Cache-Efficient Parallel Partition Algorithms

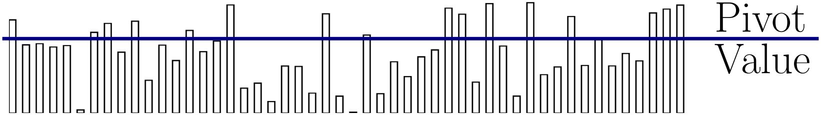
Alek Westover

MIT PRIMES

October 20, 2019

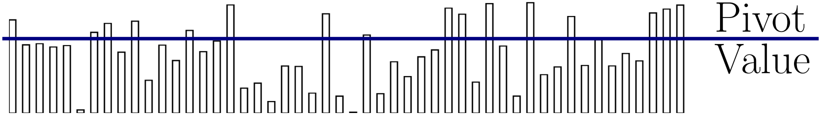
The Partition Problem

An unpartitioned array:

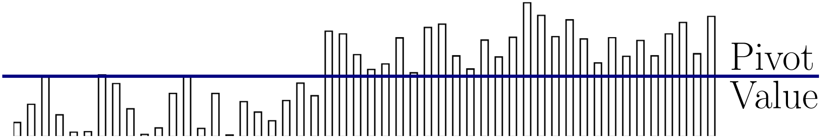


THE PARTITION PROBLEM

An unpartitioned array:



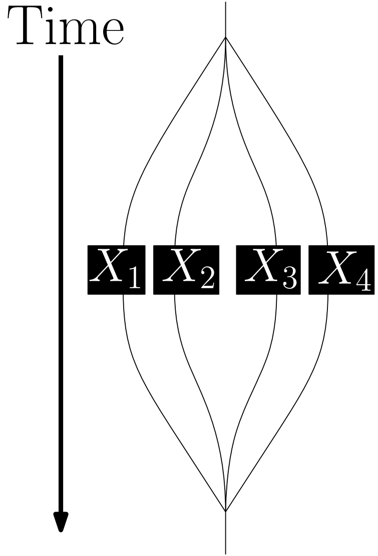
An array partitioned relative to a pivot value:



WHAT IS A PARALLEL ALGORITHM?

Fundamental primitive:
Parallel for loop

parallel-for $i \in \{1, 2, 3, 4\}$
  do X_i
endparallel-for



What is a Parallel Algorithm?

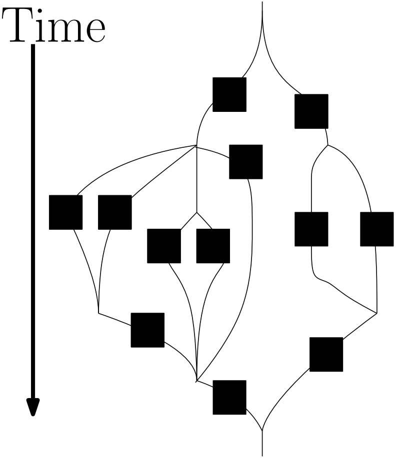
More complicated parallel structures can be made by combining parallel for loops and recursion.

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]

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

Brent's Theorem implies that analyzing an algorithm's work and span is sufficient to determine the algorithms performance on p processors.

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

The Standard Algorithm is slow in practice. Why?

The Problem

The Standard Algorithm is slow in practice. Why?

- Algorithm is not in-place

The Problem

The Standard Algorithm is slow in practice. Why?

- Algorithm is not in-place
- Algorithm has poor cache behavior

The Problem

The Standard Algorithm is slow in practice. Why?

- Algorithm is not in-place
- Algorithm has poor cache behavior

But the fastest algorithms in practice lack theoretical guarantees

The Problem

The Standard Algorithm is slow in practice. Why?

- Algorithm is not in-place
- Algorithm has poor cache behavior

But the fastest algorithms in practice lack theoretical guarantees

- Lock-based and atomic-variable based algorithms

The Problem

The Standard Algorithm is slow in practice. Why?

- Algorithm is not in-place
- Algorithm has poor cache behavior

But the 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

The Standard Algorithm is slow in practice. Why?

- Algorithm is not in-place
- Algorithm has poor cache behavior

But the 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?

Cache Efficiency

Rough definition:
The number of passes over the input data.

A cache-efficient algorithm will make relatively few requests to load data from memory into cache where it can be manipulated.

Poor cache behavior will harm an algorithms performance in practice.

