)$ using dynamic programming. As we are mostly interested in finding highly similar strings within a previously defined distance threshold, we use the k-banded alignment algorithm [6] with time complexity $O(k * max|r|, |s|)$ instead.

Our parallelization strategy is inspired by the well-known programming model MapReduce, a two-step approach that consists of a map and a reduce phase [5]. Essentially, data is stored in `<key, value>`-pairs and partitioned into several subsets. In the map step, a user-defined function is applied to each item <k_i,v_i> from the input and a list of <k_j,v_j> pairs is emitted. In an intermediate phase, all items generated by map are grouped together on the basis of the keys k_j. Afterwards, the user-defined function in the reduce step is applied to each group and generates a final result set for each k_j.

3 Data Structure and Algorithms

In this section, we introduce our data structure PeARL together with algorithms for executing similarity string searches and joins in parallel. Conceptually, a PeARL index (see Fig. 1) is based on radix trees [12] and defined as follows:

Definition 2 (PeARL index):
Let R be a bag of strings. A PeARL index P_R for R consists of a set of rooted, compressed tries T_R , a sequence string seq, and a `<key,value>` data structure $StringIDMap$ and meets the following conditions:

1. (*Identification of strings*) The string seq is a concatenation of all $r \in R$. We assign a unique ID to each r, assembled from a serial number, the length of r, and the start position of r in seq.
2. (*Node types*) We distinguish between infix nodes and string nodes. An infix node is a node that represents some substring r_l of r, $|r_l| \geq 1$. Every r maps to exactly one node $x \in T_R$ such that the concatenation of all labels from T_R's root to x exactly is r. Such nodes x are called string node. We store a pair that consists of the node ID of x and the UID of r in the *StringIDMap*. If R contains multiple copies of r, all corresponding UIDs are assigned to x.
3. (*Storing infixes*) Each node u represents a sequence of characters of length $l \geq 1$. The labels of any two children v, w of u start with different characters. Node labels are not stored directly in the node, but retrieved via lookups in seq. Thus, u stores length and start position of the represented infix in seq .
4. (*Additional information*) Each node u stores additional attributes, namely the minimum (min) and maximum (max) lengths of strings stored in the subtrie starting at u, a character frequency vector fv and a bit-string qGr.

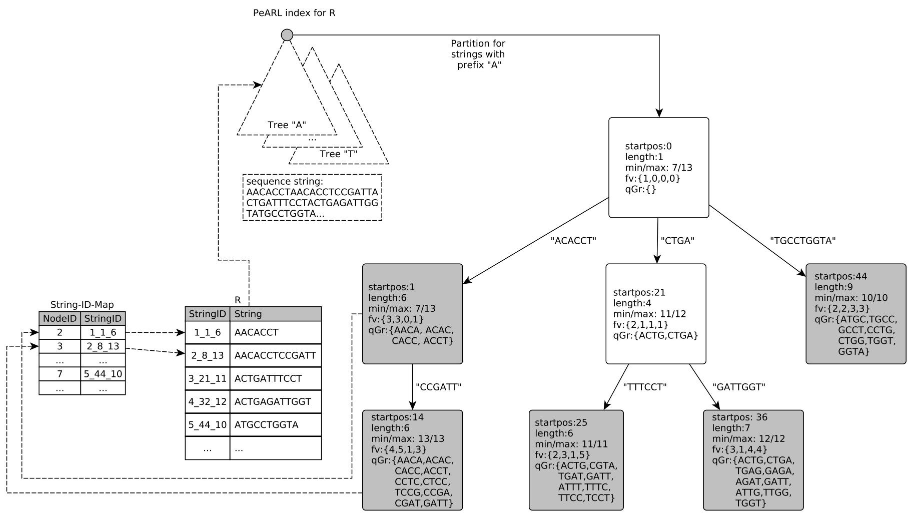
PeARL index for R
Partition for strings with prefix "A"
Tree "A"
...
Tree "T"
sequence string: AACACCTAACACCTCCGATTA CTGATTTCCTACTGAGATTGG TATGCCTGGTA...
String-ID-Map
NodeID | StringID
2 | 1_1_6
3 | 2_8_13
... | ...
7 | 5_44_10
... | ...
R
StringID | String
1_1_6 | AACACCT
2_8_13 | AACACCTCCGATT
3_21_11 | ACTGATTTCCT
4_32_12 | ACTGAGATTGGT
5_44_10 | ATGCCTGGTA
... | ...
startpos:0 length:1 min/max: 7/13 fv:{1,0,0,0} qGr:{}
"ACACCT"
"CTGA"
"TGCCTGGTA"
startpos:1 length:6 min/max: 7/13 fv:{3,3,0,1} qGr:{AACA, ACAC, CACC, ACCT}
startpos:21 length:4 min/max: 11/12 fv:{2,1,1,1} qGr:{ACTG,CTGA}
startpos:44 length:9 min/max: 10/10 fv:{2,2,3,3} qGr:{ATGC,TGCC, GCCT,CCTG, CTGG,TGGT, GGTA}
"CCGATT"
"TTTCCT"
"GATTGGT"
startpos:14 length:6 min/max: 13/13 fv:{4,5,1,3} qGr:{AACA,ACAC, CACC,ACCT, CCTC,CTCC, TCCG,CCGA, CGAT,GATT}
startpos:25 length:6 min/max: 11/11 fv:{2,3,1,5} qGr:{ACTG,CGTA, TGAT,GATT, ATTT,TTTC, TTCC,TCCT}
startpos: 36 length:7 min/max: 12/12 fv:{3,1,4,4} qGr:{ACTG,CTGA, TGAG,GAGA, AGAT,GATT, ATTG,TTGG, TGGT}

