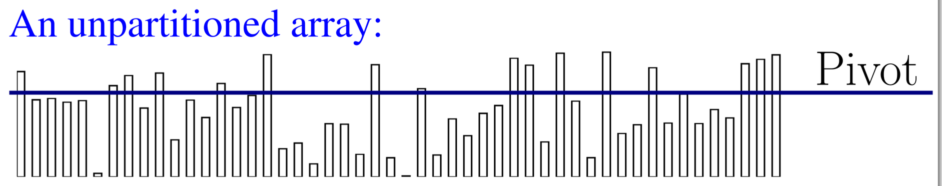
Cache Efficient Parallel Partition Algorithms
An In-Place Exclusive Read/Write Memory Algorithm

What is the Partition Problem?

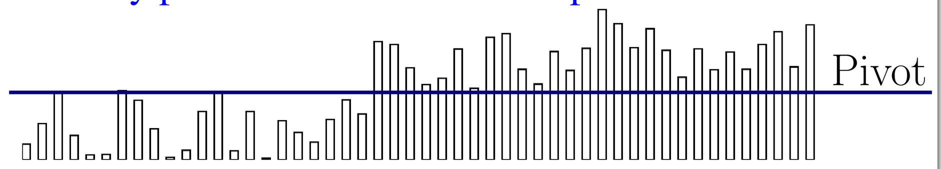
Explanation: The *Partition Problem* is to reorder the elements in a list so that elements in the same group occur in the same part of the list.

Example: A common way of grouping elements is based on whether they exceed or fall short of a certain "pivot" value.

An unpartitioned array:



An array partitioned relative to the pivot value:



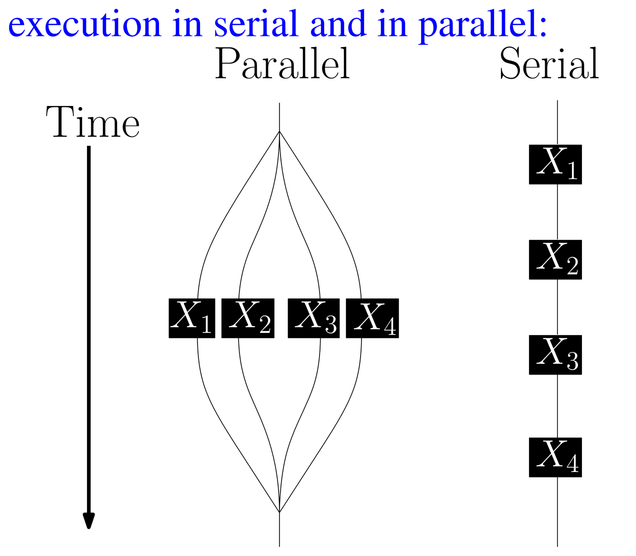
What is a Parallel Algorithm?

Explanation: Whereas a typical (i.e. serial) algorithm runs on a single processor, a *parallel algorithm* runs on $p \geq 1$ processors.

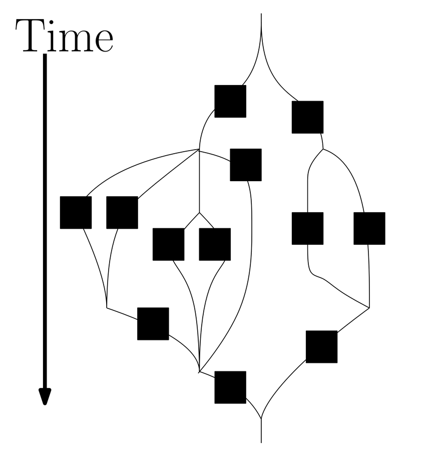
In our model of parallelism, we only allow the concurrency mechanism of parallel-for-loops; in particular our algorithm doesn't make concurrent writes to data (it is **EREW**): we don't allow locks or atomic variables. Being EREW is desirable because theoretical predictions apply more readily to them, and because EREW algorithms are hardware independent.

Example: Many tasks have parts that can be performed concurrently; such tasks can be performed faster with parallel computing.

