Cache Efficient Parallel Partition Algorithms
An In-Place Exclusive Read/Write Memory Algorithm

Why is the Partition Problem Important?

The Partition Problem is a fundamental problem in computer science. Additionally, it is used in many algorithms such as

- Parallel Quicksort. This is the most well-known application of parallel-partition. Sorting is a very fundamental problem.
- Filtering operations.

Humans like sorted data.
Computers like sorted data.

Our Research Question

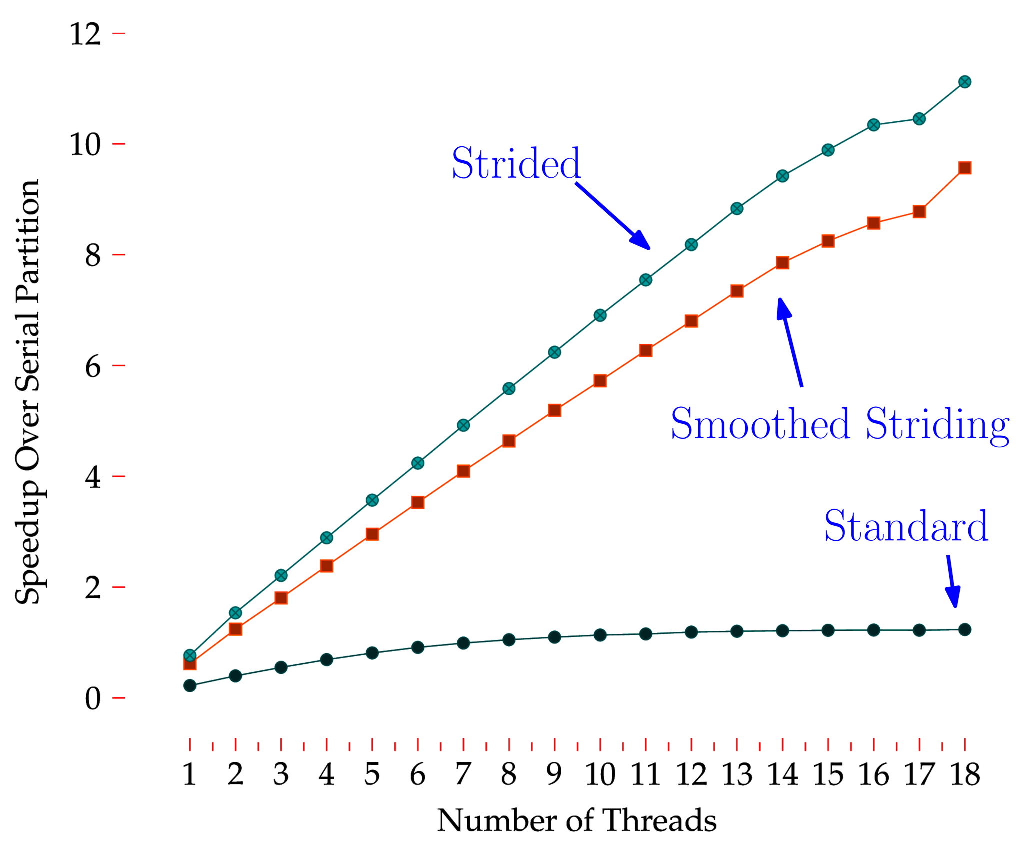
Can we create an algorithm with *theoretical guarantees* that is *fast in practice*?

Result

We created 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)

Smoothed Striding Algorithm's Performance



Strided Versus Smoothed-Striding Algorithm

Strided Algorithm [Francis and Pannan, 92; Frias and Petit, 08]	**Smoothed-Striding Algorithm**
Good cache behavior in practice	Provably optimal cache behavior
	Span is $T_\infty = O(\log n \log\log n)$ with high probability in n
Worst case span is $T_\infty \approx n$	
On random inputs span is $T_\infty = \tilde{O}(n^{2/3})$	Uses randomization *inside* the algorithm