Fig. 1. PeARL index structure

The character frequency vector $fv(u)$ consists of $|\Sigma|$ components and counts the number of occurrences of $c_i \in \Sigma$ in the prefix represented by u in component i. Similarly, a bit in qGr at position i represents the *ith* string of all strings over Σ of length q in lexicographical order. Bit i is set to 1, if the prefix represented by node u contains the corresponding q-gram.

5. (*Trie partitioning*) For very large string collections, we expect the upper levels of a trie to be completely filled. Therefore, we partition a single PeARL trie into multiple tries on the basis of shared prefixes. Each partition is identified by the prefix which was used for partitioning (see Fig. 1). The prefix length used for partitioning is user-defined. $\square$

Figure 1 displays a PeARL index for strings over $\Sigma = \{A, C, G, T\}$. Grey nodes are string nodes, white nodes are internal nodes. The substring represented at some node can be retrieved in constant time via lookup in the sequence string. Edge labels are not stored in the index itself, but are displayed for better comprehensibility only. Displayed $q-$gram sets indicate which bits in qGr are set.

3.1 Algorithms

Building the PeARL index for a set R of strings works as follows: In a first step, R is sorted lexicographically, UIDs are assembled, and R is split into multiple partitions based on shared prefixes. For each partition $R_i \subseteq R$, we start with an empty trie T_{R_i} and iteratively insert each string contained in R_i using preorder DFS traversal. After all strings from R_i have been inserted, we iterate once over the whole trie and update the information min/max, fv and qGr.

