Cache-Efficient Parallel Partition Algorithms

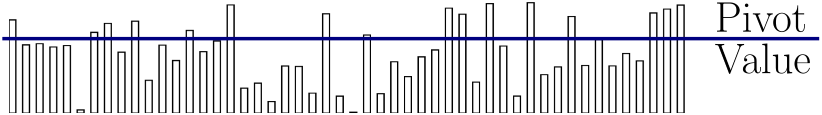
Alek Westover

MIT PRIMES

October 20, 2019

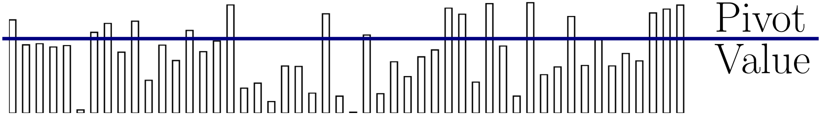
The Partition Problem

An unpartitioned array:

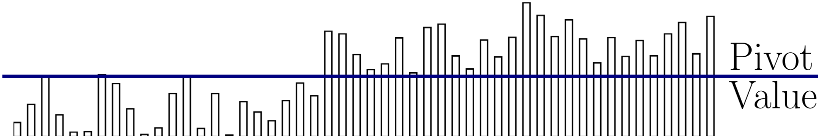


The Partition Problem

An unpartitioned array:

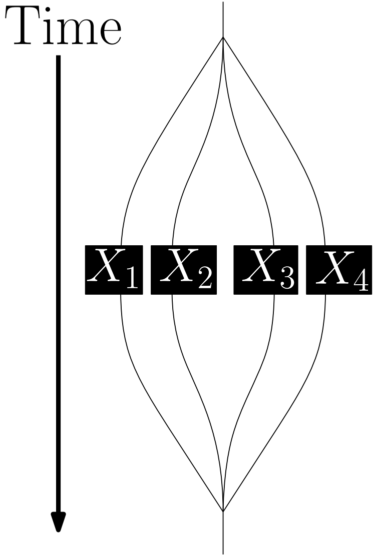


An array partitioned relative to a pivot value:



What is a Parallel Algorithm?

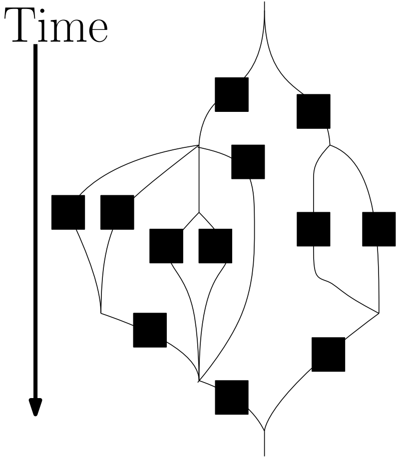
Fundamental primitive:
Parallel for loop



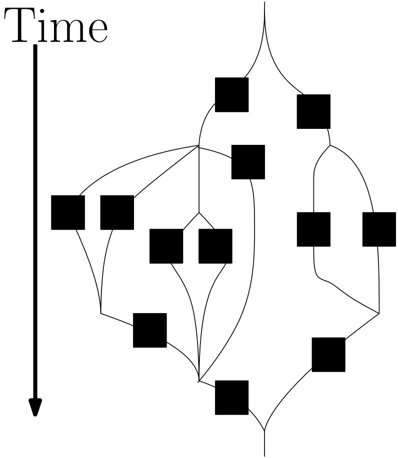
Parallel-For i **from** 1 **to** 4**:**
    Do X_i

What is a Parallel Algorithm?

More complicated parallel structures can be made by combining parallel for loops and recursion.



T_p: Time to run on p processors



Important extreme cases:

Work: T_1

- time to run in serial
- ″sum of all work″

Span: T_∞

- time to run on infinitely many processors
- ″height of the graph″

Bounding T_p with Work and Span

Brent's Theorem: [Brent, 74]

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

Take away: Work T_1 and span T_∞ determine T_p.

