# ERA revisited: Theoretical and Experimental evaluation

Matevž Jekovec, Andrej Brodnik

University of Ljubljana
Faculty of Computer and Information Science

KAUST, March 1-5, 2014

# Text indexing problem

# Text indexing problem

## Problem statement

Given unstructured input string $S$ consisting of $N$ characters from alphabet $\Sigma$ of size $\sigma$ build an index such that for the pattern $P$ we:

- determine whether $P$ occurs in $S$ in time $O(P)$,
- find all occurrences of $P$ in $S$ in time $O(P + occ)$,
- find the longest common prefix (LCP) of $P$ and any suffix of $S$ in time $O(LCP(P, S))$.

# Text indexing problem

## Problem statement

Given unstructured input string $S$ consisting of $N$ characters from alphabet $\Sigma$ of size $\sigma$ build an index such that for the pattern $P$ we:

- determine whether $P$ occurs in $S$ in time $O(P)$,
- find all occurrences of $P$ in $S$ in time $O(P + occ)$,
- find the longest common prefix (LCP) of $P$ and any suffix of $S$ in time $O(LCP(P, S))$.

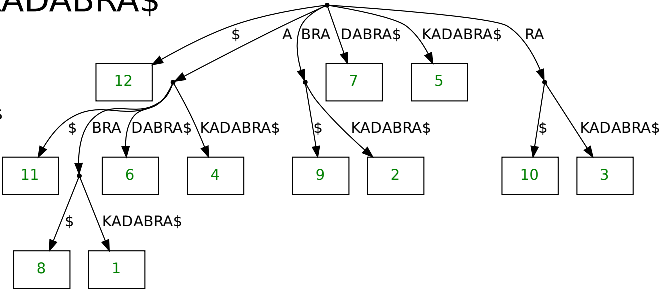
## Solution

*Suffix tree* and *suffix array* (SA) with LCP information are fundamental data structures for indexing unstructured text.

# Suffix tree — Example

1 2 3 4 5 6 7 8 9 10 11 12
T=ABRAKADABRA$

1 $
2 A$
3 ABRA$
4 ABRAKADABRA$
5 ADABRA$
6 AKADABRA$
7 BRA$
8 BRAKADABRA$
9 DABRA$
10 KADABRA
11 RA$
12 RAKADABRA$



# Suffix tree construction algorithms

- Theoretical:

| | W ('73), McC ('78) | U ('95) | F-C et al. ('00) |
|---|---|---|---|
| Work w.c. | $O(N)$ | $O(N)$ | $O(N \lg N)$ |
| Online | No | Yes | Yes[1] |
| I/O efficiency | String | String | Result+String |
| Unbounded $\Sigma$ | No | No | Yes |
| Parallel | No | No | PDAM |

[1] Bedathur and Haritsa (2004)

# Suffix tree construction algorithms

- Theoretical:

| | W ('73), McC ('78) | U ('95) | F-C et al. ('00) |
|---|---|---|---|
| Work w.c. | $O(N)$ | $O(N)$ | $O(N \lg N)$ |
| Online | No | Yes | Yes[1] |
| I/O efficiency | String | String | Result+String |
| Unbounded $\Sigma$ | No | No | Yes |
| Parallel | No | No | PDAM |

- Practical:

| | Semi-disk-based | | | Out-of-core | | |
|---|---|---|---|---|---|---|
| | TDD ('04) | TRLS. ('07) | $B^2ST$ ('09) | WF ('09) | ERA ('11) | PCF ('13) |
| Work w.c. | $O(N^2)$ | $O(N^2)$ | $O(N^2)$ | $O(N^2)$ | $O(N^2)$ | $O(\sqrt{p}N)$ |
| I/O eff. | R. | R. | R.+S. | R.+S. | R.+S. | R.+S. |
| Unbnd. $\Sigma$ | No | No | No | No | No | No |
| Parallel | No | No | No | Yes | Yes | Yes |

[1] Bedathur and Haritsa (2004)

# Suffix tree construction algorithms

- Theoretical:

| | W ('73), McC ('78) | U ('95) | F-C et al. ('00) |
|---|---|---|---|
| Work w.c. | $O(N)$ | $O(N)$ | $O(N \lg N)$ |
| Online | No | Yes | Yes[1] |
| I/O efficiency | String | String | Result+String |
| Unbounded $\Sigma$ | No | No | Yes |
| Parallel | | | PDAM |

Huge gap between the theoretical and practical results!

- Practical:

| | Semi-disk-based | | | Out-of-core | | |
|---|---|---|---|---|---|---|
| | TDD ('04) | TRLS. ('07) | B$^2$ST ('09) | WF ('09) | ERA ('11) | PCF ('13) |
| Work w.c. | $O(N^2)$ | $O(N^2)$ | $O(N^2)$ | $O(N^2)$ | $O(N^2)$ | $O(\sqrt{p}N)$ |
| I/O eff. | R. | R. | R.+S. | R.+S. | R.+S. | R.+S. |
| Unbnd. $\Sigma$ | No | No | No | No | No | No |
| Parallel | No | No | No | Yes | Yes | Yes |

[1] Bedathur and Haritsa (2004)

# Suffix tree construction lower bounds

- Sequential:

| | bounded Σ | unbounded Σ |
|---|---|---|
| Time | $\Omega(Sort(N))$ | $\Omega(Sort(N))$ |
| I/Os[2] | $\Omega(Sort(N))$ | $\Omega(Sort(N))$ |
| Space[3] | $\Omega(N \lg \sigma)$ bits | $\Omega(N \lg \sigma)$ bits |

[2] EM model
[3] Uncompressed index in word RAM
[4] PEM model

# Suffix tree construction lower bounds

- Sequential:

| | bounded $\Sigma$ | unbounded $\Sigma$ |
|---|---|---|
| Time | $\Omega(Sort(N))$ | $\Omega(Sort(N))$ |
| I/Os[2] | $\Omega(Sort(N))$ | $\Omega(Sort(N))$ |
| Space[3] | $\Omega(N \lg \sigma)$ bits | $\Omega(N \lg \sigma)$ bits |

- Parallel on $p$ processing units:

| | bounded $\Sigma$ | unbounded $\Sigma$ |
|---|---|---|
| Parallel time | $\Omega(Sort_p(N))$ | $\Omega(Sort_p(N))$ |
| Parallel I/Os[4] | $\Omega(Sort_p(N))$ | $\Omega(Sort_p(N))$ |
| Space[3] | $\Omega(N \lg \sigma)$ bits | $\Omega(N \lg \sigma)$ bits |

[2] EM model

[3] Uncompressed index in word RAM

[4] PEM model

# Suffix tree construction lower bounds

- Sequential:

| | bounded $\Sigma$ | unbounded $\Sigma$ |
|---|---|---|
| Time | $\Omega(N)$ | $\Omega(N \log N)$ |
| I/Os[2] | $\Omega\left(\frac{N}{B}\right)$ | $\Omega\left(\frac{N}{B} log_{\frac{M}{B}} \frac{N}{B}\right)$ |
| Space[3] | $\Omega(N \lg \sigma)$ bits | $\Omega(N \lg \sigma)$ bits |

- Parallel on $p$ processing units:

| | bounded $\Sigma$ | unbounded $\Sigma$ |
|---|---|---|
| Parallel time | $\Omega\left(\frac{N}{p}\right)$ | $\Omega\left(\frac{N}{p} \log N\right)$ |
| Parallel I/Os[4] | $\Omega\left(\frac{N}{pB}\right)$ | $\Omega\left(\frac{N}{pB} log_{\frac{M}{B}} \frac{N}{B}\right)$ |
| Space[3] | $\Omega(N \lg \sigma)$ bits | $\Omega(N \lg \sigma)$ bits |

[2] EM model

[3] Uncompressed index in word RAM

[4] PEM model

# Challenges

# Challenges

Theoretical algorithms: Lack of locality of reference.