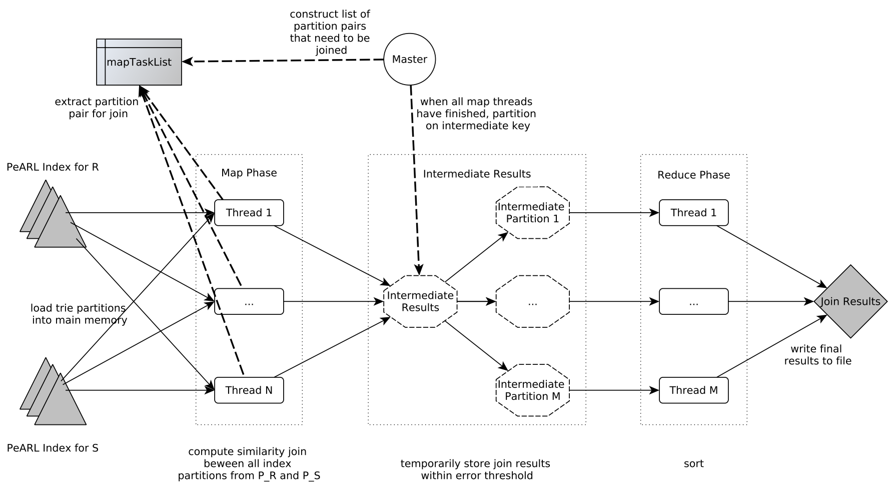
construct list of partition pairs that need to be joined
mapTaskList
Master
extract partition pair for join
when all map threads have finished, partition on intermediate key
PeARL Index for R
Map Phase
Intermediate Results
Reduce Phase
Thread 1
Intermediate Partition 1
Thread 1
load trie partitions into main memory
...
Intermediate Results
...
...
Join Results
Thread N
Intermediate Partition M
Thread M
write final results to file
PeARL Index for S
compute similarity join beween all index partitions from P_R and P_S
temporarily store join results within error threshold
sort

Fig. 2. MapReduce workflow of similarity joins in PeARL.

Similar to indexing, our algorithms for similarity-based searches and joins are also grounded on preorder DFS traversal of all trie partitions. Each algorithm is equipped with filtering strategies. These filters, namely prefix and edit distance pruning [16], character frequency pruning [1], and $q-$gram filtering [7], have been introduced in slightly different contexts before. Their concrete usage and efficiency for trie-based search and join queries is shown in [14]. Therefore, we only briefly summarize our search and join strategies in the following and concentrate on our novel parallelization scheme later.

Similarity search starts with a given search string q and traverses each trie partition in a PeARL index starting at root. Whenever a new child of the current node is reached, we first check whether we can prune this node (see [14] for details on filtering). If all filters have been passed successfully, we compute the edit distance between the query and the prefix of the node. If the distance exceeds a threshold k, we start a backtracking routine and traverse the remaining, not yet examined paths in the trie. Otherwise we descend forward to the leaves. When a string node x is reached and $d(q, x) \leq k$ holds, we report a match.

Similarity join for two sets R, S takes two PeARL indices P_R, P_S as input. Each trie partition T_{R_i} is joined with each partition T_{S_j}. Recall that both tries are partitioned by prefixes. We first check the partition prefixes on edit distance and it might happen that k is already exceeded. In this case, we skip the corresponding trie pair. Otherwise, we compute the similarity-based intersection of both partitions. As for search, we start at the root nodes and traverse both tries concurrently. When unseen nodes are reached, we check all filters and prune, if possible. Whenever two string nodes $x \in T_{R_i}, y \in T_{S_j}$ are reached, and given that $d(x, y) \leq k$ holds, we report a match.

3.2 Parallelization with In-Memory MapReduce

We use MapReduce to parallelize PeARL for an execution on multi-core servers. However, a usage in distributed scenarios is conceptually also possible as PeARL trie partitions could as well be spread over nodes in a distributed file system.

Recall that a user-defined function is applied to each input item `<`k_i`,`v_i`>` in the map phase. Depending on the specific task, we use the map phase to either execute the similarity join of any two PeARL partitions or, to search a certain string in each partition of a PeARL index. Reduce phases are typically used to compute aggregates of intermediate results. However, we use reduce to sort the similarity-based search or join results, and to perform some nice to have operations. Figure 2 shows the workflow for parallelizing similarity joins in PeARL with MapReduce. A master routine takes two PeARL indices P_R, P_S as input, together with an error threshold k, and a number of available threads t. As string collections stored with PeARL are already partitioned into multiple tries, we get a natural data partitioning for the map phase. The master generates a set of map tasks (stored in a FIFO data structure `mapTaskList`), such that each trie partition $T_{R_i} \in P_R$ is joined with each trie partition $T_{S_j} \in P_S$ and starts the map phase. To clearly identify each task, the master assembles a key from the partition prefixes for each (T_{R_i}, T_{S_j})-pair that is inserted into `mapTaskList`.