THE STANDARD PARALLEL PARTITION ALGORITHM

Step	*Span*
Create filtered array	$O(1)$
Compute prefix sums of filtered array	$O(\log n)$
Use prefix sums to partition array	$O(1)$

Total work: $T_1 = O(n)$
Total span: $T_\infty = O(\log n)$

The Problem

Standard Algorithm is slow in practice

- Uses extra memory
- Makes multiple passes over array

"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

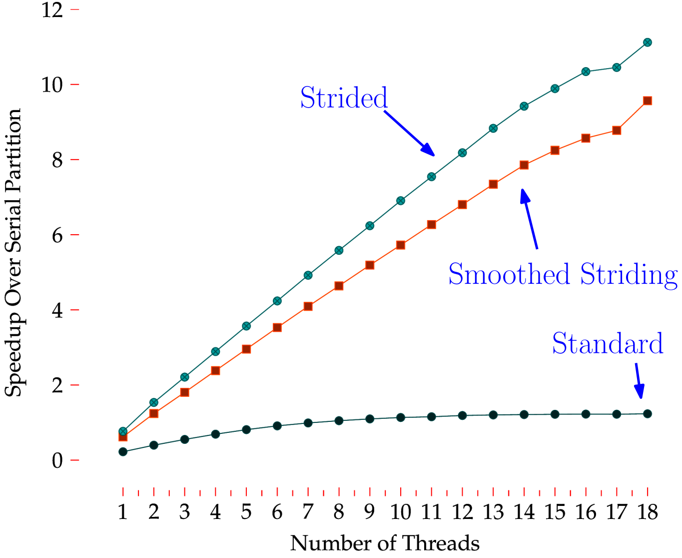
[Francis and Pannan, 92; Frias and Petit, 08]

- 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

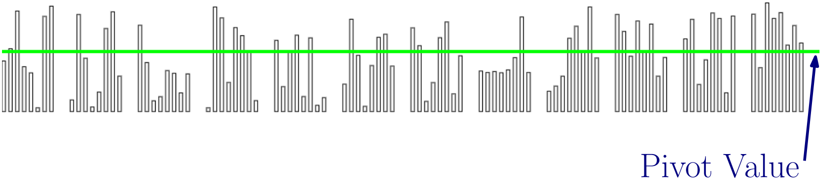
Smoothed-Striding Algorithm's performance



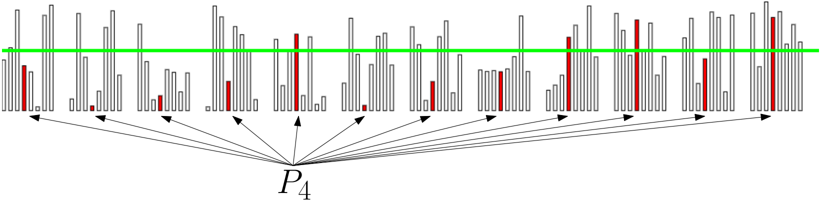
The Strided Algorithm

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

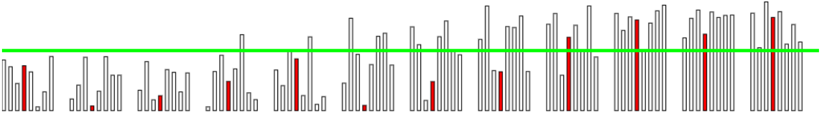
Logically partition the array into chunks of adjacent elements