# Challenges

Theoretical algorithms: Lack of locality of reference.

Goal: Use scans only both for input text and the resulting suffix tree!

# Challenges

Theoretical algorithms: Lack of locality of reference.

Goal: Use scans only both for input text and the resulting suffix tree!

Counterintuitive: Input text is arbitrary, suffix tree is lexicographically ordered.

# Challenges contd.

I/O efficient solution (eg. WF-ERA, $B^2$ST-PCF):

1. Scan part of the input text from the disk,

# Challenges contd.

I/O efficient solution (eg. WF-ERA, $B^2$ST-PCF):

1. Scan part of the input text from the disk,
2. do in-memory sorting (=random accesses in fast memory only!),

# Challenges contd.

I/O efficient solution (eg. WF-ERA, B$^2$ST-PCF):

1. Scan part of the input text from the disk,
2. do in-memory sorting (=random accesses in fast memory only!),
3. construct the corresponding part of the suffix tree,

# Challenges contd.

I/O efficient solution (eg. WF-ERA, $B^2$ST-PCF):

1. Scan part of the input text from the disk,
2. do in-memory sorting (=random accesses in fast memory only!),
3. construct the corresponding part of the suffix tree,
4. glue parts together,

# Challenges contd.

I/O efficient solution (eg. WF-ERA, $B^2$ST-PCF):

1. Scan part of the input text from the disk,
2. do in-memory sorting (=random accesses in fast memory only!),
3. construct the corresponding part of the suffix tree,
4. glue parts together,
5. and contiguously write it to disk.

# ERA — Elastic Range[5]

[5] Mansour, Allam, Skiadopoulos, Kalnis (2011)

# ERA — Elastic Range[5]

- The fastest practical, parallel suffix tree construction algorithm to date.

[5] Mansour, Allam, Skiadopoulos, Kalnis (2011)

# ERA — Elastic Range[5]

- The fastest practical, parallel suffix tree construction algorithm to date.
- Time complexity: $O(N^2)$ w.c. — for extremely skewed text!

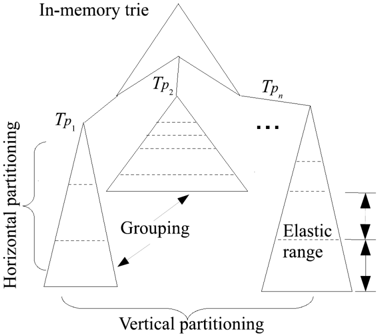
[5] Mansour, Allam, Skiadopoulos, Kalnis (2011)

# ERA — Elastic Range[5]

- The fastest practical, parallel suffix tree construction algorithm to date.
- Time complexity: $O(N^2)$ w.c. — for extremely skewed text!
- Yet, it's **fast** in practice: Constructs and stores the human genome's suffix tree in 20 minutes on 16-core desktop PC with HDD or 13 minutes with SSD!

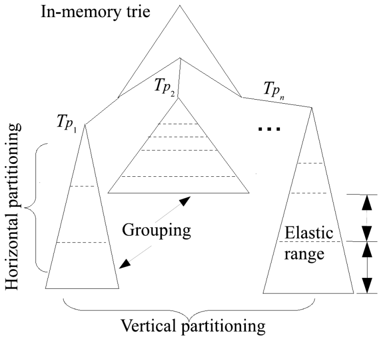
[5] Mansour, Allam, Skiadopoulos, Kalnis (2011)

# ERA contd.




ERA constructs the suffix tree in two steps:

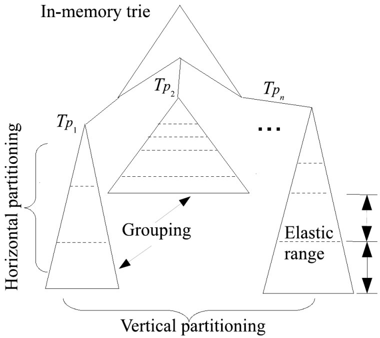
# ERA contd.




ERA constructs the suffix tree in two steps:

1. The **vertical partitioning** step determines 1) the suffix subtrees just fitting into the main memory $M$ and 2) constructs the suffix tree top.

# ERA contd.




ERA constructs the suffix tree in two steps:

1. The **vertical partitioning** step determines 1) the suffix subtrees just fitting into the main memory $M$ and 2) constructs the suffix tree top.
2. The **horizontal partitioning** step builds the actual suffix subtrees.

# ERA contd.

**Algorithm 1:** ERA

**Input**: String $S$, Alphabet $\Sigma$, Processors $P$, Private cache size $M$

**Output**: Suffix tree $\mathcal{T}$

```
T_top, G ← VerticalPartitioning(S, Σ, M)
T ← T_top
while |G| > 0 do
    for p ∈ P do in parallel
        if |G| > 0 then
            π ← G.pop()
            T_π ← HorizontalPartitioning(S, Σ, π)
            Link(T, T_π)
return T
```

# Vertical partitioning

Define **S-prefix** $\pi$ as the prefix of the suffixes in the text.

[6] Yue (1991)

# Vertical partitioning

Define **S-prefix** $\pi$ as the prefix of the suffixes in the text.
Idea: The S-prefix frequency $f_\pi$ equals # of leaves in the suffix subtree corresponding to $\pi$. Assume $2f_\pi$ is the w. c. subtree size.

[6] Yue (1991)

# Vertical partitioning

Define **S-prefix** $\pi$ as the prefix of the suffixes in the text.
Idea: The S-prefix frequency $f_\pi$ equals # of leaves in the suffix subtree corresponding to $\pi$. Assume $2f_\pi$ is the w. c. subtree size.
Vertical partitioning steps:

1. Scan the text and obtain the characters frequency $f_\pi : \pi \in \Sigma$.
    - We counted all S-prefixes of length 1.

[6] Yue (1991)

# Vertical partitioning

Define **S-prefix** $\pi$ as the prefix of the suffixes in the text.
Idea: The S-prefix frequency $f_\pi$ equals # of leaves in the suffix subtree corresponding to $\pi$. Assume $2f_\pi$ is the w. c. subtree size.
Vertical partitioning steps:

1. Scan the text and obtain the characters frequency $f_\pi : \pi \in \Sigma$.
   - We counted all S-prefixes of length 1.
2. For each $\pi : f_\pi > M$, expand S-prefix with the right character and count the frequency of obtained S-prefixes (now length 2).

[6] Yue (1991)

# Vertical partitioning

Define **S-prefix** $\pi$ as the prefix of the suffixes in the text.
Idea: The S-prefix frequency $f_\pi$ equals # of leaves in the suffix subtree corresponding to $\pi$. Assume $2f_\pi$ is the w. c. subtree size.
Vertical partitioning steps:

1. Scan the text and obtain the characters frequency $f_\pi : \pi \in \Sigma$.
   - We counted all S-prefixes of length 1.
2. For each $\pi : f_\pi > M$, expand S-prefix with the right character and count the frequency of obtained S-prefixes (now length 2).
3. Repeat step two for S-prefixes of length $3, 4\ldots$, until all $\mathcal{T}_\pi$ just fit into the memory $M$.