Each map thread has access to `mapTaskList` and extracts one task (T_{R_i}, T_{S_j}) out of this list. After some initialization steps, map calls the join routine, that executes the similarity join of $T_{R_i} \bowtie_k T_{S_j}$ and returns the set of all similar string pairs contained in (T_{R_i}, T_{S_j}) within the given distance k. These items are inserted into an intermediate data structure, that will be processed further by reduce. For each similar string pair (r, s), an intermediate key is set to the UID of r. When one map iteration has finished and as long as `mapTaskList` is not empty, the map thread extracts the next (T_{R_i}, T_{S_j}) pair out of this list and again computes the similarity join.

When all map tasks have been processed, the master partitions all intermediate data on the basis of intermediate keys and passes each partition to a separate reduce thread. This ensures that all similar string pairs which involve r are assigned to the same intermediate partition. Finally, reduce sorts all (r, s) pairs based by edit distance. Optionally, reduce can also emit the number of similar strings found in S for each r, or filter the results found for r on best score.

Parallelizing similarity searches is analog to the parallelization of similarity joins. The main difference is that P_S is replaced with one or a list of search sequences. If not existent, each search pattern is assigned a unique ID. For searches, the `mapTaskList` contains `<`k_i`,`v_i`>` pairs where k_i is a partition prefix of and v_i consists of T_{R_i} and the search sequence(s). A map step performs the similarity search for each search pattern in T_{R_i}. All resulting similar string pairs are stored an intermediate `<`k_i`,`v_i`>` data structure, such that k_i consists of the UID of the pattern, and v_i consists of the string pair together with its edit distance score. When all map threads have finished, the intermediate result set is partitioned on k_i and finally, reduce sorts the strings found for one search pattern by score.

In our approach, we reduced the necessary disk I/O to a minimum. Before initializing the MapReduce workflow, PeARL indices and search patterns are read from disk into memory and finally, after all reduce steps are terminated, the result set is either written to file or printed to `stdout`. In between these steps, all necessary operations are performed in main memory.

4 Evaluation

We evaluated the performance of PeARL on a NUMA server with 24 cores and 256 GB main memory available. All experiments were executed with `numactl -localalloc` to control the memory accession strategy and thread placement. Test data sets (see Table 1) are extracted from dbEST[1] as of March 7th, 2011 for the organism mouse. Indexing is linear in the number of indexed strings [14] and is not included in the reported measurements. In terms of memory consumption, PeARL needs roughly 20 GB of main memory to index all infixes of length 2,000 bp in the C. elegans genome (roughly 100M strings). For computing the similarity join III$\bowtie_k$IV, PeARL needs approx. 8 GB of main memory.

Set	# strings	⌀ length (min / max)	# characters
I	10,000	511.99 (49 / 1,190)	5,120,495
II	240,000	455.94 (18 / 2,160)	109,425,487
III	300,000	446.74 (18 / 2,160)	134,023,819
IV	1,000,000	512.12 (7 / 3,920)	512,123,043

Table 1. Data sets extracted from dbEST

4.1 Performance of Similarity-Based Operators

First, we compared the performance of all similarity-based operations in PeARL with its predecessor PETER in single-threaded mode. The main difference of both tools is that in PeARL, all parts of the index are kept in main memory whereas in PETER, disk I/O was necessary during search and join. Another difference is that q-gram sets in PeARL are stored persistently in the index whereas previously, q-grams were computed on the fly. Trie partitioning and parallelization was also not present in the predecessor. Overall, we observed that these improvements increased the efficiency of our filters. Whereas in PETER, filtering lead to runtime improvements of up to 80% compared to the baseline with no filters enabled, we now achieve runtime improvements of up to 99% caused by filtering (data not shown).