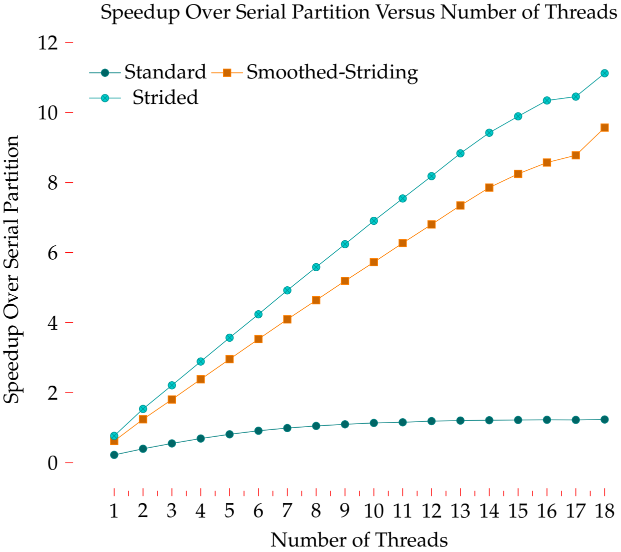
Our Result

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

The Smoothed-Striding algorithm:

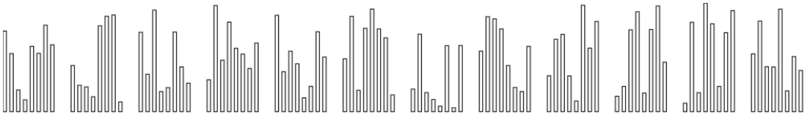
- Has polylogarithmic span like the Standard Algorithm
- Has theoretically optimal cache behavior (up to low order factors)
- Has performance comparable to that of the Strided Algorithm

The Smoothed-Striding Algorithm's performance



Strided Algorithm Description [Francis and Pannan, 92; Frias and Petit, 08]

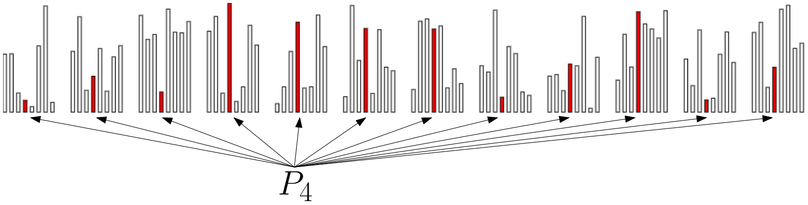
Logically partition the array into chunks of adjacent elements:



$C_1 \quad C_2 \quad C_3 \quad \cdots$

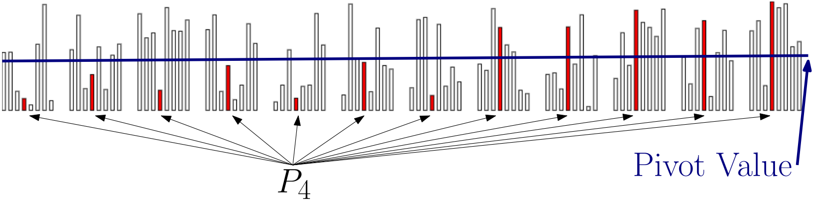
Strided Algorithm Description [Francis and Pannan, 92; Frias and Petit, 08]

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



Strided Algorithm Description [Francis and Pannan, 92; Frias and Petit, 08]

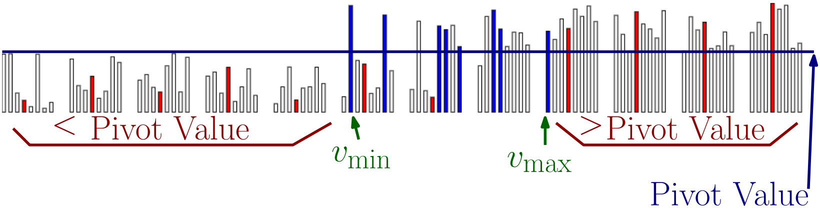
Perform serial partitions on each P_i in parallel over the P_i's:



This step is highly parallel.

Strided Algorithm Description [Francis and Pannan, 92; Frias and Petit, 08]

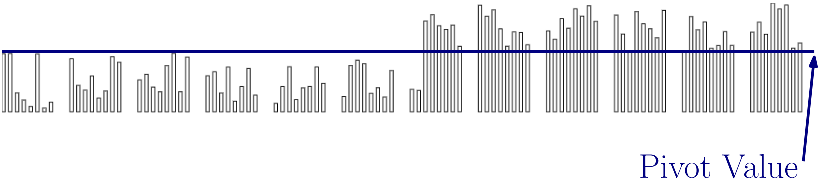
Identify the splitting index v_i (the first element greater than the pivot) of each P_i.



Note that all elements below the minimum splitting index are less than the pivot and all elements greater than the maximum splitting index are greater than the pivot.

Strided Algorithm Description [Francis and Pannan, 92; Frias and Petit, 08]

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

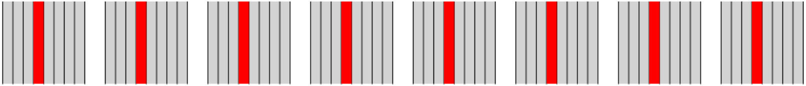


Note that this step has no parallelism. In general this results in span $O(n)$. However, if the number of elements less than the pivot in each P_i is similar, then size of the subarray to be partitioned can be very small.