Program execution in serial and in parallel:



Performance Metrics for Parallel Algorithms



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"

What is Cache Efficiency?

Explanation: *Cache* is a small part of memory that can be accessed much faster than ordinary RAM. When data is already loaded into Cache a program can rapidly access it; this is called a *cache hit*. When data needed by a program isn't in cache it must be loaded into cache; this is called a *cache miss*, and takes time.

Remark: An algorithm with very few cache misses is *Cache Efficient*; cache efficiency leads to faster performance in practice.

Factors in Cache-Efficiency:

- Perform low number of passes over the data
- Don't use extra memory, i.e. are *In-Place*
- Deal with elements that are close in memory together

Previous Work on the Partition Problem

The "Standard Algorithm" is theoretically optimal with span $O(\log n)$, but slow in practice due to poor cache behavior.

The 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]
  Not Exclusive Read/Write Memory
- The Strided Algorithm
  [Francis and Pannan, 92; Frias and Petit, 08]
  No locks or atomic-variables, but no bound on span

Why is the Partition Problem Important?

The Partition Problem is a fundamental problem in computer science. Additionally, it is a subproblem that must be solved in many algorithms such as:

- *Parallel Quicksort* (the most famous and important application partition algorithms)
- *Filtering operations*

Furthermore, the partition problem is of great practical importance as:

- Humans want organized data often, e.g. performing "ORDER BY" on information from a database, or simply ordering data in a spreadsheet
- Many algorithms run faster, or rely on, having sorted data

Our Research Question

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

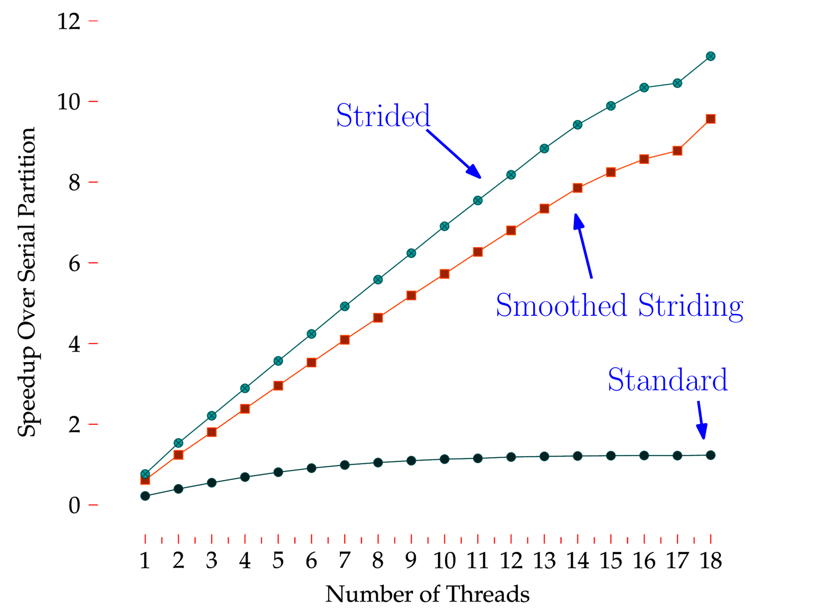
Result

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

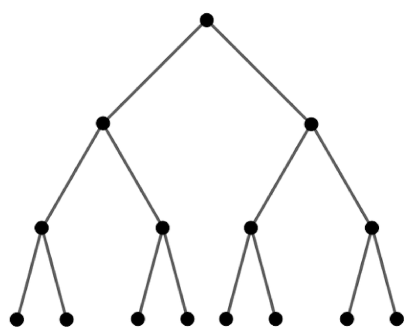
Strided Algorithm [Francis and Pannan, 92; Frias and Petit, 08]	Smoothed-Striding Algorithm
Good cache behavior in practice	Provably optimal cache behavior
Worst case span is $T_\infty \approx n$	Span is $T_\infty = O(\log n \log \log n)$ with high probability in n
On random inputs span is $T_\infty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