[6] Yue (1991)

# Vertical partitioning

Define **S-prefix** $\pi$ as the prefix of the suffixes in the text.
Idea: The S-prefix frequency $f_\pi$ equals # of leaves in the suffix subtree corresponding to $\pi$. Assume $2f_\pi$ is the w. c. subtree size.
Vertical partitioning steps:

1. Scan the text and obtain the characters frequency $f_\pi : \pi \in \Sigma$.
   - We counted all S-prefixes of length 1.
2. For each $\pi : f_\pi > M$, expand S-prefix with the right character and count the frequency of obtained S-prefixes (now length 2).
3. Repeat step two for S-prefixes of length $3, 4 \ldots$, until all $\mathcal{T}_\pi$ just fit into the memory $M$.
4. Extra: To optimally fill the main memory, combine the S-prefixes into *virtual groups* $G$, fitting into the main memory as tight as possible.
   - Use First-Fit Decreasing heuristic for bin packing problem[6].

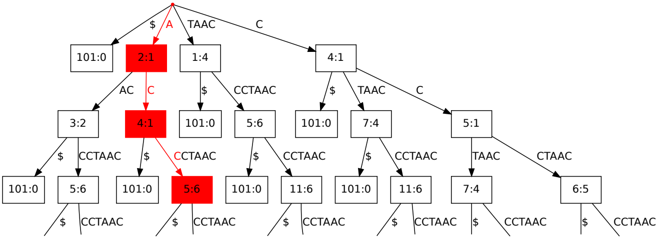
[6] Yue (1991)

# Vertical partitioning — Example

$\pi = ACC$

Frequency $f_{ACC} = 12$

TAACCCTA
ACCCTAAC
CCTAACCC
TAACCCTA
ACCCTAAC
CCTAACCC
TAACCCTA
ACCCTAAC
CCTAACCC
TAACCCTA
ACCCTAAC
CCTAACCC
TAAC



# Horizontal partitioning

For each virtual group, construct the corresponding suffix subtrees in parallel:

# Horizontal partitioning

For each virtual group, construct the corresponding suffix subtrees in parallel:

1. Locate and store all positions of S-prefixes for the virtual group. Each of located S-prefixes is to become a leaf in the working suffix subtree.

# Horizontal partitioning

For each virtual group, construct the corresponding suffix subtrees in parallel:

1. Locate and store all positions of S-prefixes for the virtual group. Each of located S-prefixes is to become a leaf in the working suffix subtree.
2. Calculate the optimal buffer length *range*
   - Note: The name *Elastic Range*.

# Horizontal partitioning

For each virtual group, construct the corresponding suffix subtrees in parallel:

1. Locate and store all positions of S-prefixes for the virtual group. Each of located S-prefixes is to become a leaf in the working suffix subtree.
2. Calculate the optimal buffer length *range*
   - Note: The name *Elastic Range*.
3. Read the next *range* characters for each S-prefix occurrence.

# Horizontal partitioning

For each virtual group, construct the corresponding suffix subtrees in parallel:

1. Locate and store all positions of S-prefixes for the virtual group. Each of located S-prefixes is to become a leaf in the working suffix subtree.
2. Calculate the optimal buffer length *range*
   - Note: The name *Elastic Range*.
3. Read the next *range* characters for each S-prefix occurrence.
4. Do in-memory sorting of read text, remember branching information (=LCP) and the original position (=SA).

# Horizontal partitioning

For each virtual group, construct the corresponding suffix subtrees in parallel:

1. Locate and store all positions of S-prefixes for the virtual group. Each of located S-prefixes is to become a leaf in the working suffix subtree.
2. Calculate the optimal buffer length *range*
   - Note: The name *Elastic Range*.
3. Read the next *range* characters for each S-prefix occurrence.
4. Do in-memory sorting of read text, remember branching information (=LCP) and the original position (=SA).
5. Until all the read buffers are unique, goto step 2.
   - In the next step: While less leafs are orphans, *range* increases, frequency drops.

# Horizontal partitioning

For each virtual group, construct the corresponding suffix subtrees in parallel:

1. Locate and store all positions of S-prefixes for the virtual group. Each of located S-prefixes is to become a leaf in the working suffix subtree.
2. Calculate the optimal buffer length *range*
   - Note: The name *Elastic Range*.
3. Read the next *range* characters for each S-prefix occurrence.
4. Do in-memory sorting of read text, remember branching information (=LCP) and the original position (=SA).
5. Until all the read buffers are unique, goto step 2.
   - In the next step: While less leafs are orphans, *range* increases, frequency drops.
6. Construct suffix subtree in D-F manner using SA and LCP.

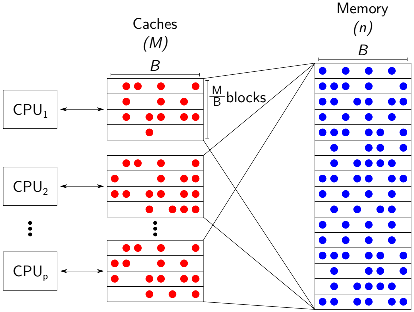
# Model of computation

[7] Arge, Goodrich, Nelson, Sitchinava 2008

# Model of computation

Parallel External Memory model (PEM):[7]

- Shared memory model,
- 2-level memory hierarchy:
  - $p$ processors, each with private cache of size $M$ bytes.
  - parallel memory transfers in blocks of size $B$ bytes.
- Performance metrics:
  - parallel time,
  - parallel block transfers (cache complexity).
- Concurrent reads assumed.




[7] Arge, Goodrich, Nelson, Sitchinava 2008

# Assumption

If worst-case input text for ERA is skewed:

$$T = AAA...$$

# Assumption

If worst-case input text for ERA is skewed:

$$T = AAA...$$

then

- vertical partitioning requires $N$ scans $= O(N^2)$ comparisons,
- cache complexity $O\left(\frac{N^2}{B}\right)$.

# Assumption

If worst-case input text for ERA is skewed:

$$T = AAA...$$

then

- vertical partitioning requires $N$ scans $= O(N^2)$ comparisons,
- cache complexity $O\left(\frac{N^2}{B}\right)$.

Our assumption:

- Input text is random (viable for a single human genome, proteins).
- At any place the probability of each character to occur is $\frac{1}{\sigma}$.
- Goal: Calculate expected time and cache complexity.

**Algorithm 2:** VerticalPartitioning

**Input**: Input string $S$, alphabet $\Sigma$, $1^{st}$ level memory size $M$
**Output**: Set of *VirtualTrees*

```
VirtualTrees ← ∅
P ← ∅
P' ← {∀ symbol s ∈ Σ generate a S-prefix π_i ∈ P'}
repeat
    scan input string S
    count in S the frequency f_{π_i} of every S-prefix π_i ∈ P'
    forall the π_i ∈ P' do
        if 0 < f_{π_i} ≤ M then add π_i to P
        else forall the symbol s ∈ Σ do add π_i s to P'
        remove π_i from P'
until P' = ∅
sort P in descending f_{π_i} order
repeat
    G ← ∅
    add P.head to G and remove the item from P
    curr ← next item in P
    while NOT end of P do
        if f_curr + SUM_{γ_i ∈ G}(f_{γ_i}) ≤ M then
            add curr to G and remove the item from P
        curr ← next item in P
    add G to VirtualTrees
until P = ∅
return VirtualTrees
```

# Analysis: Vertical partitioning

# Analysis: Vertical partitioning

Expected behavior:

1. Extension of S-prefixes:
   - Initially $\sigma$ S-prefixes of frequency $f_\pi = \frac{N}{\sigma}$ each.
   - $f_\pi$ divided by $\sigma$ each iteration until $f_\pi < M$.
   - Total $\log_\sigma N - \log_\sigma M = \log_\sigma \frac{N}{M}$ iterations.
   - Finally $\frac{N}{M}$ unique S-prefixes with frequency $\frac{M}{\sigma} < f_\pi \leq M$.