We evaluated the runtime of similarity search and measured 10,000 individual searches of non-indexed patterns from set I in the PeARL index for set IV, see Fig. 3. In single-threaded mode, searches in PETER ran significantly faster than in PeARL (up to factor 10 on $k = 2$). This is not surprising, as there is some

[1] www.ncbi.nlm.nih.gov/dbEST/

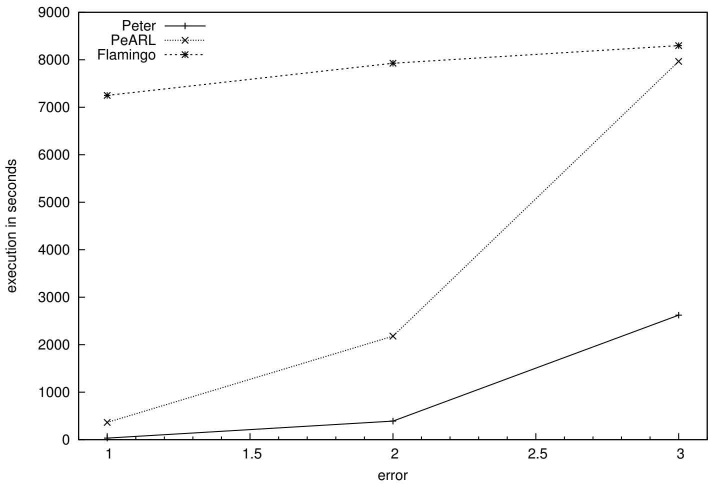

Fig. 3. Performance of single-threaded similarity search of 10,000 patterns from set I in set IV.

overhead introduced in PeARL by the added functionality for MapReduce based parallelization, which is also present in single-threaded searches. However, we will see in the following section that this overhead pays out for multi-threaded similarity searches and joins. We also compared PeARL to Flamingo, a library for string searching developed at UC Irvine[2]. As displayed in Fig. 3, PeARL outperforms Flamingo for search in single threaded mode for small thresholds (factor 20 for $k = 1$ and factor 3 for $k = 2$). For larger k, Flamingo begins to outperform PeARL.

For evaluating the runtime of similarity joins in PETER and PeARL, we computed the join between set IV and varying subsets taken from set II. As shown in Fig. 4, similarity joins in PeARL are computed considerably faster than in PETER. For example, we reached an improvement of factor 3 on $k = 2$ at a join cardinality of 2e+11. Generally speaking, the implemented improvements in PeARL are the more profitable when indexed string sets grow large. We could not compare PeARL to Flamingo for joins, since no reference implementation was available.

4.2 Scalability of PeARL

We compared the multi-threaded execution of 10,000 individual searches of patterns from set I in set IV with PeARL (24 threads) to a single-threaded execution with PeARL and Flamingo. As displayed in Fig. 5, the multi-threaded execution in PeARL outperforms the single-threaded execution in Flamingo with factors

[2] http://flamingo.ics.uci.edu/

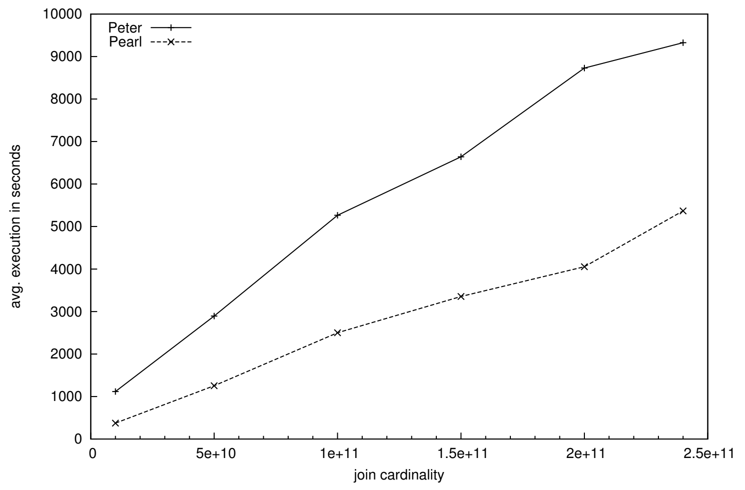

Fig. 4. Performance of single-threaded similarity join IV$\bowtie_{k=2}$II on subsets of II with sizes $|i| \in \{10, 50, 100, 150, 200, 240\} * 10^4$.

in the range of 6 ($k = 3$) to 57 ($k = 1$). We also observed that the 24-threaded outperforms the single-threaded execution in PeARL with factors in the range of 5.5($k = 1$) to 6.2 ($k = 3$).