Form groups P_i that contain the i-th element from each chunk

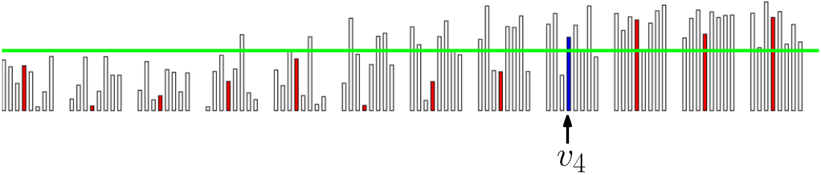


Perform serial partitions on each P_i in parallel over the P_i's

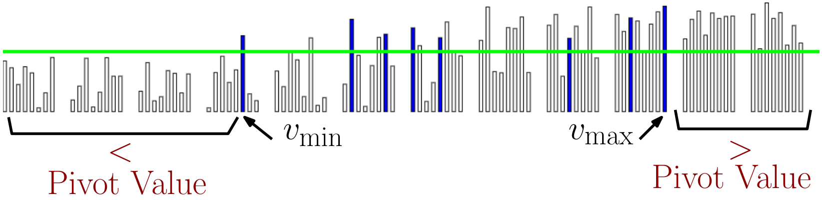


This step is highly parallel.

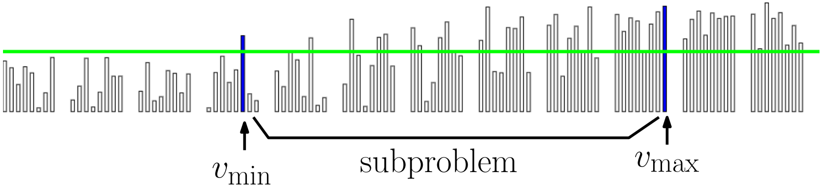
Define v_i = index of first element greater than the pivot in P_i



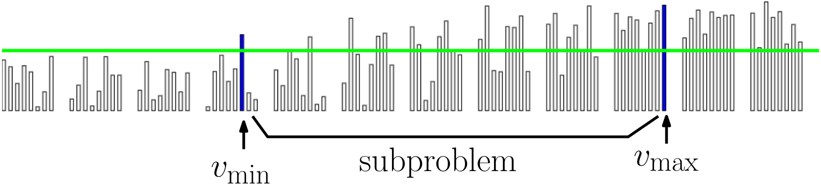
Identify leftmost and rightmost v_i



Final step: Recursively partition the subarray



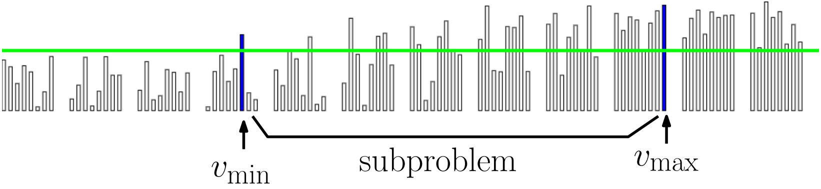
~~**Final step:** Recursively partition the subarray~~



- Recursion is impossible!
- **Final Step:** Partition the subarray *in serial*.

Subproblem Span $T_\infty \approx v_{\max} - v_{\min}$

~~**Final step:** Recursively partition the subarray~~

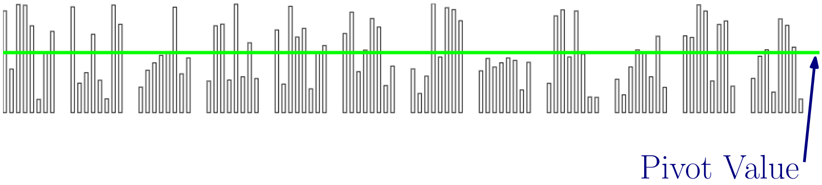


- Recursion is impossible!
- **Final Step:** Partition the subarray *in serial*.