# Analysis: Vertical partitioning

Expected behavior:

1. Extension of S-prefixes:
   - Initially $\sigma$ S-prefixes of frequency $f_\pi = \frac{N}{\sigma}$ each.
   - $f_\pi$ divided by $\sigma$ each iteration until $f_\pi < M$.
   - Total $\log_\sigma N - \log_\sigma M = \log_\sigma \frac{N}{M}$ iterations.
   - Finally $\frac{N}{M}$ unique S-prefixes with frequency $\frac{M}{\sigma} < f_\pi \leq M$.
2. Atomic sorting the frequencies using one of the comparison-based sorting algorithms.

# Analysis: Vertical partitioning

Expected behavior:

1. Extension of S-prefixes:
   - Initially $\sigma$ S-prefixes of frequency $f_\pi = \frac{N}{\sigma}$ each.
   - $f_\pi$ divided by $\sigma$ each iteration until $f_\pi < M$.
   - Total $\log_\sigma N - \log_\sigma M = \log_\sigma \frac{N}{M}$ iterations.
   - Finally $\frac{N}{M}$ unique S-prefixes with frequency $\frac{M}{\sigma} < f_\pi \leq M$.
2. Atomic sorting the frequencies using one of the comparison-based sorting algorithms.
3. Virtual trees construction (bin packing problem):
   - At least 1 and at most $\sigma$ S-prefixes are packed each iteration.
   - External loop iterated between $\frac{N}{\sigma M}$ and $\frac{N}{M}$ times.

# Analysis: Vertical partitioning — Time

# Analysis: Vertical partitioning — Time

1. Extension of S-prefixes

$$\sum_{i=1}^{\log_\sigma \frac{N}{M}} \left(scan(n) + \sigma^{i+1}\right) = \log_\sigma \frac{N}{M} \cdot scan(n) + \frac{\sigma^2(N-M)}{M \cdot \sigma - M} = O\left(N \log_\sigma \frac{N}{M} + \frac{\sigma N}{M}\right)$$

# Analysis: Vertical partitioning — Time

1. Extension of S-prefixes

$$\sum_{i=1}^{\log_\sigma \frac{N}{M}} \left(scan(n) + \sigma^{i+1}\right) = \log_\sigma \frac{N}{M} \cdot scan(n) + \frac{\sigma^2(N-M)}{M\cdot\sigma - M} =$$

$$O\left(N \log_\sigma \frac{N}{M} + \frac{\sigma N}{M}\right)$$

2. Sorting

$$O\left(\frac{N}{M} \lg \frac{N}{M}\right)$$

# Analysis: Vertical partitioning — Time

1. Extension of S-prefixes

$$\sum_{i=1}^{\log_\sigma \frac{N}{M}} \left(scan(n) + \sigma^{i+1}\right) = \log_\sigma \frac{N}{M} \cdot scan(n) + \frac{\sigma^2(N-M)}{M \cdot \sigma - M} =$$

$$O\left(N \log_\sigma \frac{N}{M} + \frac{\sigma N}{M}\right)$$

2. Sorting

$$O\left(\frac{N}{M} \lg \frac{N}{M}\right)$$

3. Virtual trees construction

$$O\left(\left(\frac{N}{M}\right)^2\right)$$

# Analysis: Vertical partitioning — Time

1. Extension of S-prefixes
   $$\sum_{i=1}^{\log_\sigma \frac{N}{M}} \left(scan(n) + \sigma^{i+1}\right) = \log_\sigma \frac{N}{M} \cdot scan(n) + \frac{\sigma^2(N-M)}{M \cdot \sigma - M} =$$
   $$O\left(N \log_\sigma \frac{N}{M} + \frac{\sigma N}{M}\right)$$
2. Sorting
   $$O\left(\frac{N}{M} \lg \frac{N}{M}\right)$$
3. Virtual trees construction
   $$O\left(\left(\frac{N}{M}\right)^2\right)$$

Overall:

- If $\sigma < M$: $O\left(N \log_\sigma \frac{N}{M} + \left(\frac{N}{M}\right)^2\right)$
- If $\sigma \geq M$: $O\left(N \log_\sigma \frac{N}{M} + \frac{\sigma N}{M} + \frac{N}{M} \lg \frac{N}{M} + \left(\frac{N}{M}\right)^2\right)$

# Analysis: Vertical partitioning — I/O

1. Extension of S-prefixes
   - Line 6: $scan(N)$ for reading

# Analysis: Vertical partitioning — I/O

1. Extension of S-prefixes
    - Line 6: $scan(N)$ for reading
      $|P'| = O\left(\frac{N}{M}\right)$
      If $|P'| \leq M$: no I/Os for writing $f_\pi$
      If $|P'| > M$: $\frac{M}{|P'|} = \frac{M^2}{N}$ I/Os for storing $f_\pi$

# Analysis: Vertical partitioning — I/O

1. Extension of S-prefixes
   - Line 6: $scan(N)$ for reading
     $|P'| = O\left(\frac{N}{M}\right)$
     If $|P'| \leq M$: no I/Os for writing $f_\pi$
     If $|P'| > M$: $\frac{M}{|P'|} = \frac{M^2}{N}$ I/Os for storing $f_\pi$
   - Lines 7-10:
     If $|P'| \leq M$: no I/Os
     If $|P'| > M$: $\frac{P'}{B} = \frac{N}{M \cdot B}$ I/Os

# Analysis: Vertical partitioning — I/O

1. Extension of S-prefixes
   - Line 6: $scan(N)$ for reading
     $|P'| = O\left(\frac{N}{M}\right)$
     If $|P'| \leq M$: no I/Os for writing $f_\pi$
     If $|P'| > M$: $\frac{M}{|P'|} = \frac{M^2}{N}$ I/Os for storing $f_\pi$
   - Lines 7-10:
     If $|P'| \leq M$: no I/Os
     If $|P'| > M$: $\frac{P'}{B} = \frac{N}{M \cdot B}$ I/Os
2. Sorting $|P| = \frac{N}{M}$ elements:
   If $M \geq \sqrt{N}$: no I/Os
   If $M < \sqrt{N}$: $O(\frac{N}{M \cdot B} \log_{\frac{M}{B}} \frac{N}{M \cdot B})$ I/Os

# Analysis: Vertical partitioning — I/O

1. Extension of S-prefixes
   - Line 6: $scan(N)$ for reading
     $|P'| = O\left(\frac{N}{M}\right)$
     If $|P'| \leq M$: no I/Os for writing $f_\pi$
     If $|P'| > M$: $\frac{M}{|P'|} = \frac{M^2}{N}$ I/Os for storing $f_\pi$
   - Lines 7-10:
     If $|P'| \leq M$: no I/Os
     If $|P'| > M$: $\frac{P'}{B} = \frac{N}{M \cdot B}$ I/Os
2. Sorting $|P| = \frac{N}{M}$ elements:
   If $M \geq \sqrt{N}$: no I/Os
   If $M < \sqrt{N}$: $O(\frac{N}{M \cdot B} \log_{\frac{M}{B}} \frac{N}{M \cdot B})$ I/Os
3. Virtual tree $G \leq M$:
   $M \geq \sqrt{N}$: no I/Os
   $M < \sqrt{N}$: $\frac{|P|}{B} = \frac{N}{M \cdot B}$ I/Os

