Cache-Efficient Parallel Partition Algorithms

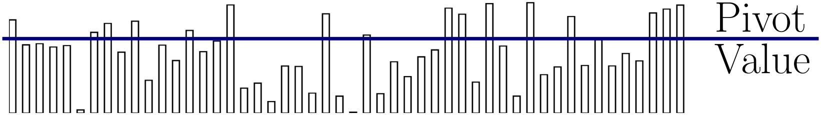
Alek Westover

MIT PRIMES

October 20, 2019

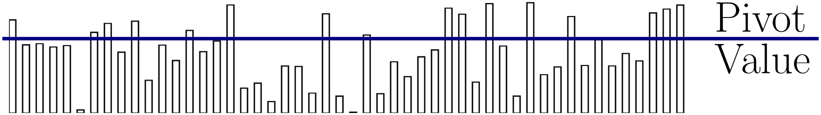
The Partition Problem

An unpartitioned array:

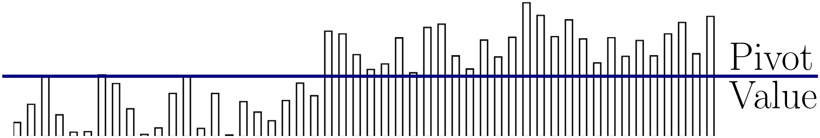


THE PARTITION PROBLEM

An unpartitioned array:



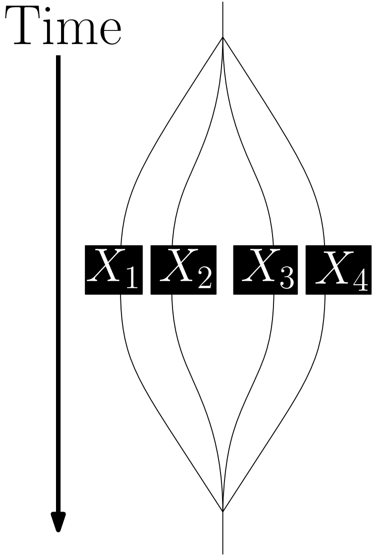
An array partitioned relative to a pivot value:



What is a Parallel Algorithm?

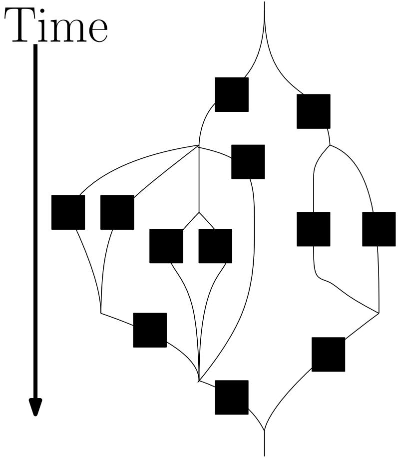
Fundamental primitive:
Parallel for loop

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

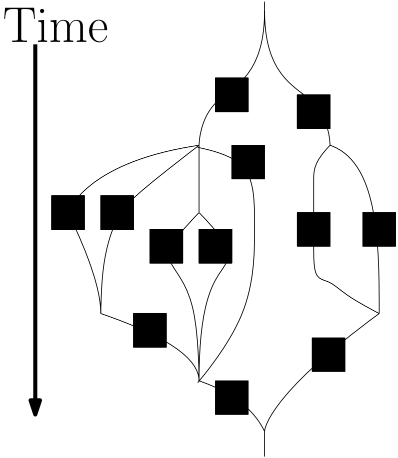


What is a Parallel Algorithm?

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 span: $O(\log n)$

THE PROBLEM

Why is the Standard Algorithm is slow in practice?

The Problem

Why is the Standard Algorithm is slow in practice?

- Uses extra memory

The Problem

Why is the Standard Algorithm is slow in practice?

- Uses extra memory
- Makes multiple passes over array

THE PROBLEM

Why is the Standard Algorithm is slow in practice?

- Uses extra memory
- Makes multiple passes over array

"bad cache behavior"

THE PROBLEM

Why is the Standard Algorithm is slow in practice?