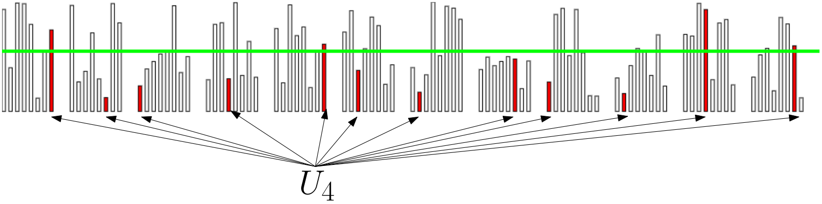
Subproblem Span $T_\infty \approx v_{\max} - v_{\min} \longleftarrow$ n in worst case.

The Smoothed-Striding Algorithm

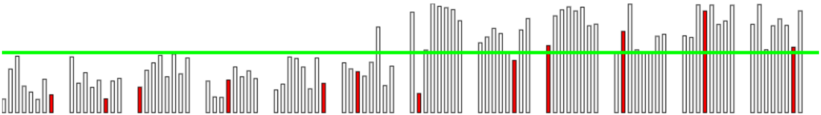
Logically partition the array into chunks of adjacent elements



Key difference: Form groups U_i that contain a random element from each chunk

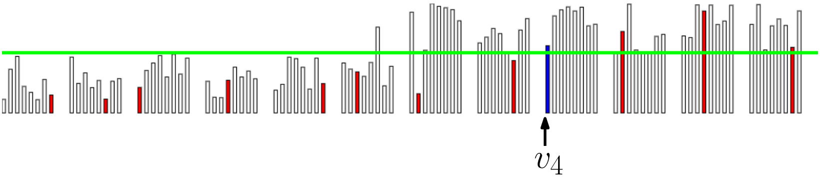


Perform serial partitions on each U_i in parallel over the U_i's

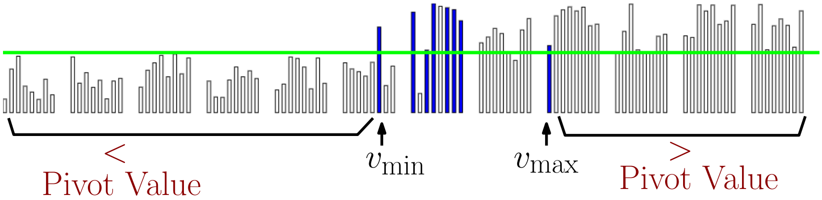


This step is highly parallel.

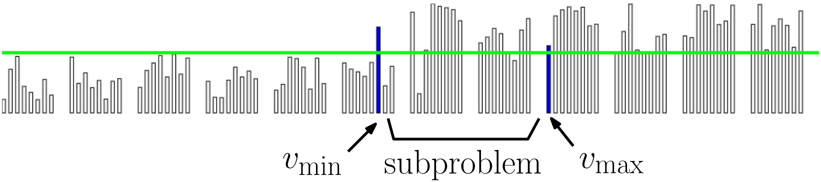
Define v_i = index of first element greater than the pivot in U_i



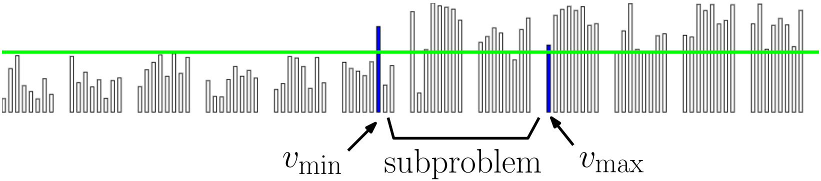
Identify leftmost and rightmost v_i



Final step: Recursively partition the subarray



Final step: Recursively partition the subarray



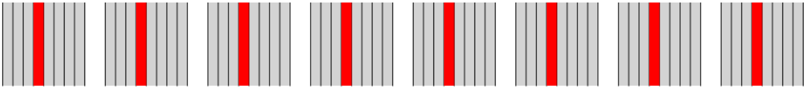
- Recursion is now possible!
- Randomness guarantees that $v_{\max} - v_{\min}$ is small

A Key Challenge

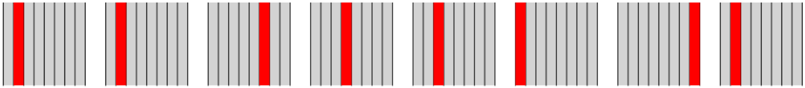
How do we store the U_i's if they are all random?

