Scalable Suffix Array Construction Using Map/Reduce

Aidan Fowler, Lavanya Sivakumar, Kartik Thapar

December 2, 2013

Suffix Arrays

- Suffix array: a lexicographically sorted array of all suffixes of a string.
- Suffix arrays are a simple and powerful data structure for text processing that can be used for full text indexes, data compression, string matching, genome analysis, and many other applications in particular in the field of computational biology.

Suffix Array Construction

- Algorithms we discussed in class For Building suffix array:
  1. Build a suffix tree $\rightarrow$ Do a lexicographical depth first traversal and report leaf offsets.
  2. Sort suffixes using your favorite sorting algorithm (quicksort) $\rightarrow O(m^2 logm)$, m^2 because suffix comparison takes $O(m)$ time.
- We can do better!

Prior Work

- Linear Work Suffix Array Construction (Juha Krkkinen, Peter Sanders, Stefan Burkhard): the original $O(n)$ time algorithm (DC3)
- Scalable parallel suffix array construction (Fabian Kulla, Peter Sanders): DC3 implementation using MPI
- The Performance of Linear Time Suffix Sorting Algorithms (Simon J. Puglisi, William F. Smyth, Andrew Turpin)

Motivation

- $\textsc{Space}(\text{Suffix Arrays}) \ll \textsc{Space}(\text{Suffix Trees})$
- The PDC3 algorithm using MPI, requires substantial low latency and high bandwidth interconnects that are not generally available.
- Hadoop Streaming using Python
  1. more compact and easy to understand than larger java implementation
  2. map/reduce algorithms exist for $O(n \lg n)$ time suffix array constructions (and other slow algorithms)

I. Construct A Sample

Reference Text: '*alphabeta*'

For $k = 0, 1, 2$, define:

$$B_k = \{i \in [0, n] | i \ mod \ 3 = k\}$$

- Sample positions: $C = B_1 \cup B_2$; Sample Suffixes: S_C
- Non-Sample positions: B_0; Non-Sample Suffixes: S_{B_0}
- $B_1 = \{1, 4, 7\}$, $B_2 = \{2, 5, 8\}$, $C = \{1, 4, 7, 2, 5, 8\}$

II. Sort Sample Suffixes

For $k = 1, 2$, define:

$$\begin{aligned} R_k &= [t_k t_{k+1} t_{k+2}][t_{k+3} t_{k+4} t_{k+5}] \ldots [t_{maxB_k} t_{maxB_{k+1}} t_{maxBk+2}] \\ R &= R_1 + R_2 = (lph, abe, ta0, pha, bet, a00) \end{aligned}$$

$\rightarrow$ Ordering in R gives an ordering for the sample suffixes S_C
$\Rightarrow$ Sort R
$\Rightarrow$ Assign ranks to sample suffixes
Sorted-$S_C = [8, 4, 5, 1, 2, 7]$
Suffix-Ranks $= [-, 4, 5, -, 2, 3, -, 6, 1, 0, 0, 0]$

III. Sort Non-Sample Suffixes

For suffix $S_i, S_j \in S_{B_0}$:

$$S_i \leq S_j \Leftrightarrow (t_i, \textsc{Rank}(S_i + 1)) \leq (t_j, \textsc{Rank}(S_j + 1))$$

Sorted-$S_{B_0} = [0, 6, 3]$

IV. Merge Sorted-S_C & Sorted-S_{B_0}

Two separate conditions for if the comparison is between $S_{B_1} - S_{B_0}$ or $S_{B_2} - S_{B_0}$.

$$i \in B_1 : S_i \leq S_j \Leftrightarrow (t_i, \text{RANK}(S_i + 1)) \leq (t_j, \text{RANK}(S_j + 1))$$
$$i \in B_2 : S_i \leq S_j \Leftrightarrow (t_i, t_{i+1}, \text{RANK}(S_{i+2})) \leq (t_j, t_{j+1}, \text{RANK}(S_{j+2}))$$

Result* $= [8, 4, 0, 5, 6, 3, 1, 2, 7]$

* – suffix '$' not included

Sorting Sample Suffixes - Creating R'

- **Mapper:**
  - Input: (_, ReferenceText); Output: (triples, _)
  - create triples from R ($R_1 + R_2$) and release (k, v) of type (triple, _)
- **Combiner:**
  - Input: (triple, occurrences); (_, tripleRankPair)
  - combine triples and rank them as they come to create a map of type $\{\text{triple} \Rightarrow \text{rank}\}$
  - number of occurrences points to duplicates if any
- **Reducer:**
  - Input: (_, tripleRankPairs); Output: ()
  - traverse original R and dictionary look up for triples and substitute with ranks to create R'