Application To Parallel Quicksort

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

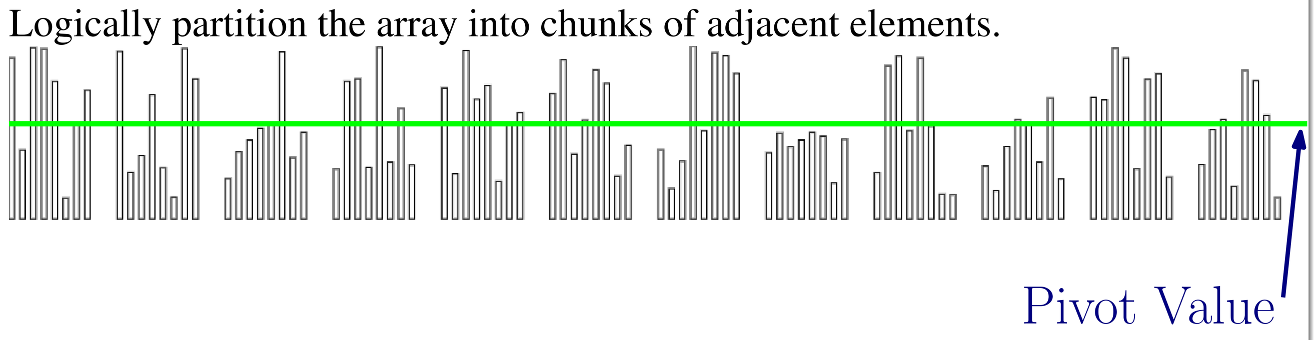
- Chose a pivot value randomly from the array
- *Partition* the array relative to the pivot value
- Recursively sort the subarrays (in parallel)

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. This results in span $O(\log^2 n \log \log n)$ and work $O(n \log n)$ for the entire Parallel Quicksort – which is within a factor of $\log \log n$ of optimal span – while additionally guaranteeing cache-friendliness.

