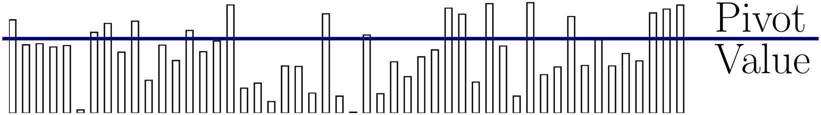
Cache-Efficient Parallel Partition Algorithms Using Exclusive-Read-and-Write Memory

William Kuszmaul, Alek Westover

July 1, 2020

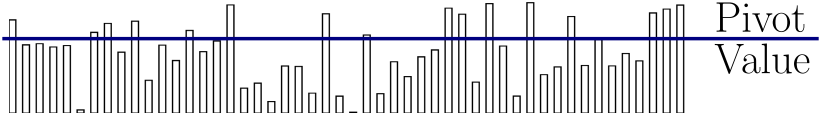
The Partition Problem

An unpartitioned array:

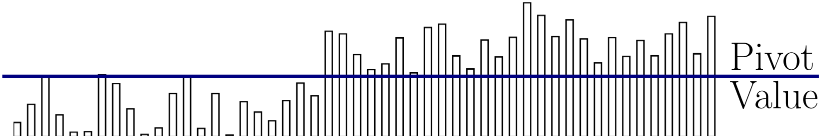


The Partition Problem

An unpartitioned array:



An array partitioned relative to a pivot value:

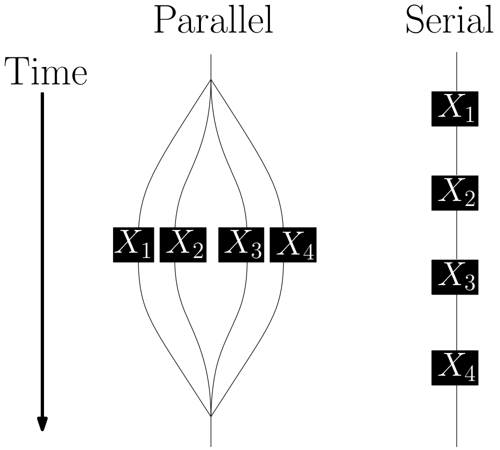


Example Applications

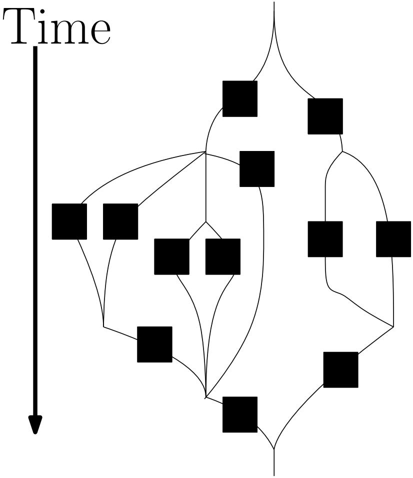
- Parallel Partition
- Parallel Quicksort
- Filtering Operations

WHAT IS A PARALLEL ALGORITHM?

Fundamental primitive: ***Parallel for loop***



Parallel Algorithm Performance Metrics



T_p: Time to run on p processors

Work: T_1

- time to run in serial

Span: T_∞

- time to run on infinitely many processors

Brent's Theorem:

$$T_p = \Theta\left(\frac{T_1}{p} + T_\infty\right)$$

What is Cache Efficiency?

Roughly, an algorithm is **cache-efficient** if it

- Performs low number of passes over the data
- Doesn't use extra memory (i.e. is *In-Place*)
- Opperates on elements close together in memory at the same time

Cache-Efficiency makes an algorithm faster in practice.

The Problem

Standard Algorithm span $O(\log n)$ but is slow in practice

- Uses extra memory
- Makes multiple passes over array

$\Big\}$ bad cache behavior

Fastest algorithms in practice lack theoretical guarantees

- Lock based and atomic-variable based algorithms

  [Michael Axtmann, Sascha Witt, Daniel Ferizovic, and Peter Sanders, 2017; Philip Heidelberger, Alan Norton, and John T. Robinson, 1990; Philippas Tsigas and Yi Zhang, 2003]

- The Strided Algorithm

  [Francis and Pannan, 92; Frias and Petit, 08]

  No locks or atomic-variables, but no bound on span

OUR QUESTION

Can we create an algorithm with *theoretical guarantees* that is *fast in practice*?