- Uses extra memory
- Makes multiple passes over array

”bad cache behavior”

But fastest algorithms in practice lack theoretical guarantees

THE PROBLEM

Why is the Standard Algorithm is slow in practice?

- Uses extra memory
- Makes multiple passes over array

”bad cache behavior”

But fastest algorithms in practice lack theoretical guarantees

- Lock-based and atomic-variable based algorithms

THE PROBLEM

Why is the Standard Algorithm is slow in practice?

- Uses extra memory
- Makes multiple passes over array

$\Big\}$ "bad cache behavior"

But fastest algorithms in practice lack theoretical guarantees

- Lock-based and atomic-variable based algorithms
- The Strided Algorithm

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

The Problem

Why is the Standard Algorithm is slow in practice?

- Uses extra memory
- Makes multiple passes over array

"bad cache behavior"

But fastest algorithms in practice lack theoretical guarantees

- Lock-based and atomic-variable based algorithms
- The Strided Algorithm

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

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

THE PROBLEM

Why is the Standard Algorithm is slow in practice?

- Uses extra memory
- Makes multiple passes over array

$\Big\}$ ”bad cache behavior”

But fastest algorithms in practice lack theoretical guarantees

- Lock-based and atomic-variable based algorithms
- The Strided Algorithm

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

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

Our Question: Can we create an algorithm with theoretical guarantees that is fast in practice?

Our Result

We created a randomized algorithm for the parallel partition problem: the ***Smoothed-Striding Algorithm***.

The Smoothed-Striding algorithm has...

- performance comparable to that of the Strided Algorithm
- polylogarithmic span like the Standard Algorithm
- theoretically optimal cache behavior

Algorithm Comparison

Strided Algorithm

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

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

Smoothed-Striding Algorithm

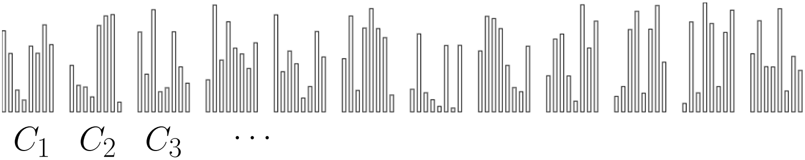
- Provably optimal cache behavior
- Span is $O(\log n \log \log n)$ with high probability in n
- Uses randomization inside the algorithm to remove the need for randomized input

Speedup over Serial Partition: Smoothed-Striding Algorithm's performance

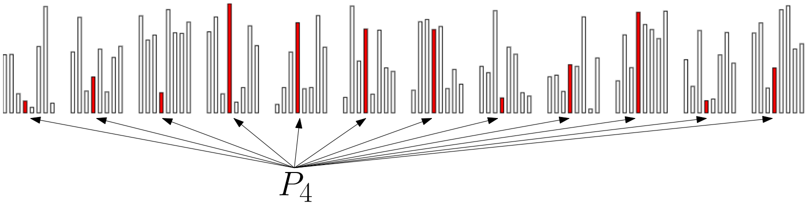
The Strided Algorithm

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

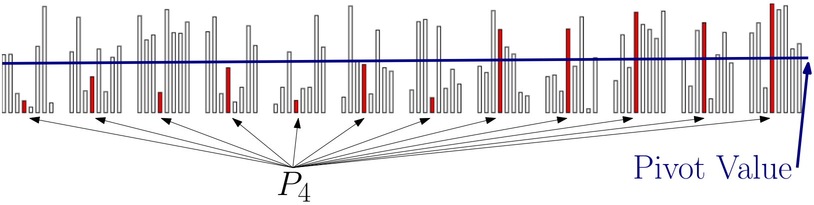
Logically partition the array into chunks of adjacent elements:



Form groups P_i where P_i contains the i-th element from each chunk:

