A Simple Parallel Cartesian Tree Algorithm and its Application to Parallel Suffix Tree Construction

Julian Shun and Guy Blelloch
Motivation for Suffix Trees

• To efficiently search for patterns in large texts
  – Example: Bioinformatic applications

• Suffix trees allow us to do this
  – $O(N)$ work for construction with $O(M)$ work for search, where $N$ is the text size and $M$ is the pattern size
    • In contrast, Knuth-Morris-Pratt’s algorithm takes $O(M)$ work for construction and $O(N)$ work for search
  – Other supported operations: longest common substring, maximal repeats, longest palindrome, etc.
  – There are sequential implementations but no parallel ones that are both theoretically and practically efficient

• We developed a new (practical) linear-work parallel algorithm and analyzed it experimentally
Outline: Suffix Array to Suffix Tree (in parallel)

- Suffix array + Longest Common Prefixes

  (interleave SA and LCPs)

- Multiway Cartesian tree

  (label edges, insert into hash table)

- Suffix tree

  • There are standard techniques to perform all of these steps in parallel, except for building the multiway Cartesian Tree
Suffix Arrays and Longest-common-prefixes (LCPs)

Original String
mississippi$

Suffixes
mississippi$
ississippi$
ssississippi$
sissippi$
issippi$
ssippi$
spipi$
ippipi$
pipi$
pipi$
i$
$

Suffix array
$
$i$
ippi$
issippi$
ississippi$
mississippi$
ipi$
ppi$
sippi$
ssippi$
ssippi$
ssissippi$

LCPs
0
1
4
0
0
1
2
3

Sort suffixes
Suffix Trees

- String = mississippi$
- Store suffixes in a patricia tree (trie with one-child nodes collapsed)
Multiway Cartesian Tree

- Maintains heap property
- Inorder traversal gives back the sequence
- Components of same value treated as one “cluster”

Sequence = 1 2 0 4 1 1 3 1 2
Suffix Tree History

- Sequential $O(n)$ work algorithms based on incrementally adding suffixes [Weiner ‘73, McCreight ‘76, Ukkonen ‘95]
- Parallel $O(n)$ work algorithms very complicated, no implementations [Sahinalp-Vishkin ‘94, Hariharan ‘94, Farach-Muthukrishnan ‘96]
- Parallel algorithms used in practice are not linear-work
- **Practical linear-work parallel algorithm?**
  - Simple $O(n)$ work parallel algorithm
  - Fastest algorithm in practice
More Related Work

- Cartesian trees
  - Sequential $O(n)$ work stack-based algorithm
  - Work-optimal parallel algorithm for Cartesian tree on distinct values (Berkman, Schieber and Vishkin 1993)

- Suffix arrays to suffix trees
  - Sequential $O(n)$ work algorithms
  - Two parallel algorithms for converting a suffix array into a suffix tree (Iliopoulos and Rytter 2004)
    - Both require $O(n \log n)$ work

- Our contributions
  - A parallel algorithm for converting suffix arrays to suffix trees, which requires only $O(n)$ work and is based on multiway Cartesian trees
Suffix Array/LCPs $\rightarrow$ Suffix Tree

- Interleave suffix lengths and LCP values
- Build a multiway Cartesian tree on that
- This returns the suffix tree!
String = mississippi$

SA + LCPs = 1, 0, 2, 1, 5, 1, 8, 4, 11, 0, 12, 0, 3, 1, 4, 0, 6, 2, 9, 1, 7, 3, 10
(interleaved)
Suffix Array to Suffix Tree (in parallel)

- Suffix array + Longest Common Prefixes
- (interleave SA and LCPs)
- Multiway Cartesian tree
  - (label edges, insert into hash table)
  - Suffix tree

Karkkainen and Sander’s algorithm
O(n) work and O(\log^2 n) span
Cartesian Tree (in parallel)

- Divide-and-conquer approach
- Merge spines of subtrees (represented as lists) together using standard techniques

\[
\text{SA + LCPs} = 1, 0, 2, 0, 5, 1, 8, 1, 11, 4, 12, 0, 3, 0, 4, 1, 6, 0, 9, 2, 8, 1, 7, 3, 10
\]
Cartesian Tree (in parallel)
Cartesian Tree (in parallel)

- Input: Array A[1...N]

```
Build(A[1...n]){
    if n < 2 return;
    else in parallel do:
        t1 = Build(A[1...n/2]);
        t2 = Build(A[(n/2)+1...n]);
    Merge(t1, t2);
}

Merge(t1, t2){
    R-spine = rightmost branch of t1;
    L-spine = leftmost branch of t2;
    use a parallel merge algorithm on R-spine and L-spine;
}
```
String = mississippi$

○ = Leaf node with suffix length
○ = Internal node with LCP value

$SA + LCPs = 1, 0, 2, 1, 5, 1, 8, 4, 11, 0, 12, 0, 3, 1, 4, 0, 6, 2, 9, 1, 7, 3, 10$
(interleaved)
String = mississippi$

〇 = Leaf node with suffix length 〣 = Internal node with LCP value

SA + LCPs = 1, 0, 2, 1, 5, 1, 8, 4, 11, 0, 12, 0, 3, 1, 4, 0, 6, 2, 9, 1, 7, 3, 10
(interleaved)
String = mississippi$

= Leaf node with suffix length  
= Internal node with LCP value

SA + LCPs = 1, 0, 2, 1, 5, 1, 8, 4, 11, 0, 12, 0, 3, 1, 4, 0, 6, 2, 9, 1, 7, 3, 10 (interleaved)
String = mississippi$