# Analysis: Vertical partitioning — I/O contd.

Overall:

- If $M \geq \sqrt{N}$:
  $O\left(\frac{N}{B} \log_\sigma \frac{N}{M}\right)$
- If $M < \sqrt{N}$:
  $O\left(\log_\sigma \frac{N}{M} \cdot \left(\frac{N}{B} + M^2\right) + \frac{N}{M \cdot B} \log_{\frac{M}{B}} \frac{N}{M \cdot B} + \left(\frac{N}{M \cdot B}\right)^2\right)$

**Algorithm 3:** HorizontalPartitioning.SubTreePrepare

**Input**: Input string $S$, S-prefix $\pi$

**Output**: Arrays $SA$ and $LCP$ corresponding suffix sub-tree $\mathcal{T}_\pi$

```
SA contains the locations of S-prefix π in string S
LCP ← {}
ISA ← {0, 1, ..., |SA| − 1}
A ← {0, 0, ..., 0}
Buf ← {}
P ← {0, 1, ..., |L| − 1}
start ← |π|
while there exists an undefined Buf[i], 1 ≤ i ≤ |SA| − 1 do
    range ← GetRangeOfSymbols
    for i ← 0 to |SA| − 1 do
        if ISA[i] ≠ done then
            Buf[ISA[i]] ← ReadRange(S, SA[ISA[i]] + start, range)
            // ReadRange(S,a,b) reads b symbols of S starting at
            position a
    for every active area AA do
        Reorder the elements of Buf, P and SA in AA so that Buf
        is lexicographically sorted. In the process maintain the
        index ISA
        If two or more elements {a_1, ..., a_t} ∈ AA, 2 ≤ t, exist such
        that Buf[a_1] = ... = Buf[a_i] introduce for them a new
        active area
    for all i such that Buf[i] is not defined, 1 ≤ i ≤ |SA| − 1 do
        cp is the common prefix of Buf[i − 1] and Buf[i]
        if |cp| < range then
            Buf[i] ← (Buf[i − 1][|cp|], Buf[i][|cp|], start + |cp|)
            if Buf[i − 1] is defined or i = 1 then
                Mark ISA[P[i − 1]] and A[i − 1] as done
            if Buf[i + 1] is defined or i = [SA] − 1 then
                Mark ISA[P[i]] and A[i] as done // last element of
                an active area
    start ← start + range
return (SA,LCP)
```

# Analysis: Horizontal partitioning

# Analysis: Horizontal partitioning

Expected behaviour:

# Analysis: Horizontal partitioning

Expected behaviour:

- Define $n$ the number of unfinished branches, then $n \cdot range = O(M)$.

# Analysis: Horizontal partitioning

Expected behaviour:

- Define $n$ the number of unfinished branches, then $n \cdot range = O(M)$.
- Intuitively $n$ decreases and *range* increases during execution of lines 8-24.

# Analysis: Horizontal partitioning

Expected behaviour:

- Define $n$ the number of unfinished branches, then $n \cdot range = O(M)$.
- Intuitively $n$ decreases and *range* increases during execution of lines 8-24.
- For length $k$, there can be at most $\sigma^k$ unique strings. For random text and step $1 \leq i \leq k$, strings are non-unique until $k$ is reached.

# Analysis: Horizontal partitioning

Expected behaviour:

- Define $n$ the number of unfinished branches, then $n \cdot range = O(M)$.
- Intuitively $n$ decreases and *range* increases during execution of lines 8-24.
- For length $k$, there can be at most $\sigma^k$ unique strings. For random text and step $1 \leq i \leq k$, strings are non-unique until $k$ is reached.
- If $O(M)$ random strings need to be processed, then lines 8-24 is iterated $O(\log_\sigma M)$ times. The big-oh constant depends on *range*.

# Analysis: Horizontal partitioning — Time

# Analysis: Horizontal partitioning — Time

Each iteration:

1. Lines 10-12 required $n$ time to fill the buffers (constant time read).

# Analysis: Horizontal partitioning — Time

Each iteration:

1. Lines 10-12 required $n$ time to fill the buffers (constant time read).
2. String sorting requires $O(n \cdot range)$ time since the average *distinguising prefix size* equals $O(range)$.

# Analysis: Horizontal partitioning — Time

Each iteration:

1. Lines 10-12 required $n$ time to fill the buffers (constant time read).
2. String sorting requires $O(n \cdot range)$ time since the average *distinguising prefix size* equals $O(range)$.
3. Lines 16-23 require $O(n \cdot range)$ time in the worst case.

# Analysis: Horizontal partitioning — Time

Each iteration:

1. Lines 10-12 required $n$ time to fill the buffers (constant time read).
2. String sorting requires $O(n \cdot range)$ time since the average *distinguising prefix size* equals $O(range)$.
3. Lines 16-23 require $O(n \cdot range)$ time in the worst case.

Overall: Assuming $p$ processors equally balanced after processing $O(N/M)$ virtual groups require

$$O\left(\frac{N}{M}\frac{M\log_\sigma M}{p}\right) = O\left(\frac{N}{p}\log_\sigma M\right)$$

# Analysis: Horizontal partitioning — I/O

# Analysis: Horizontal partitioning — I/O

1. Cache misses occur in lines 10-12 only:

# Analysis: Horizontal partitioning — I/O

1. Cache misses occur in lines 10-12 only:
   - If $n \geq \frac{N}{B}$, then $O\left(\frac{N}{B}\right)$ I/Os.

# Analysis: Horizontal partitioning — I/O

1. Cache misses occur in lines 10-12 only:
    - If $n \geq \frac{N}{B}$, then $O\left(\frac{N}{B}\right)$ I/Os.
    - Else: $O(n)$ I/Os

# Analysis: Horizontal partitioning — I/O

1. Cache misses occur in lines 10-12 only:
   - If $n \geq \frac{N}{B}$, then $O\left(\frac{N}{B}\right)$ I/Os.
   - Else: $O(n)$ I/Os
2. When does the change from $n \geq \frac{N}{B}$ to $n < \frac{N}{B}$ occur?

# Analysis: Horizontal partitioning — I/O

1. Cache misses occur in lines 10-12 only:
    - If $n \geq \frac{N}{B}$, then $O\left(\frac{N}{B}\right)$ I/Os.
    - Else: $O(n)$ I/Os
2. When does the change from $n \geq \frac{N}{B}$ to $n < \frac{N}{B}$ occur?
3. Assuming uniformly random text, $n = c \cdot M$ for some constant $c$ **all the time**! (all branches are open until the last iteration)

# Analysis: Horizontal partitioning — I/O

1. Cache misses occur in lines 10-12 only:
   - If $n \geq \frac{N}{B}$, then $O\left(\frac{N}{B}\right)$ I/Os.
   - Else: $O(n)$ I/Os
2. When does the change from $n \geq \frac{N}{B}$ to $n < \frac{N}{B}$ occur?
3. Assuming uniformly random text, $n = c \cdot M$ for some constant $c$ **all the time**! (all branches are open until the last iteration)
4. Suffix subtree construction from SA and LCP requires a single $scan(N)$ I/Os only and is omitted.