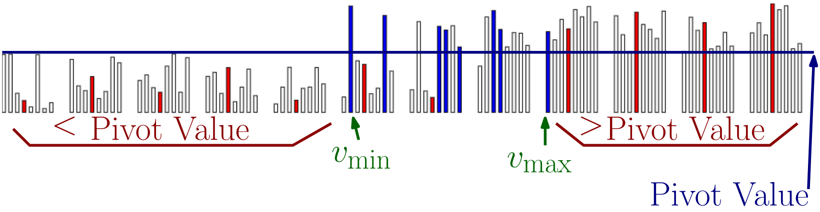


Perform serial partitions on each P_i in parallel over the P_i's:

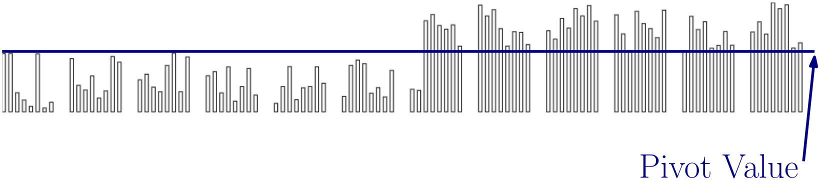


This step is highly parallel.

Identify the splitting index v_i (the first element greater than the pivot) of each P_i.

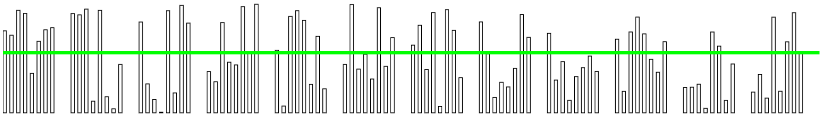


Partition the subarray from the minimum splitting index to the maximum splitting index in serial. This completes the partition.

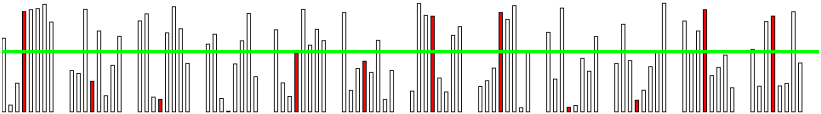


Note that this step has no parallelism. In general this results in span $O(n)$. However, if the number of elements less than the pivot in each P_i is similar, then size of the subarray to be partitioned can be very small.

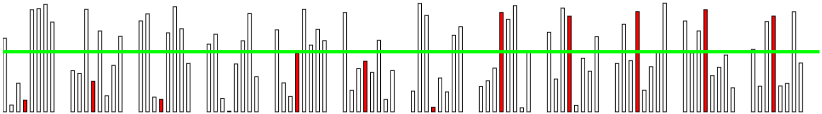
Logically partition the array into chunks of adjacent elements:



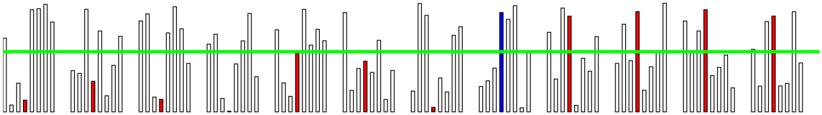
Form groups P_i where P_i contains the i-th element from each chunk:



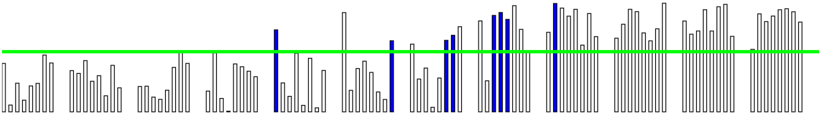
Perform serial partitions on each P_i in parallel over the P_i's:

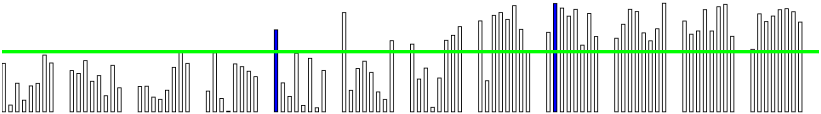


This step is highly parallel.

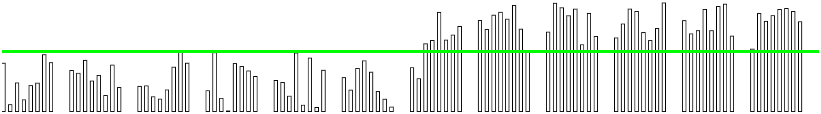


Identify the splitting index v_i (the first element greater than the pivot) of each P_i.