For similarity joins, we could only compare the 24-threaded to the single-threaded execution in PeARL since no external reference implementation was available. Thus, we measured the execution times of III$\bowtie_{k\in\{1,2,3\}}$IV. As displayed in Fig. 6, we measured a runtime improvement of factors in the range of 4.2 ($k = 2$) to 4.9 ($k = 1$).

When analyzing the parallelized search and join algorithms in terms of speed-up, the first step is to estimate the fractions of parallelizable and non-parallelizable parts in our algorithms. In general, the parallelizable fractions dominate, since only reading the indices into main memory, extracting tasks from `mapJoinList`, sorting intermediate partitions before executing reduce, and writing the final output to file is performed in serial. We estimated the size of the parallelizable fraction based on the measured speed-up using $N = 24$ CPU cores. According to this, 10 % of our search and 20 % of our join algorithm remain serial.

Figure 7 displays the speed-up of searches of all ESTs from set I in the indexed set IV with regard to the number of CPU cores. We observed that the speed-up for measured runtimes almost perfectly fits the theoretical curve of Amdahl's law for $P = 0.90$. Similarly, we observed for joins that the measured speed-up fits well to Amdahl's law for $P = 0.80$ (data not shown). This indicates that estimating the non-parallelizable fraction with 10 % for searches and 20 % for joins is sound. Using 24 CPU cores with 24 map and reduce workers, we achieve a speed-up of our join algorithm of 4.3. According to that, the maximal speed-up for join is 4.9 using $\geq 1,000$ cores. This indicates that executing the

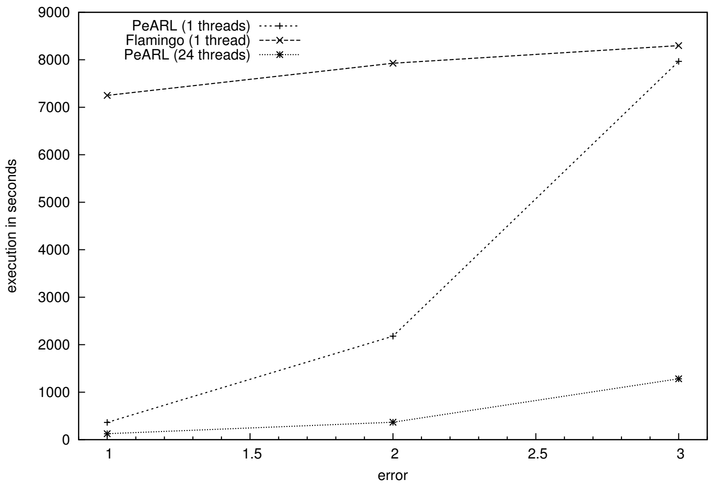

Fig. 5. Performance of parallelization for search: PeARL (24 threads) vs. Flamingo (1 thread).

current implementation of PeARL is limited by the serial parts contained in our algorithms.

5 Related Work

Morrison [12] introduced prefix trees as an index structure for storing strings and exact string matching. Shang et al. [16] extended prefix trees with dynamic programming techniques to perform inexact matching. Prefix pruning was studied in [16] and is based on the observation that edit distance can only grow with prefix length. Aghili et al. [1] proposed character frequency distance based filtering to reduce candidate sets for similarity-based string searches. Indexing methods based on q-grams restrict search spaces efficiently for edit distance based operations. They take advantage of the observation that two strings are within a small edit distance iff they share a large number of q-grams [17].

The MapReduce programming model for parallel data analysis was initially proposed by Dean and Ghemawat [5]. Vernica et al. [19] present an algorithm set-similarity string joins with distributed MapReduce. We could not compare to their solution, since no in-memory version was available. Ranger et al. [13] developed a MapReduce based programming framework for shared-memory multicore servers with a scalability almost reaching hand-coded solutions.