# Analysis: Horizontal partitioning — I/O

1. Cache misses occur in lines 10-12 only:
   - If $n \geq \frac{N}{B}$, then $O\left(\frac{N}{B}\right)$ I/Os.
   - Else: $O(n)$ I/Os
2. When does the change from $n \geq \frac{N}{B}$ to $n < \frac{N}{B}$ occur?
3. Assuming uniformly random text, $n = c \cdot M$ for some constant $c$ **all the time**! (all branches are open until the last iteration)
4. Suffix subtree construction from SA and LCP requires a single $scan(N)$ I/Os only and is omitted.
5. I/O complexity for horizontal partitioning is thus

$$O\left(min\left(M, \frac{N}{B}\right) \cdot \log_\sigma M\right)$$

# Wrap-up

# Wrap-up

Parallel time complexity of ERA (assuming $\sigma \leq M$):

$$O\left(N \log_\sigma \frac{N}{M} + \left(\frac{N}{M}\right)^2 + \frac{N}{p} \log_\sigma M\right)$$

Parallel cache complexity of ERA (assuming $M \geq \sqrt{N}$):

$$O\left(\frac{N}{B} \log_\sigma \frac{N}{M} + \frac{min\left(M, \frac{N}{B}\right) \cdot \log_\sigma M}{p}\right)$$

# Empirical evaluation

# Empirical evaluation

Testing environment:

- 2x 16-core AMD Opteron 6272 @2,100 MHz
- 128 GiB RAM
- Seagate Baracuda 250 GB, 7,200 RPM, 32 MiB cache, SATA
- Ubuntu server 12.04, Linux kernel 3.11.0
- ext4 file system, deadline I/O scheduler

ERA parameters:

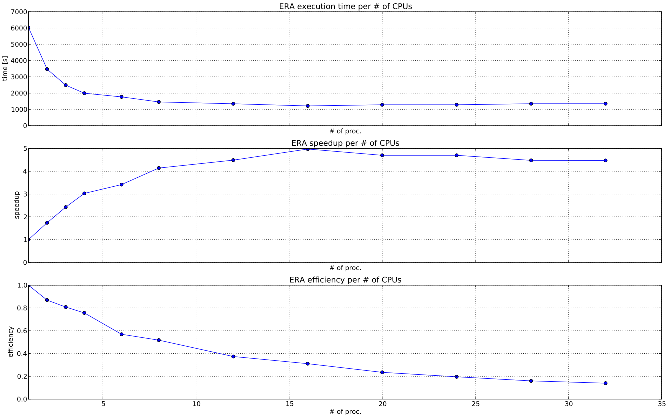
- Memory size per core: 2 GiB
- Input text: Human genome `HG18.txt`, 2.8 Gbp

ERA modification: Call `fsync()` after writing each file.

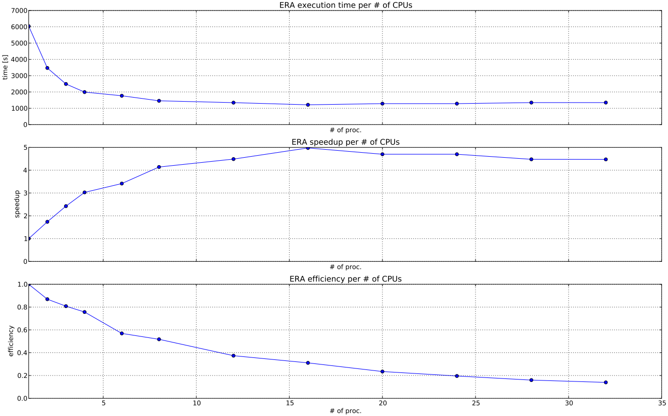
ERA output:

- Total suffix tree size: 77.3 GB stored in 187 files
- $\mathcal{T}_{top}$ size: 10.2 KB

# Results – 1


ERA execution time per # of CPUs
time [s]
# of proc.
ERA speedup per # of CPUs
speedup
# of proc.
ERA efficiency per # of CPUs
efficiency
# of proc.

# Results – 1




The time **increases** as we increase the number of cores.

# Results – 2

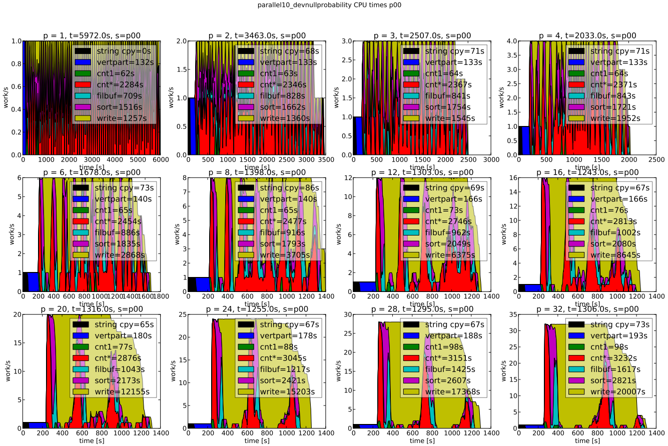
So what is the machine doing?

# Results – 2

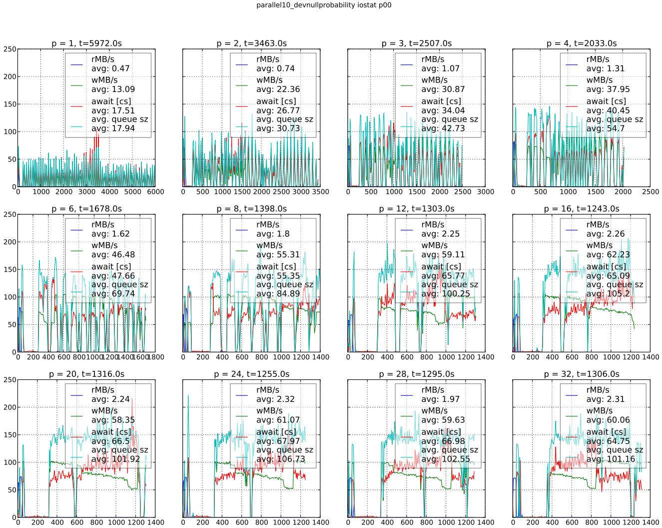
So what is the machine doing?

- **string cpy** Parsing and copying the string.
- **vertpart** Vertical partitioning.
- **cnt1, cnt*** Horizontal partitioning: determining locations of S-prefix in virtual trees of size 1 or $> 1$.
- **filbuf** Horizontal partitioning: reading range characters from S-prefix locations.
- **sort** Horizontal partitioning: string sorting, implicit LCP, SA construction.
- **write** Horizontal partitioning: extraction from LCP and SA to suffix tree, write to disk.