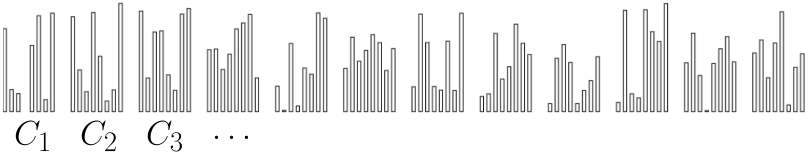
Partition the subarray from the minimum splitting index to the maximum splitting index in serial. This completes the partition.



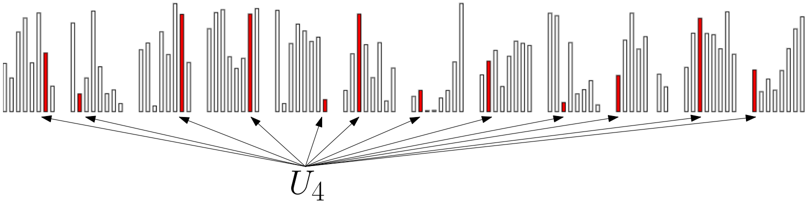
Note that this step has no parallelism. In general this results in span $O(n)$. However, if the number of elements less than the pivot in each P_i is similar, then size of the subarray to be partitioned can be very small.

The Smoothed-Striding Algorithm

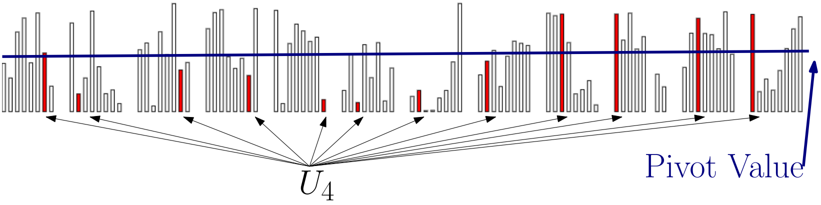
Logically partition the array into chunks of adjacent elements:



Form groups U_i where U_i contains the i-th element from each chunk:

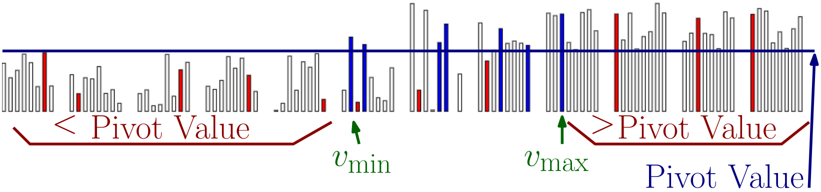


Perform serial partitions on each U_i in parallel over the U_i's:

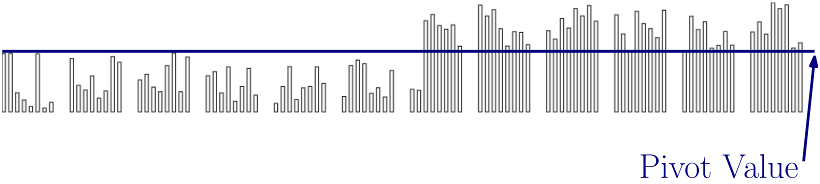


This step is highly parallel.

Identify the splitting index v_i (the first element greater than the pivot) of each P_i.

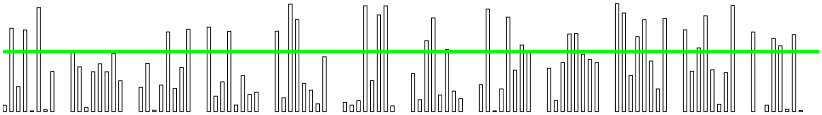


Recursively apply this algorithm to partition the subarray from the minimum splitting index to the maximum splitting index in serial. This completes the partition.