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

Strided Algorithm P_i.



Smoothed-Striding Algorithm U_i.



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 $O(\log n)$?

Acknowledgments

- MIT PRIMES
- William Kuszmaul, my PRIMES mentor
- My parents

Question Slides

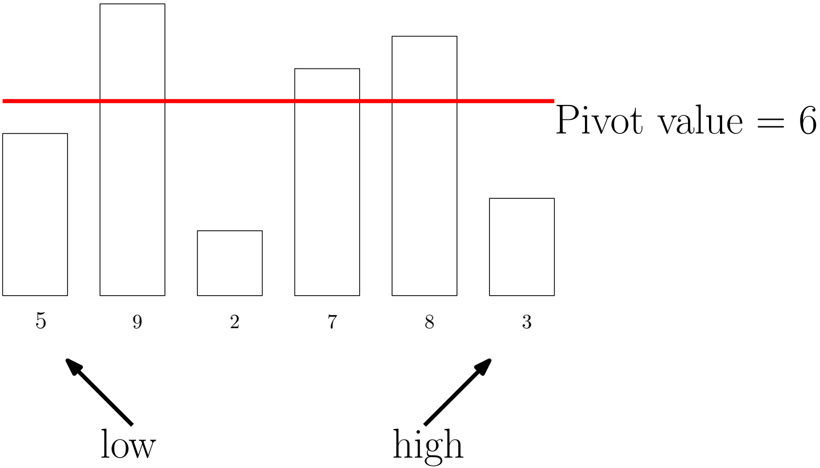
HOW TO STORE THE GROUPS

The solution is to make the groups dependent on one another. Let g be the size of a chunk. Then we only need to store a single group and then the elements of the other groups are determined by this group.

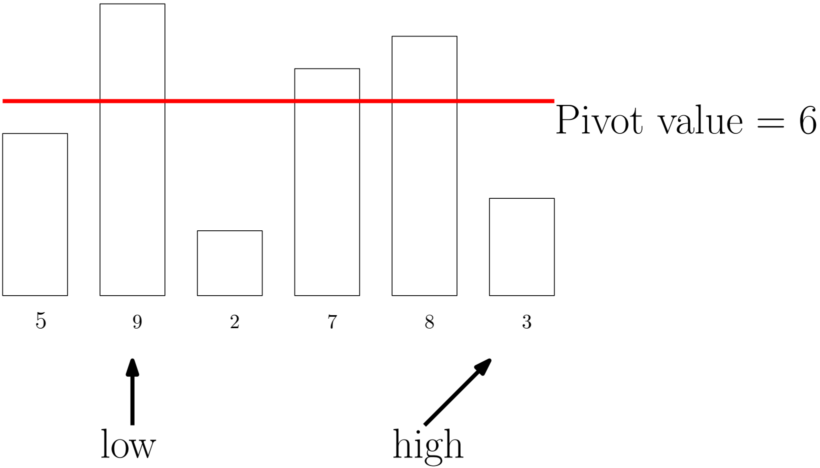
Specifically, let X be an array with values chosen uniformly from $\{1, 2, \ldots, g\}$. Then the i-th element of U_j has index