A main application for similarity-based string searches and joins in bioinformatics is read alignment. Almost all tools follow the seed-and-extend approach. BLAST [3] seeds the alignment with hash-table indices and extend the initially ungapped seeds with a banded local alignment algorithm. However, algorithms

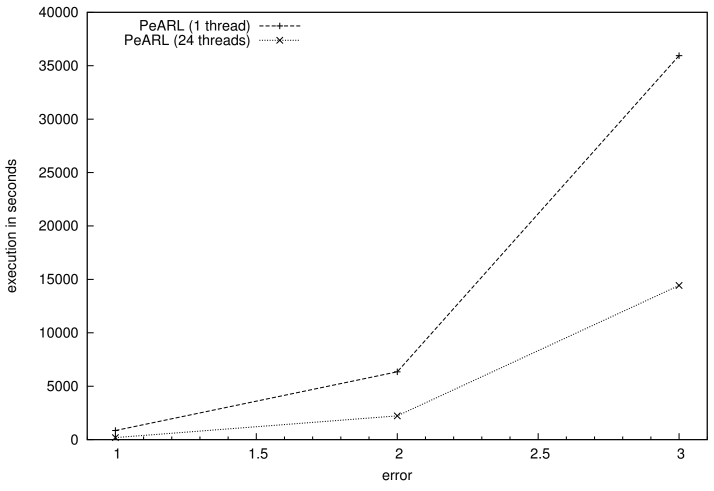

Fig. 6. Performance of parallelization for join: PeARL (24 threads) vs. PeARL (1 thread).

that use only ungapped seeds might miss some valuable alignments. Tools like BWA-SW [10] and RazerS [20] provide two approaches that allow gap and mismatches in the seeds. We also applied PeARL for read alignment and compared the execution times to BWA-SW and RazerS. Both tools significantly outperform PeARL (data not shown), but it must be noted that both are heuristics that miss solutions, while PEARL solves the alignment problem exactly. CloudBurst [15] is another another tool for read alignment using MapReduce on top of Hadoop [4]. A comparison between PEARL and CloudBurst is pending.

6 Conclusions and future work

In this paper, we presented PeARL, a data structure and parallel algorithms for similarity-based search and join operations in compressed tries. PeARL is parallelized in main memory with MapReduce on a multi-core server. Our evaluation revealed that the speed-up of our search and join algorithms executed on multi-core servers cannot grow infinitely large due to the serial parts contained in our workflow. We are currently working on reducing these bottlenecks and on performing a detailed comparison between PeARL and CloudBurst.

References

1. S. A. Aghili, D. Agrawal, and A. E. Abbadi. Bft: Bit filtration technique for approximate string joins in biological databases. In *Proc. 10th International Symposium on String Processing and Information Retrieval (SPIRE)*, volume 2857 of *Lecture Notes in Computer Science*, pages 326–340, 2003.

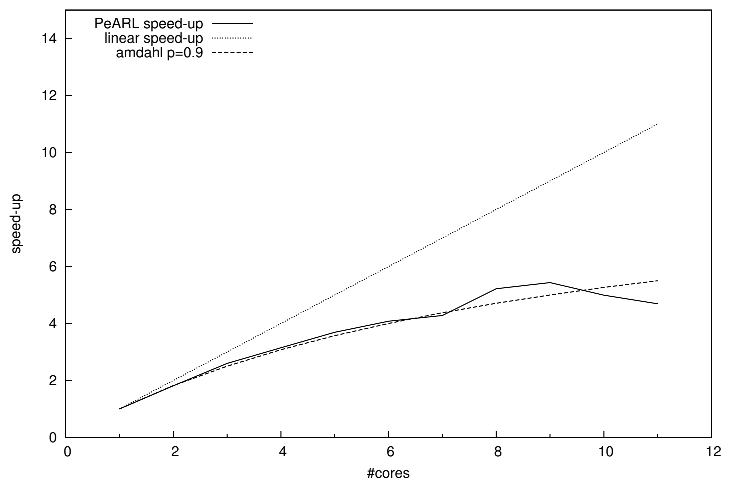

Fig. 7. PeARL speed-up for similarity search on k=2.