Circle = Leaf node with suffix length

Circle with line = Internal node with LCP value

$SA + LCPs = 1, 0, 2, 1, 5, 1, 8, 4, 11, 0, 12, 0, 3, 1, 4, 0, 6, 2, 9, 1, 7, 3, 10$

(interleaved)
String = mississippi$

= Leaf node with suffix length

= Internal node with LCP value

SA + LCPs = 1, 0, 2, 1, 5, 1, 8, 4, 11, 0, 12, 0, 3, 1, 4, 0, 6, 2, 9, 1, 7, 3, 10 (interleaved)
String = mississippi$

= Leaf node with suffix length  = Internal node with LCP value

SA + LCPs = 1, 0, 2, 1, 5, 1, 8, 4, 11, 0, 12, 0, 3, 1, 4, 0, 6, 2, 9, 1, 7, 3, 10
(interleaved)
String = mississippi$

SA + LCPs = 1, 0, 2, 1, 5, 1, 8, 4, 11, 0, 12, 0, 3, 1, 4, 0, 6, 2, 9, 1, 7, 3, 10 (interleaved)
Cartesian Tree (in parallel)

- Almost all merged nodes will never be processed again (they are “protected”)

```
Charge to merge
```

```
Processed
```

```
Protected
```

- Left subtree
- Right subtree
- Left spine
- Right spine
- Charge to merge
- Left spine (right tree)
- Right spine (right tree)
- Left spine (left tree)
- Right spine (left tree)
- Left spine (right tree)
- Right spine (right tree)
Cartesian Tree - Complexity bounds

- Observation: All nodes processed, except for two, become protected during a merge.
- Charge the processing of those two nodes to the merge itself (there are only $2n-1$ merges). Other nodes pay for themselves and then get protected.
  - It is important that when one spine has been completely processed, the merge does not process the rest of the other spine, otherwise we get $O(n \log n)$ work.
- Therefore, the merges contribute a total of $O(n)$ work to the algorithm.
Cartesian Tree - Complexity bounds

- Maintain binary search trees for each spine so that the endpoint of the merge can be found efficiently (in $O(\log n)$ work and span)
- A parallel merge takes linear work and $O(\log n)$ span
- Merges contribute $O(n)$ work, and searches and binary tree maintenance in the spine cost $O(\log n)$ work per merge
  - $W(n) = 2W(n/2) + O(\log n) = O(n)$
- Span: $O(\log n)$ levels of recursion, and merges + binary search tree operations take $O(\log n)$ span
  - $S(n) = S(n/2) + O(\log n) = O(\log^2 n)$
Multiway Cartesian Tree - Complexity bounds

- To obtain multiway Cartesian tree, use parallel tree contraction to contract adjacent nodes with the same value.
- This can be done in $O(n)$ work and $O(\log n)$ span, which is within our bounds.
- We have a $O(n)$ work and $O(\log^2 n)$ span algorithm for constructing a multiway Cartesian tree.
Parallel Cartesian Tree Code

```c
struct node { node* parent; int value; }

void merge(node* left, node* right) {
    node* head;
    if (left->value > right->value) {
        head = left; left = left->parent;
    } else {head = right; right= right->parent;}

    while(1) {
        if (left == NULL) {head->parent = right; break;}
        if (right == NULL) {head->parent = left; break;}
        if (left->value > right->value) {
            head->parent = left; left = left->parent;
        } else {head->parent = right; right = right->parent;}
        head = head->parent;
    }

    void cartesianTree(node* Nodes, int n) {
        if (n < 2) return;
        cilk_spawn cartesianTree(Nodes, n/2);
        cartesianTree(Nodes+n/2, n-n/2);
        cilk_sync;
        merge(Nodes+n/2-1, Nodes+n/2);
    }
```
Suffix Array to Suffix Tree (in parallel)

Suffix array + Longest Common Prefixes

(interleave SA and LCPs)

Multiway Cartesian tree

(label edges, insert into hash table)

Suffix tree

Karkkainen and Sander’s algorithm
O(n) work and O(log² n) span

Our parallel merging algorithm
O(n) work and O(log² n) span

Parallel hash table
O(n) work and O(log n) span
Experimental Setup

• Implementations in Cilk Plus
• 40-core Intel Nehalem machine
• Inputs: real-world and artificial texts
Suffix Tree Experiments

- Compared to best sequential algorithm [Kurtz ‘99]

- Speedup varies from 5.4x to 50x on 40 cores
- Self-relative speedup 23x to 26x on 40 cores
- Differences due to various factors
  - Shared memory vs. distributed memory
  - Algorithmic differences
Conclusions

• Developed an $O(n)$ work and $O(\log^2 n)$ span algorithm for parallel multiway Cartesian Tree construction
• This allows us to transform a suffix array into a suffix tree in parallel
• Experiments show that our implementations outperform existing ones and achieve good speedup
Project Presentation

• Project presentations on Tuesday
  – 5 minutes per team member, and 5 minutes for Q&A
  – Problem and motivation
  – Prior work
  – Your technical contributions
  – Challenges encountered
  – Experimental results
  – Work breakdown among team members

• Project report due on Tuesday
Course Summary

• Congratulations on making it through all the lectures!
• Lots of exciting research going on in algorithm and performance engineering
• Look out for relevant seminars
  – CSAIL seminars mailing list: seminars@csail.mit.edu
• Relevant conferences: SPAA, APOCS, PPoPP, ALENEX, ESA, SEA, PODC, IPDPS, SC, VLDB, SIGMOD, and more