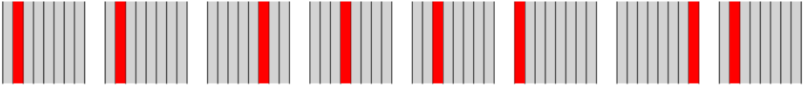
The Smoothed Striding Algorithm

The Smoothed-Striding Algorithm creates groups analogous to the Strided Algorithm's P_i's, but rather than taking the i-th element from each chunk of the array to form groups P_i, the Smoothed-Striding algorithm takes a random element from each chunk for each of the groups U_i.

Blocked Strided Algorithm P_i.

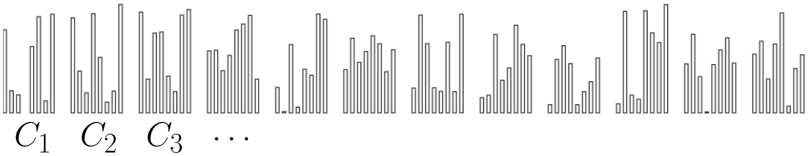


Smoothed-Striding Algorithm U_i.



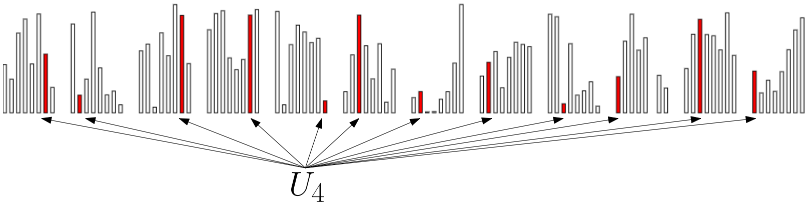
Smoothed Striding Algorithm Description

Logically partition the array into chunks of adjacent elements:



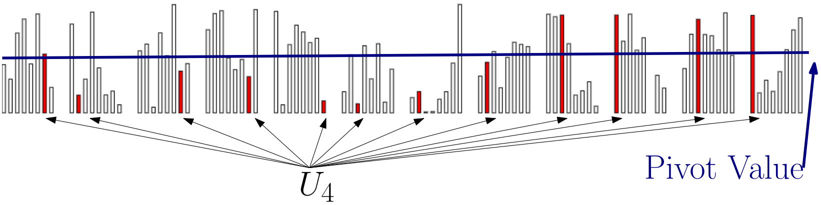
Smoothed Striding Algorithm Description

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



Smoothed Striding Algorithm Description

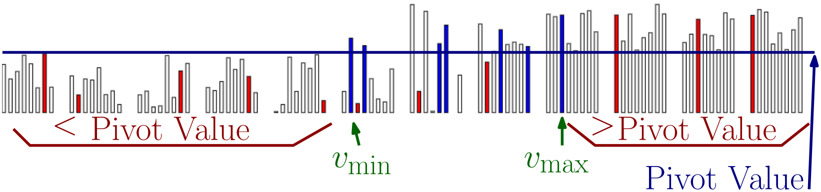
Perform serial partitions on each U_i in parallel over the U_i's:



This step is highly parallel.

Smoothed Striding Algorithm Description

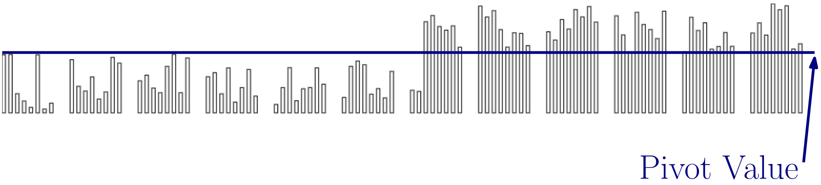
Identify the splitting index v_i (the first element greater than the pivot) of each P_i.



Note that all elements below the minimum splitting index are less than the pivot and all elements greater than the maximum splitting index are greater than the pivot.

Smoothed Striding Algorithm Description

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.

Technical Notes

Storing the groups U_i is a challenge. We can't explicitely store them because then the algorithm would not be in-place. By design they do not have a regular structure like the P_i of the Strided Algorithm.
The solution is to store U_1 and then specify that all other U_i's are a circular shift within the chunks of U_1.
More precisely, Let $X[1], \ldots, X[s]$ be chosen uniformly at random from $\{1, \ldots, g\}$. Then let U_i be the union of the $(X[j] + i) \mod g$-th cache-line from each chunk C_j.
Note that the U_i's are not indpendent, but this doesn't affect the union bound.

In order to compute the minimum and maxmium splitting indices $v_{\min}, v_{\max}$ in parallel we use a recursive structure rather than a parallel-for loop.

Open Questions

By recursively applying the Smoothed-Striding algorithm 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