2. A. Alexandrov, M. Heimel, V. Markl, D. Battré, F. Hueske, E. Nijkamp, S. Ewen, O. Kao, and D. Warneke. Massively parallel data analysis with pacts on nephele. *Proc. VLDB Endow.*, 3:1625–1628, September 2010.
3. S. F. Altschul, W. Gish, W. Miller, E. W. Myers, and D. J. Lipman. Basic local alignment search tool. *Journal of Molecular Biology*, 215(3):403–410, 1990.
4. A. Bialecki, M. Cafarella, D. Cutting, and O. O'Malley. Hadoop. *http://hadoop.apache.org/.*
5. J. Dean and S. Ghemawat. MapReduce: simplified data processing on large clusters. *Communications of the ACM*, 51(1):107, 2008.
6. J. W. Fickett. Fast optimal alignment. *Nucleic Acids Research*, 12(1Part1):175–179, 1984.
7. L. Gravano, P. G. Ipeirotis, H. V. Jagadish, N. Koudas, S. Muthukrishnan, and D. Srivastava. Approximate string joins in a database (Almost) for free. In *Proc. 27th International Conference on Very Large Data Bases (VLDB)*, pages 491–500, 2001.
8. T. H. Haveliwala, A. Gionis, D. Klein, and P. Indyk. Evaluating strategies for similarity search on the web. In *Proceedings of the 11th international conference on World Wide Web*, WWW '02, pages 432–442, 2002.
9. T. C. Hoad and J. Zobel. Methods for identifying versioned and plagiarized documents. *J. American Society for Information Science and Technology*, 54:203–215, February 2003.
10. H. Li and R. Durbin. Fast and accurate long-read alignment with Burrows–Wheeler transform. *Bioinformatics*, 26(5):589–595, 2010.
11. H. Li and N. Homer. A survey of sequence alignment algorithms for next-generation sequencing. *Briefings in Bioinformatics*, 11(5):473 –483, 2010.
12. D. R. Morrison. PATRICIA – practical algorithm to retrieve information coded in alphanumeric. *Journal of the ACM*, 15(4):514–534, 1968.
13. C. Ranger, R. Raghuraman, A. Penmetsa, G. R. Bradski, and C. Kozyrakis. Evaluating mapreduce for multicore and multiprocessor systems. In *Proc. 13st Inter-*

national Conference on High-Performance Computer Architecture (HPCA), pages 13–24. IEEE Computer Society, 2007.

14. A. Rheinländer, M. Knobloch, N. Hochmuth, and U. Leser. Prefix tree indexing for similarity search and similarity joins on genomic data. In *Proc. 22nd International Conference on Scientific and Statistical Database Management (SSDBM)*, volume 6187 of *Lecture Notes in Computer Science*, pages 519–536. Springer, 2010.
15. M. C. Schatz. Cloudburst. *Bioinformatics*, 25:1363–1369, June 2009.
16. H. Shang and T. H. Merrett. Tries for approximate string matching. *IEEE Transactions on Knowledge and Data Engineering*, 8(4):540–547, 1996.
17. E. Sutinen and J. Tarhio. Filtration with q-samples in approximate string matching. In *Proc. 7th Annual Symposium on Combinatorial Pattern Matching (CPM)*, volume 1075 of *Lecture Notes in Computer Science*, pages 50–63. Springer, 1996.
18. A. Vakali, J. Pokorn, and T. Dalamagas. An overview of web data clustering practices. In *Current Trends in Database Technology - EDBT 2004 Workshops*, volume 3268 of *Lecture Notes in Computer Science*, pages 500–501, 2005.
19. R. Vernica, M. J. Carey, and C. Li. Efficient parallel set-similarity joins using mapreduce. In *Proceedings of the 2010 International Conference on Management of Data*, SIGMOD '10, pages 495–506, 2010.
20. D. Weese, A. Emde, T. Rausch, A. Döring, and K. Reinert. RazerS – fast read mapping with sensitivity control. *Genome Research*, 19(9):1646–1654, 2009.