Application To Parallel Quicksort

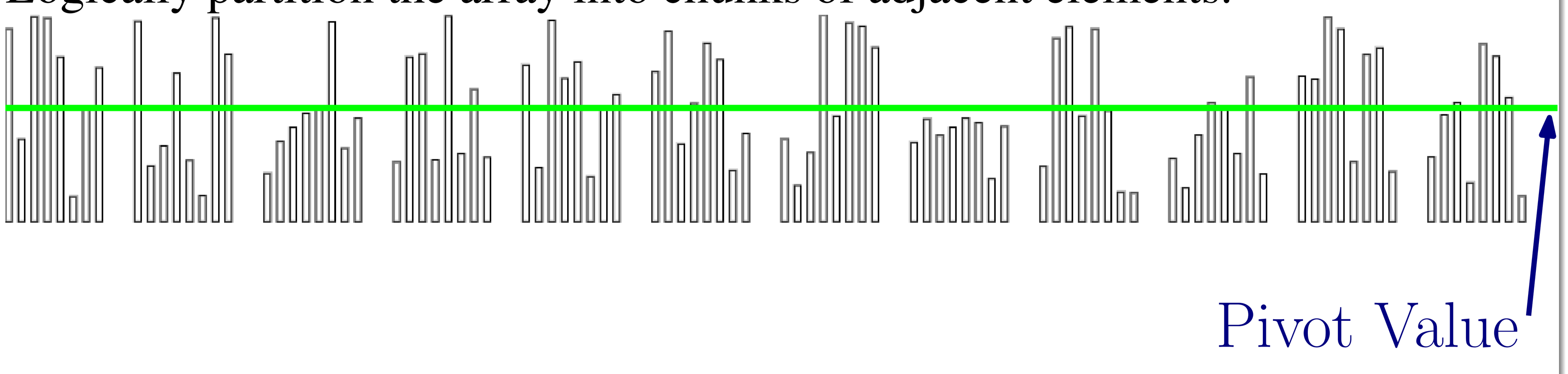
Parallel Quicksort is the most important application of Parallel Partition. Parallel Quicksort works as follows:

- Chose a pivot value randomly from the array
- *Partition* the array relative to the pivot value
- Recursively sort the subarrays (in parallel)

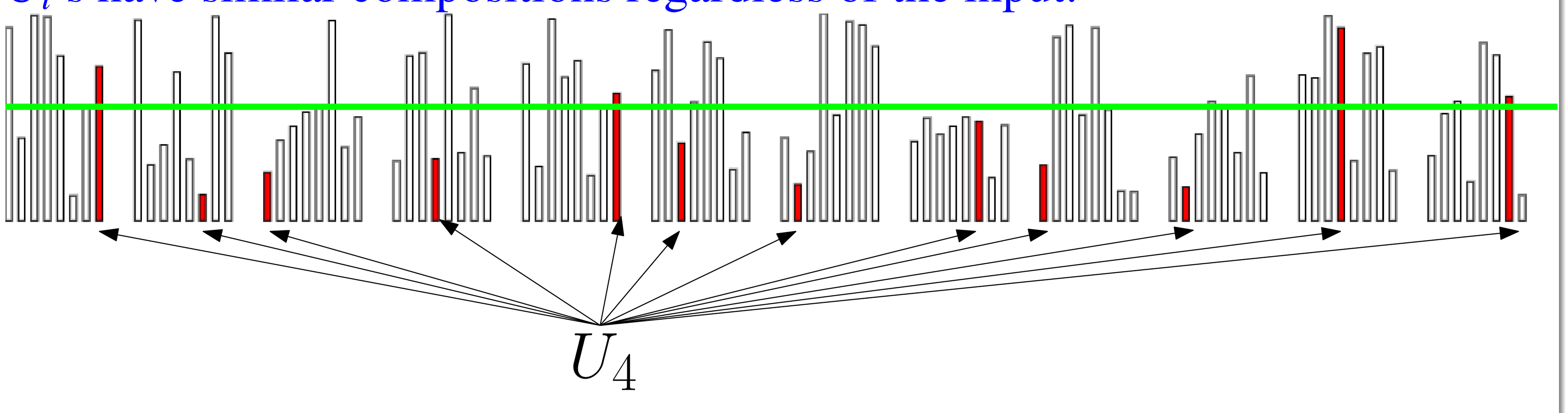
The depth of recursion is $O(\log n)$ with high probability in n, and each level of recursion requires span $O(\log n \log\log n)$ when using the smoothed striding algorithm, which results in span $O(\log^2 n \log\log n)$ and work $O(n \log n)$ for the entire parallel quicksort algorithm, which is within a factor of $\log\log n$ of optimal, while additionally guaranteeing that the algorithm is cache-friendly.