Example

Mapper:

- Input: alphabeta
- Output:
  `a00 [0]`
  `abe [0]`
  `bet [0]`
  `lph [0]`
  `pha [0]`
  `ta0 [0]`

Reducer:

- Output: [4, 2, 6, 5, 3, 1]

Combiner:

- Input: `a00 [0], ..., ta0 [0]`
- Output:
  TripleRankPairs: `[a00, 1], [abe, 2], [bet, 3], [lph, 4], [pha, 5], [ta0, 6]`
  TripleRankMap: `{ta0: 6, abe: 2, pha: 5, lph: 4, a00: 1, bet: 3}`

Sorted Suffixes - S_C, Suffix Ranks & S_{B_0}

- **Mapper:**
  - Input: (_, R') Output: (textRankPair, Index)
  - create the sorted-S_C using sets B_1 and B_2 for the specific depth in the recursion tree using either R' or the suffix array of R' incase R' has duplicates
  - rank suffixes in a suffix ranks array to obtain sorted S_{B_0} in in the next step
- **Combiner**
  - Input: (textRank, indices), Output: (_, textRankIndex)
  - `textRankPair` is always unique as the rank is unique for every offset in the reference text
  - The sorted text rank pairs are received in the sorted order to obtain the ordering for sorted S_{B_0}

Example

Mapper:

- Input: `[4, 2, 6, 5, 3, 1]`
- Working:
  Sorted-S_C = `[8, 4, 5, 1, 2, 7]`
  SuffixRanks = `[-, 4, 5, -, 2, 3, -, 6, 1, 0, 0, 0]`
- Output:
  `[a, 4]` `[0]`
  `[e, 6]` `[6]`
  `[h, 2]` `[3]`

Combiner & Reducer - Create Sorted-S_{B_0}

Sorted-S_{B_0} = `[0, 6, 3]`

Sorted Suffixes & Merge

- **Reducer:**
  - Input: (_, textRankIndexTriple) Output: ()
  - create the sorted S_{B_0} array
- **Merge**
  - Input: (textRankPair, indices); Output = ()
  - compare the first suffixes of both sets and find the smaller according to the comparison function defined if the first suffix in S_C belongs to S_{B_1} or S_{B_2}

Example

- Merge Step:
  Sorted-S_{B_0} = [0, 6, 3]
  Sorted-S_C = [8, 4, 5, 1, 2, 7]
- Comparison:

(a, 0, 0)	8	(a, 1, 5)	0
(a, b, None)	4	(a, 1, 5)	0
(b, 3)	5	(a, 4)	0
(b, 3)	5	(e, 6)	6
(1, 4)	1	(e, 6)	6
(1, 4)	1	(h, 2)	3

- Suffix Array* = [8, 4, 0, 5, 6, 3, 1, 2, 7]

* – suffix '$' not included

Remarks – Research

- Investigated a lot of time to assessing the feasability of Map/Reduce in the project.
- Peers investigated other slower algorithms as recursive algorithms were harder to implemented in Map/Reduce.
- Implemented the PDC3 algorithm before DC3 but stuck at the last state (merge) as we were unable to obtain the sorted union set from the given sets as described in the algorithm.
- Also implemented linear DC3 (non-recursive) to ensure the authenticity of data at every stage.

Remarks – About Map/Reduce

- As it turns out Map/Reduce isn't ideal for iterative algorithms, especially if done in the traditional streaming mode.
- Although map/reduce supports tail recursion, iterative algorithms incur issues as map/reduce does not hold state.
- We bypassed the constraints by using the creating two map/reduce jobs.
  1. The first job processes all different R' strings and stores the data required to process the suffix array for each one of them.
  2. The second job unfolds the recursion by recursing up the recursion tree from the maximum depth to the depth with original reference text.

Future Work

- Currently, we are working on benchmarking our algorithm for large inputs and will provide scalability results in the report.
- We are also looking into other Python MR frameworks, such as Pydoop, to see if this increases performance.
- In the future:
  - Investigate ways of minimizing data stored at every depth in the recursion tree.
  - Investigate parallelization of the merge step by implementing combined set sorts.
  - Investigation and possible use of Apache Spark instead of Hadoop, which is a desired and faster architecture for iterative algorithms.

Conclusion

- We were able to create a suffix array by implementing the PDC3 recursive algorithm.
- We used the python library MRJob to implement Hadoop Streaming jobs.