Our Result

The Smoothed-Striding Algorithm

Key Features:

- linear work and polylogarithmic span
  (like the Standard Algorithm)
- fast in practice
  (like the Strided Algorithm)
- theoretically optimal cache behavior
  (unlike any past algorithm)

STRIDED VERSUS SMOOTHED-STRIDING ALGORITHM

Strided Algorithm

[Francis and Pannan, 92; Frias and Petit, 08]

- Good cache behavior in practice
- Worst case span is $T_\infty \approx n$
- On random inputs span is $T_\infty = \tilde{O}(n^{2/3})$

Strided Versus Smoothed-Striding Algorithm

Strided Algorithm

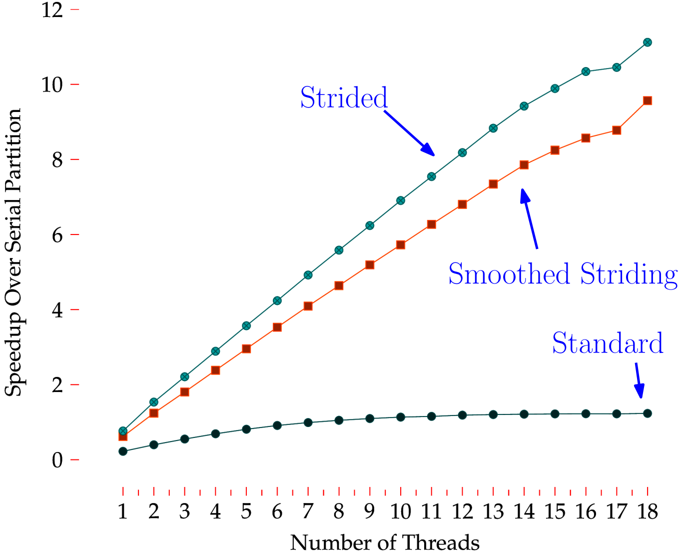
[Francis and Pannan, 92; Frias and Petit, 08]

- Good cache behavior in practice
- Worst case span is $T_\infty \approx n$
- On random inputs span is $T_\infty = \tilde{O}(n^{2/3})$

Smoothed-Striding Algorithm

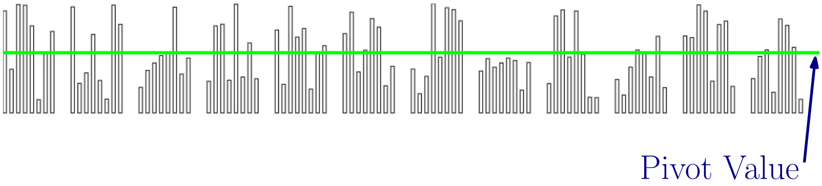
- Provably optimal cache behavior
- Span is $T_\infty = O(\log n \log \log n)$ with high probability in n
- Uses randomization *inside* the algorithm

Smoothed-Striding Algorithm's performance

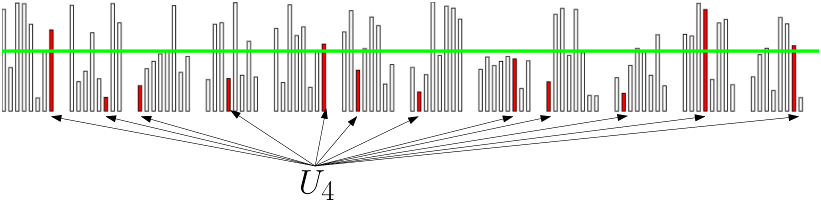


The Smoothed-Striding Algorithm

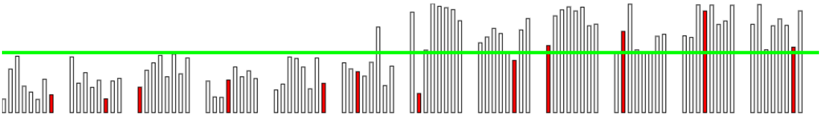
Logically partition the array into chunks of adjacent elements



Form groups U_i that contain a **random** element from each chunk (the Strided Algorithm forms groups P_i that contain the i-th element from each chunk)