Smoothed Striding Algorithm

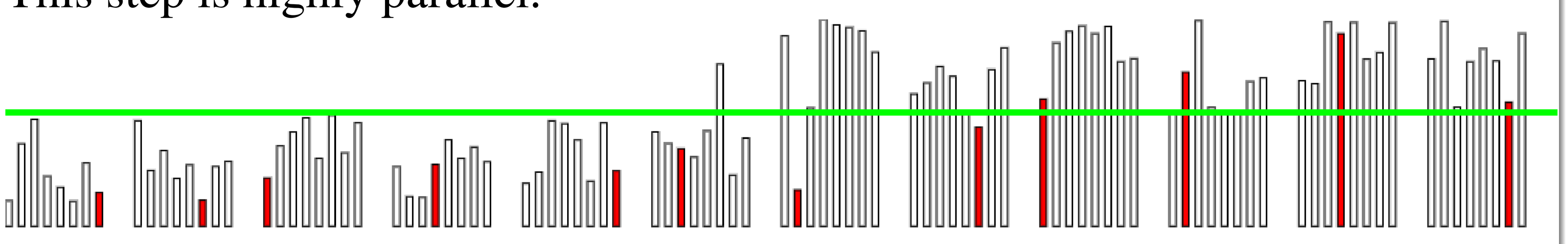
Logically partition the array into chunks of adjacent elements.



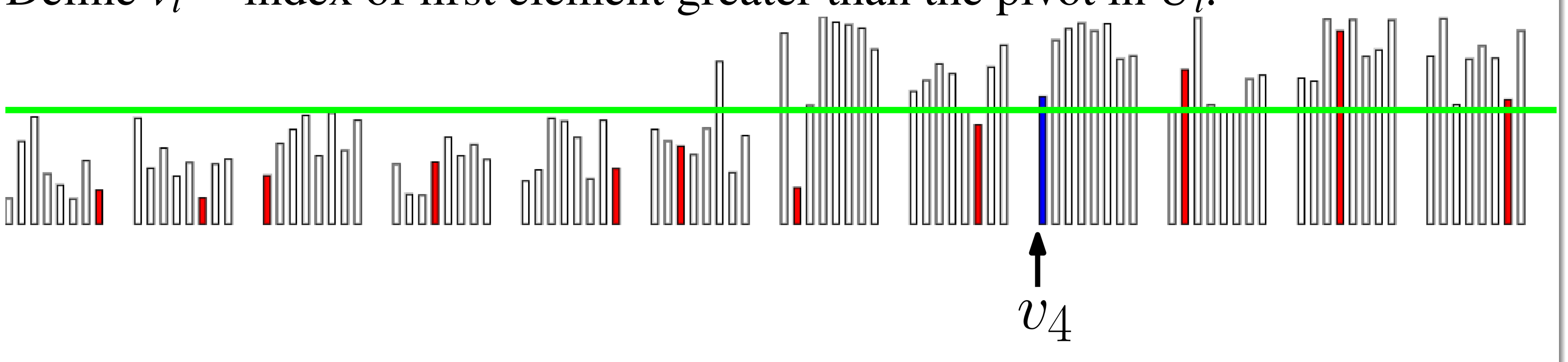
Form groups U_i that contain a random element from each chunk.
This randomization step was one of our key insights; it guarantees that the U_i's have similar compositions regardless of the input.



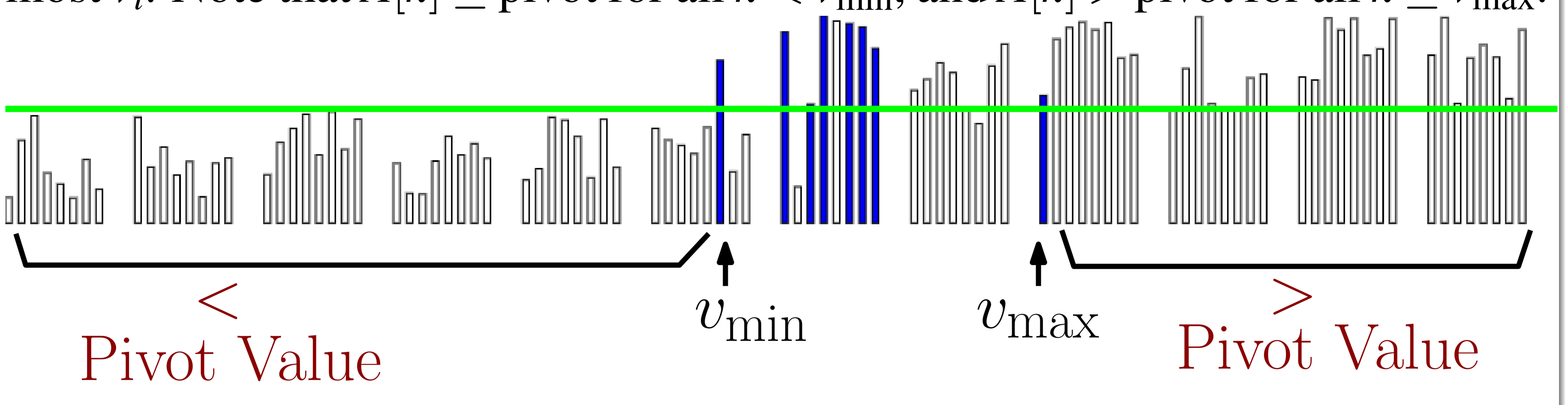
Perform serial partitions on each U_i in parallel over the U_i's.
This step is highly parallel.