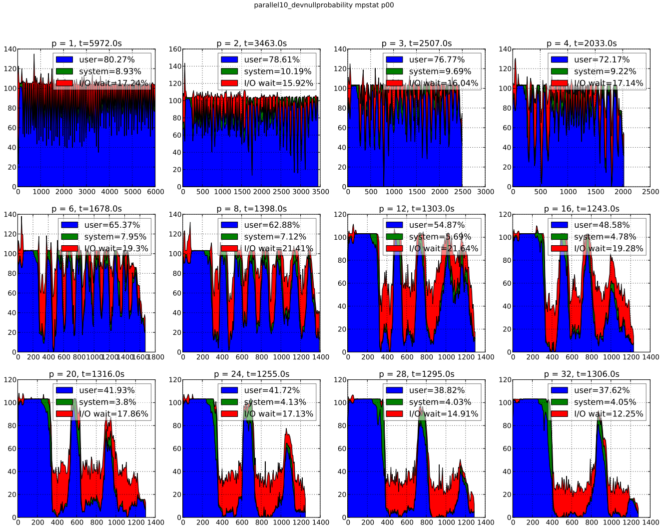
# Results – 2 contd.



# Results – 2 contd.



# Results – 2 contd.



# Hypotheis 1

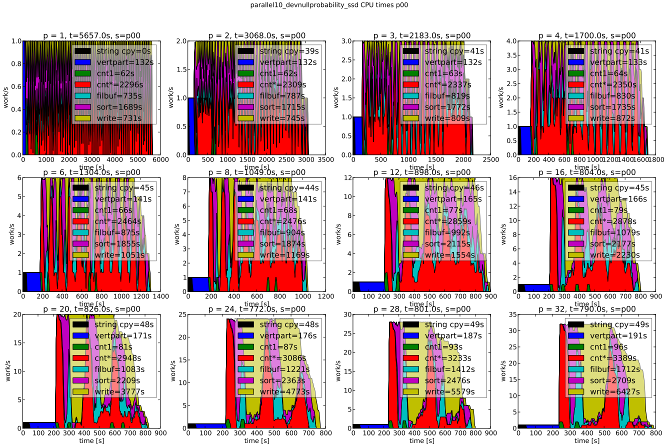
*Observation 1:* The majority of time is spent writing the final result to the disk.

# Hypotheis 1

*Observation 1:* The majority of time is spent writing the final result to the disk.

*Hypothesis 1:* Problem is the disk performance, so replace HDD with SSD.

# Results – 3


parallel10_devnullprobability_ssd CPU times p00
p = 1, t=5657.0s, s=p00
string cpy=0s
vertpart=132s
cnt1=62s
cnt*=2296s
filbuf=735s
sort=1689s
write=731s
p = 2, t=3068.0s, s=p00
string cpy=39s
vertpart=132s
cnt1=62s
cnt*=2309s
filbuf=787s
sort=1715s
write=745s
p = 3, t=2183.0s, s=p00
string cpy=41s
vertpart=132s
cnt1=63s
cnt*=2337s
filbuf=819s
sort=1772s
write=809s
p = 4, t=1700.0s, s=p00
string cpy=41s
vertpart=133s
cnt1=64s
cnt*=2350s
filbuf=830s
sort=1735s
write=872s
p = 6, t=1304.0s, s=p00
string cpy=45s
vertpart=141s
cnt1=66s
cnt*=2464s
filbuf=875s
sort=1855s
write=1051s
p = 8, t=1049.0s, s=p00
string cpy=44s
vertpart=141s
cnt1=68s
cnt*=2476s
filbuf=904s
sort=1874s
write=1169s
p = 12, t=898.0s, s=p00
string cpy=46s
vertpart=165s
cnt1=77s
cnt*=2859s
filbuf=992s
sort=2115s
write=1554s
p = 16, t=804.0s, s=p00
string cpy=45s
vertpart=166s
cnt1=79s
cnt*=2878s
filbuf=1079s
sort=2177s
write=2230s
p = 20, t=826.0s, s=p00
string cpy=48s
vertpart=171s
cnt1=81s
cnt*=2948s
filbuf=1083s
sort=2209s
write=3777s
p = 24, t=772.0s, s=p00
string cpy=48s
vertpart=176s
cnt1=87s
cnt*=3086s
filbuf=1221s
sort=2363s
write=4773s
p = 28, t=801.0s, s=p00
string cpy=49s
vertpart=187s
cnt1=93s
cnt*=3233s
filbuf=1412s
sort=2476s
write=5579s
p = 32, t=790.0s, s=p00
string cpy=49s
vertpart=191s
cnt1=96s
cnt*=3389s
filbuf=1712s
sort=2709s
write=6427s
work/s
time [s]

# Hypotheis 2

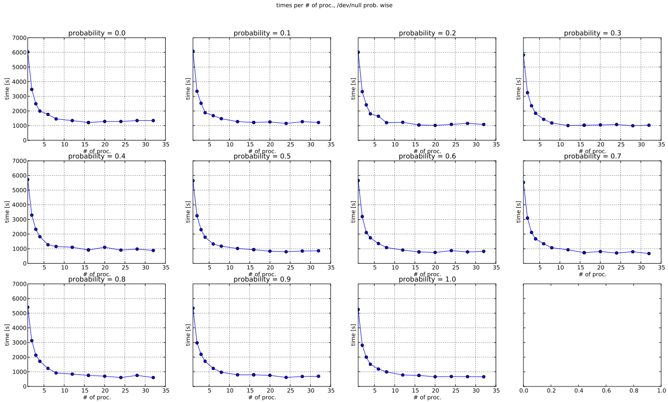
*Observation 2:* The amount of time for writting decreased, but as the number of cores grows, it is still substantial.

# Hypotheis 2

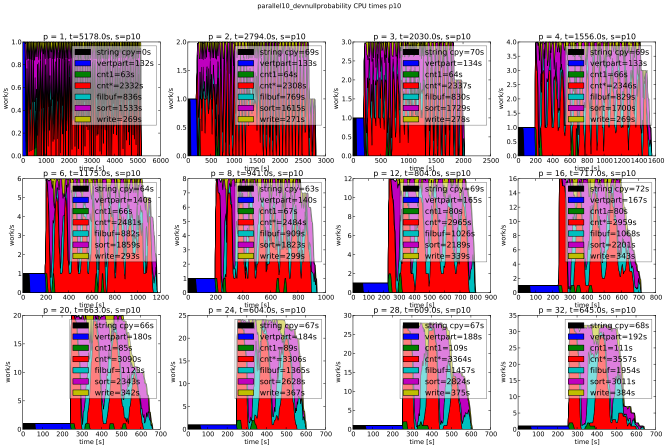
*Observation 2:* The amount of time for writting decreased, but as the number of cores grows, it is still substantial.

*Hypothesis 2:* There it is still a problem with a disk performance and consequently further speed-up disk by writting to `/dev/null`.

# Results – 4


times per # of proc., /dev/null prob. wise
probability = 0.0
probability = 0.1
probability = 0.2
probability = 0.3
probability = 0.4
probability = 0.5
probability = 0.6
probability = 0.7
probability = 0.8
probability = 0.9
probability = 1.0
time [s]
# of proc.

# Results – 4 contd.



# Hypotheis 3

*Observation 3:* Things are getting better, but there is still an increase in time when the number of cores is increased.

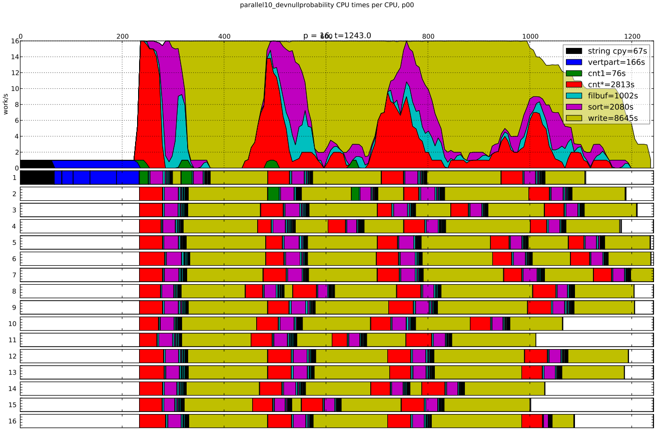
# Hypotheis 3

*Observation 3:* Things are getting better, but there is still an increase in time when the number of cores is increased.