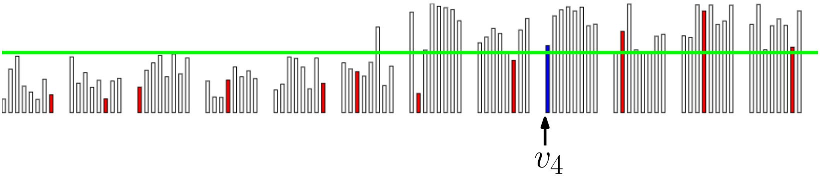


Perform serial partitions on each U_i in parallel over the U_i's

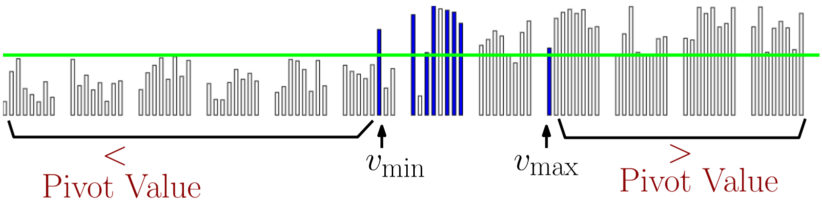


This step is highly parallel.

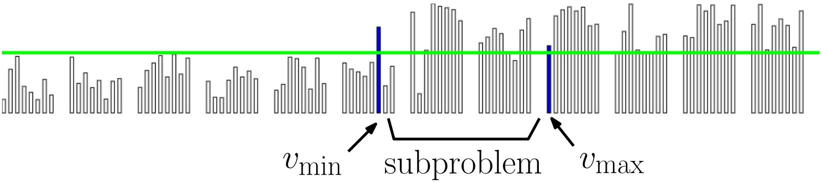
Define v_i = index of first element greater than the pivot in U_i



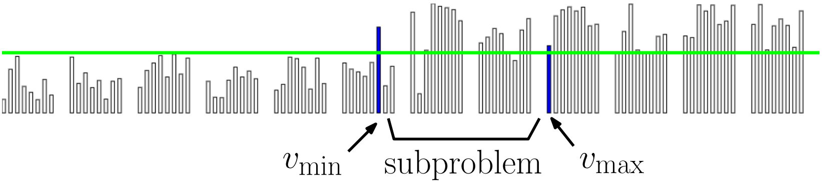
Identify leftmost and rightmost v_i



Final step: Recursively partition the subarray



Final step: Recursively partition the subarray



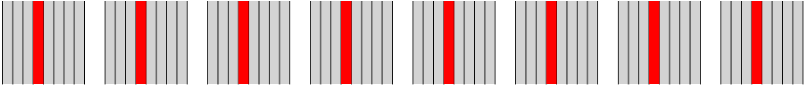
- Recursion is now possible!
  (unlike in the Strided Algorithm, where the subproblem is a worst case input)
- Randomness guarantees that $v_{\mathrm{max}} - v_{\mathrm{min}}$ is small
  (unlike in the Strided Algorithm, where $v_{\mathrm{max}} - v_{\mathrm{min}}$ could be as large as $\Theta(n)$)

A Key Challenge

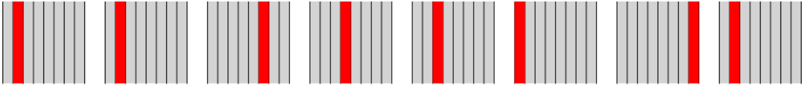
How do we store the U_i's if they are all random?

Storing which elements make up each U_i takes too much space!

Strided Algorithm P_i.



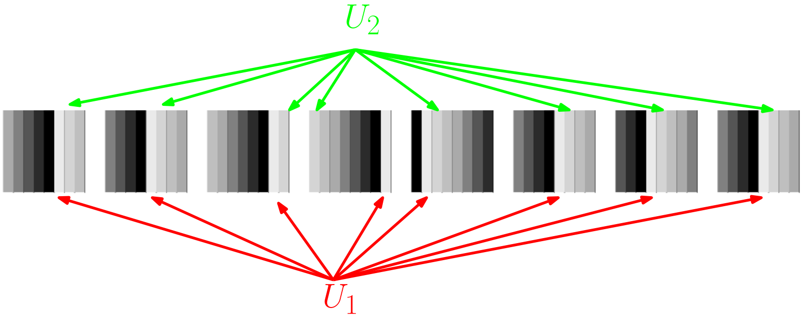
Smoothed-Striding Algorithm U_i.



How to Store the groups

Key Insight: While each U_i does need to contain a random element from each chunk, the U_i's don't need to be *independent*.

We store U_1, and all other groups are determined by a "circular shift" of U_1 (wraparound within each chunk).



An Open Question

Our algorithm: span $T_\infty = O(\log n \log \log n)$

Standard Algorithm: span $T_\infty = O(\log n)$.

Can we get optimal cache behavior and span $O(\log n)$?