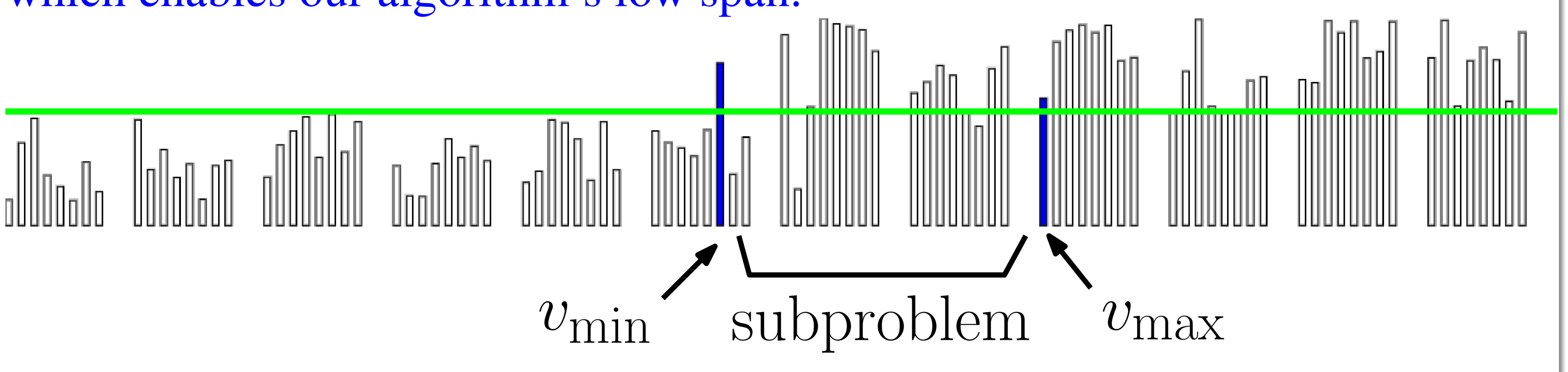
Define $v_i =$ index of first element greater than the pivot in U_i.



Identify leftmost and rightmost v_i. Note that $A[k] \le$ pivot for all $k < v_{\min}$, and $A[k] >$ pivot for all $k \ge v_{\max}$.



Recursively partition the subarray.
This step was previously impossible; adding randomization enables this step, which enables our algorithm's low span.

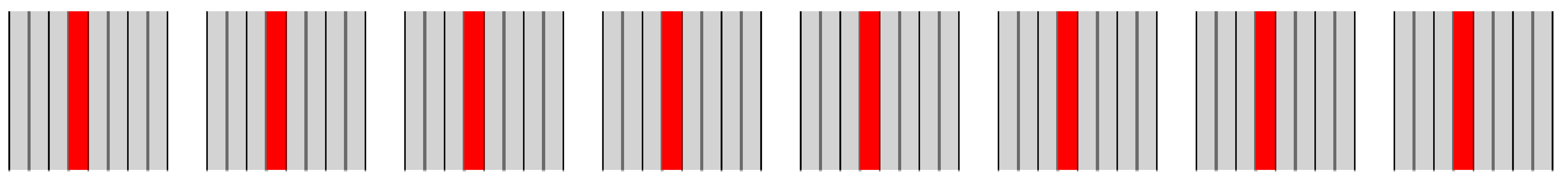


A Key Challenge

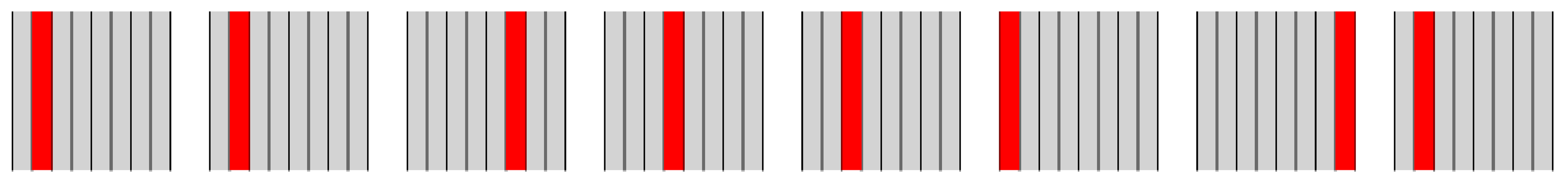
How do we store the U_i's if they are all random?

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).