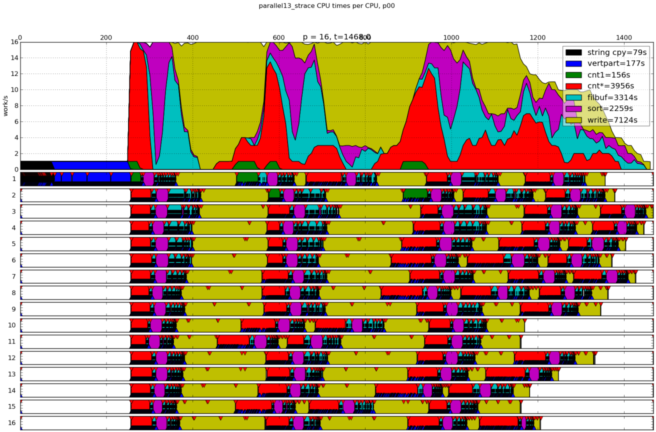
*Hypothesis 3:* ??

Check in more detail what the processes are doing.

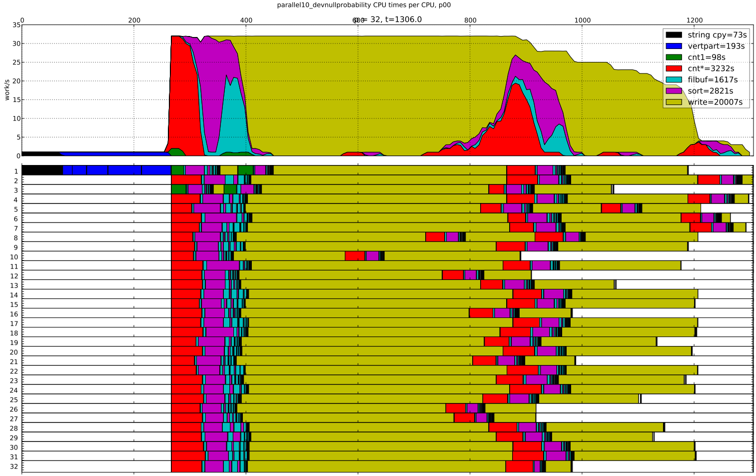
# Results – 5 ($p = 16$)



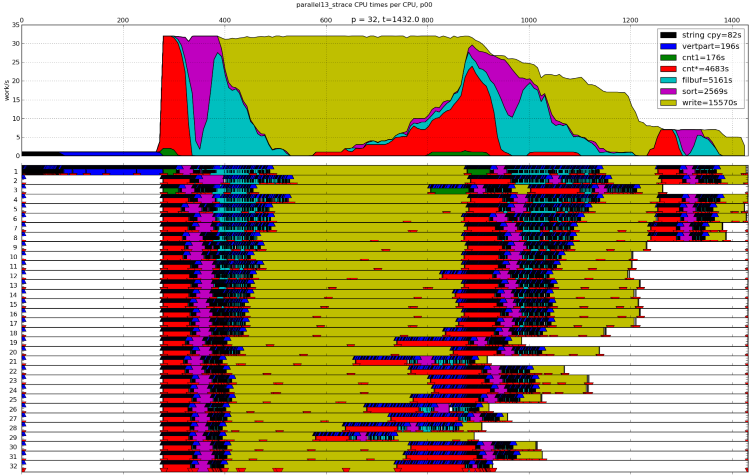
# Results – 5 ($p = 16$), strace



# Results – 5 ($p = 32$)



# Results – 5 ($p = 32$), strace



# Conclusion

# Conclusion

- Huge gap between the theoretical time and I/O asymptotically tight algorithms and the practical ones.

# Conclusion

- Huge gap between the theoretical time and I/O asymptotically tight algorithms and the practical ones.
- ERA despite being practically the fastest algorithm is **not theoretically tight** even for random input strings with uniform substring distribution.

# Conclusion

- Huge gap between the theoretical time and I/O asymptotically tight algorithms and the practical ones.
- ERA despite being practically the fastest algorithm is **not theoretically tight** even for random input strings with uniform substring distribution.

Open challenges:

# Conclusion

- Huge gap between the theoretical time and I/O asymptotically tight algorithms and the practical ones.
- ERA despite being practically the fastest algorithm is **not theoretically tight** even for random input strings with uniform substring distribution.

Open challenges:

- Analyse ERA bottlenecks for further improvements (see if they match the critical terms in time and I/O complexities).

# Conclusion

- Huge gap between the theoretical time and I/O asymptotically tight algorithms and the practical ones.
- ERA despite being practically the fastest algorithm is **not theoretically tight** even for random input strings with uniform substring distribution.

Open challenges:

- Analyse ERA bottlenecks for further improvements (see if they match the critical terms in time and I/O complexities).
- Shall we choose some other basic technique for the implementation of a practical algorithm?

# Conclusion

- Huge gap between the theoretical time and I/O asymptotically tight algorithms and the practical ones.
- ERA despite being practically the fastest algorithm is **not theoretically tight** even for random input strings with uniform substring distribution.

Open challenges:

- Analyse ERA bottlenecks for further improvements (see if they match the critical terms in time and I/O complexities).
- Shall we choose some other basic technique for the implementation of a practical algorithm?
- Design a theoretically tight yet practically competitive parallel algorithm for suffix tree construction.

# Thank you.

**Matevž Jekovec**
matevz.jekovec@fri.uni-lj.si

University *of Ljubljana*
Faculty *of Computer and Information Science*

**Andrej Brodnik**
andrej.brodnik@fri.uni-lj.si

LUSY
Laboratorij za vseprisotne sisteme
Laboratory for Ubiquitous SYstems
http://lusy.fri.uni-lj.si

# Extra — List of all experiments

Shown at the presentation:

- Execution time for different $p$, speed-up, efficiency.
- CPU times, `iostat` and `mpstat` per different phases for $p = \{1, 2, 3, 4, 6, 8, 12, 16, 20, 24, 27, 32\}$.
- `iostat` output for various $p$.
- `mpstat` output for various $p$.
- Work per core for single execution for $p = 32$.
- Using `strace`, fetching `read`, `write`, `lseek` syscalls.

# Extra — List of all experiments contd.

Test scenarios:

- Original code + added `fsync()`, various # of cores, various mem. size per core.
- Various string buffer sizes BUF_TYPE$= \{8, 16, 32, 64\}$ bit.
- Integration of Multikey cached quicksort (Rantala-Bentley-Sedgewick) instead of GNU `qsort`.
- Maximum limit of simultaneously opened files for writing $F = \{1, 2, 3, 4, 5, 6, 12, 16\}$.
- Output to `/dev/null` with probability $Pr = [0, 0.1 \ldots 1]$.
- Separated disk for writing and reading.
- SSD for reading and/or writing.
- Different file system schedulers: `noop`, `default`, `cfq`.
- Different file system max queue length.
- Output to raw device without file system.
- Execution on 12x Raspberry $\pi$ with shared NFS storage.