$$1 + ((X[i] + j) \mod g)$$

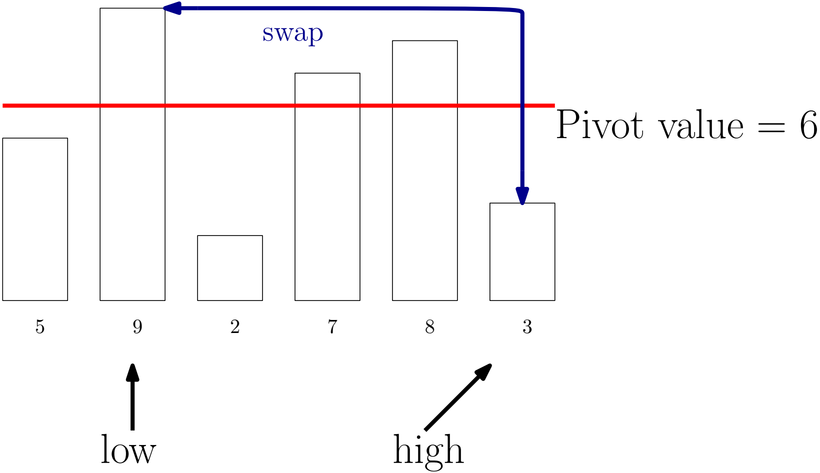
The Serial Partition Algorithm



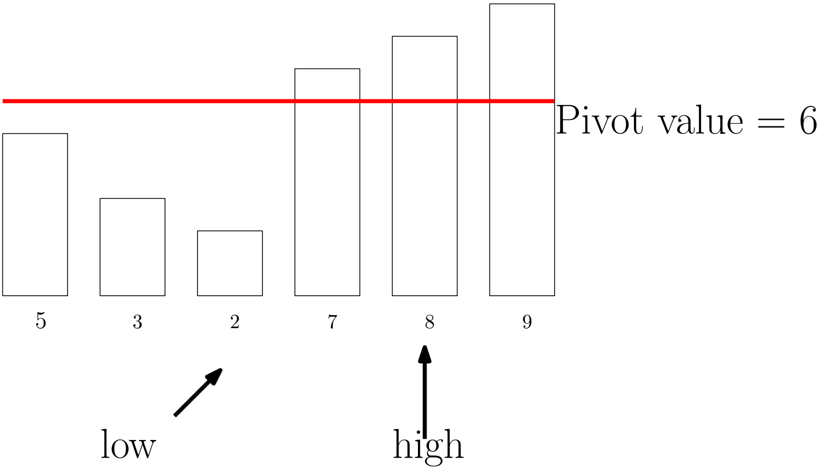
The Serial Partition Algorithm



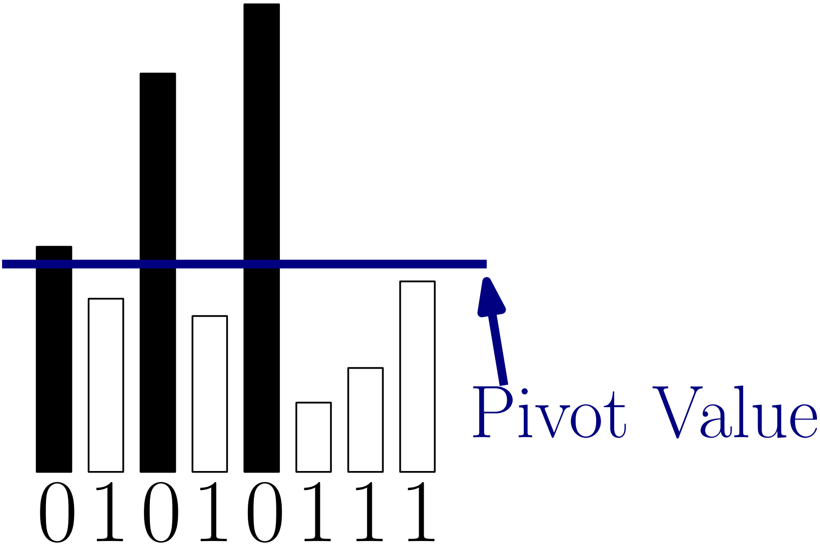
The Serial Partition Algorithm



The Serial Partition Algorithm



The Standard Parallel Partition Algorithm



The Standard Parallel Partition Algorithm