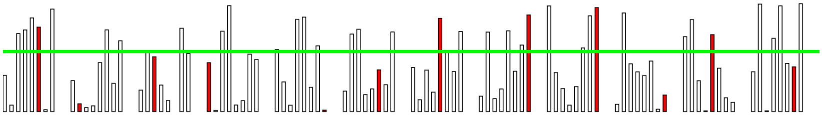


Unlike in the Strided Algorithm this step has parallelism, and is guaranteed to only run on a small subarray. The Strided Algorithm could not recurse here because the subproblem is a worst case input for it.

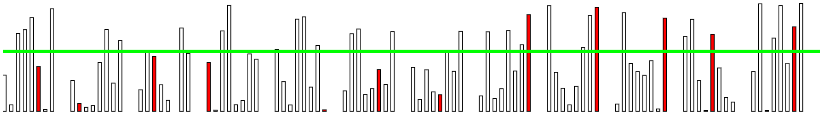
Logically partition the array into chunks of adjacent elements:



Form groups U_i where U_i contains the i-th element from each chunk:

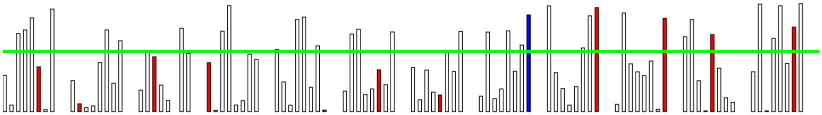


Perform serial partitions on each U_i in parallel over the U_i's:

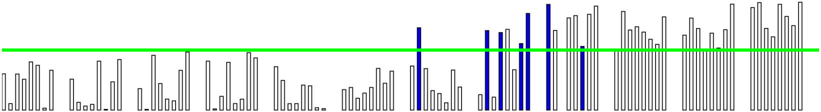


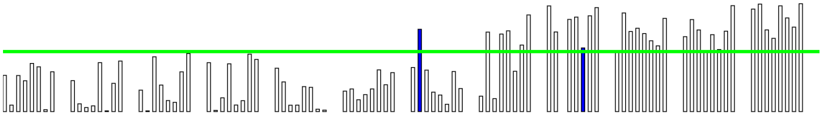
This step is highly parallel.

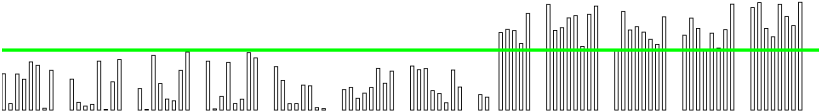
Identify the splitting index v_i (the first element greater than the pivot) of each P_i.



Recursively apply this algorithm to partition the subarray from the minimum splitting index to the maximum splitting index in serial. This completes the partition.







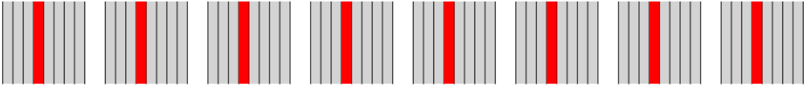
Unlike in the Strided Algorithm this step has parallelism, and is guaranteed to only run on a small subarray. The Strided Algorithm could not recurse here because the subproblem is a worst case input for it.

A Key Challenge

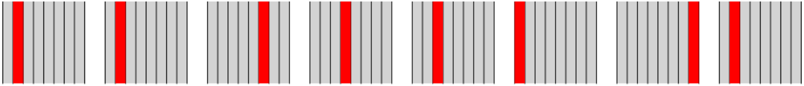
How do we store the U_i's if they are all random?

If we stored the parts that made up each U_i we would not be in-place.

Blocked Strided Algorithm P_i.



Smoothed-Striding Algorithm U_i.



Open Questions

By recursively applying the Smoothed-Striding algorithm we get an algorithm for parallel partition that incurs $n(1 + o(1))$ cache misses and has span $O(b \log^2 n)$.

There are techniques for improving this span to $O(\log n \log \log n)$ while retaining the cache behavior.

But the standard algorithm has span $O(\log n)$.

Can we construct an algorithm that achieves optimal cache behavior and span $O(\log n)$?

Acknowledgments

I would like to thank

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