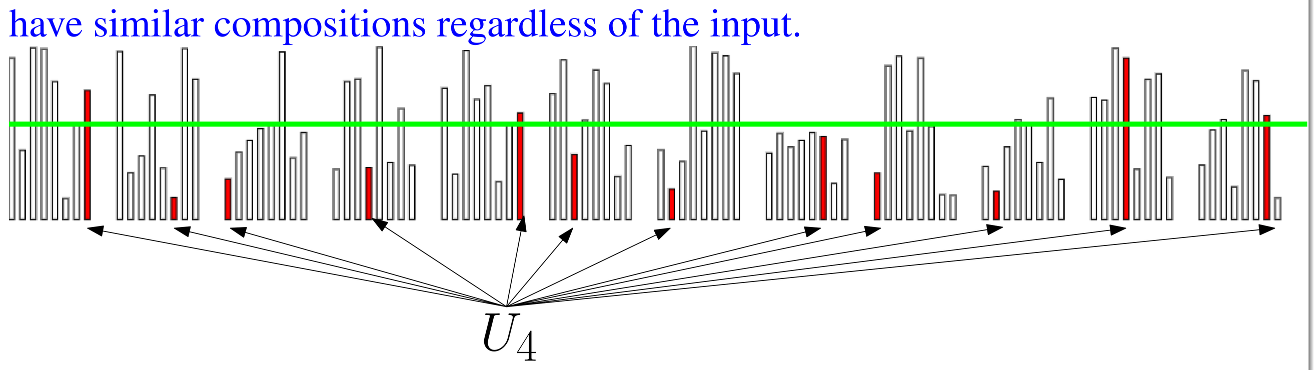


Smoothed Striding Algorithm

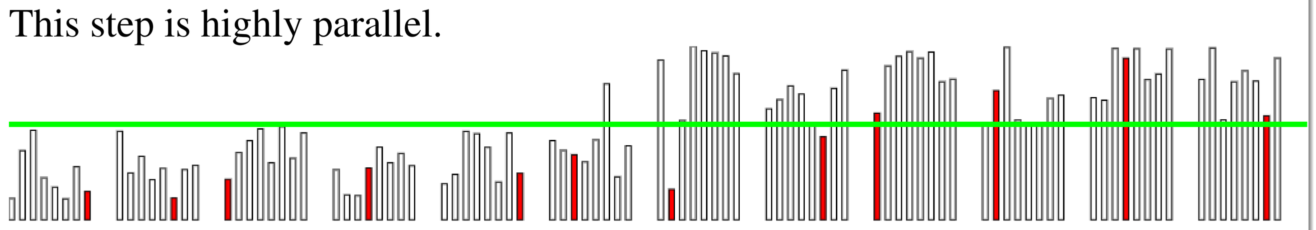
Logically partition the array into chunks of adjacent elements.



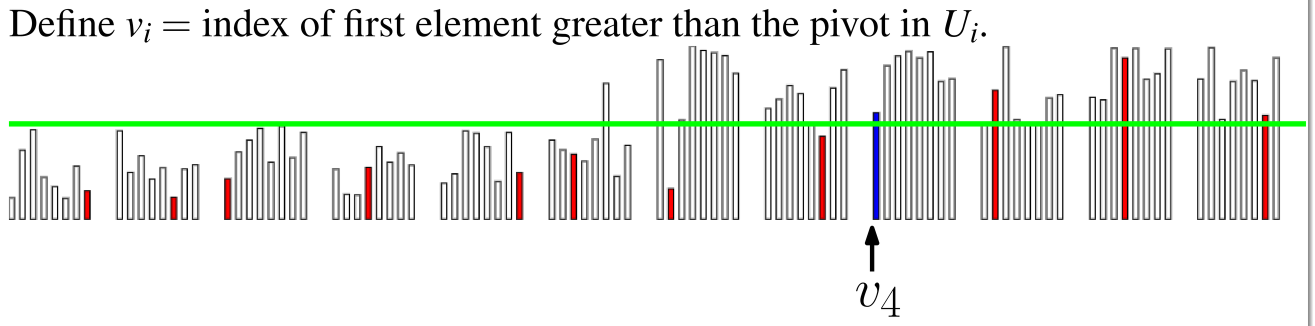
Form groups U_i that contain a random element from each chunk.
This randomization step was one of our key insights; it guarantees that the U_i's have similar compositions regardless of the input.



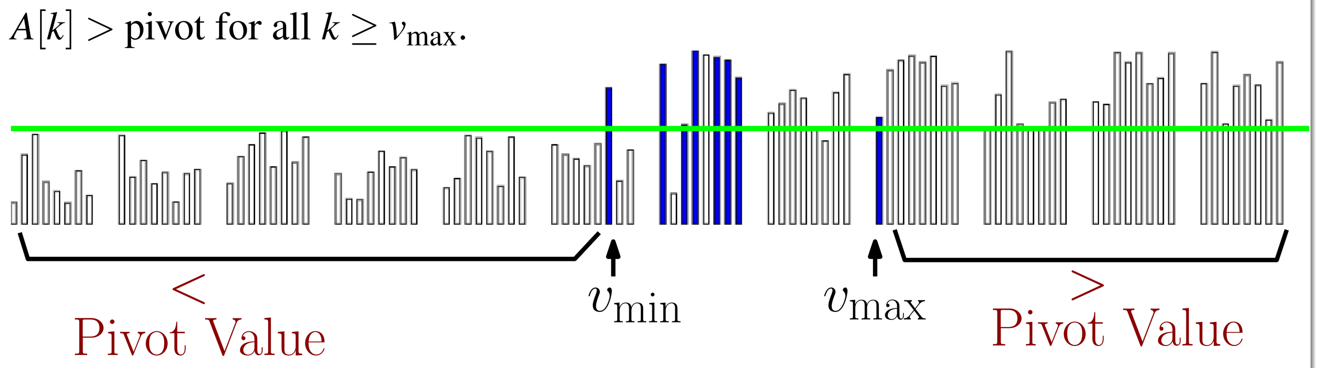
Perform serial partitions on each U_i in parallel over the U_i's.
This step is highly parallel.



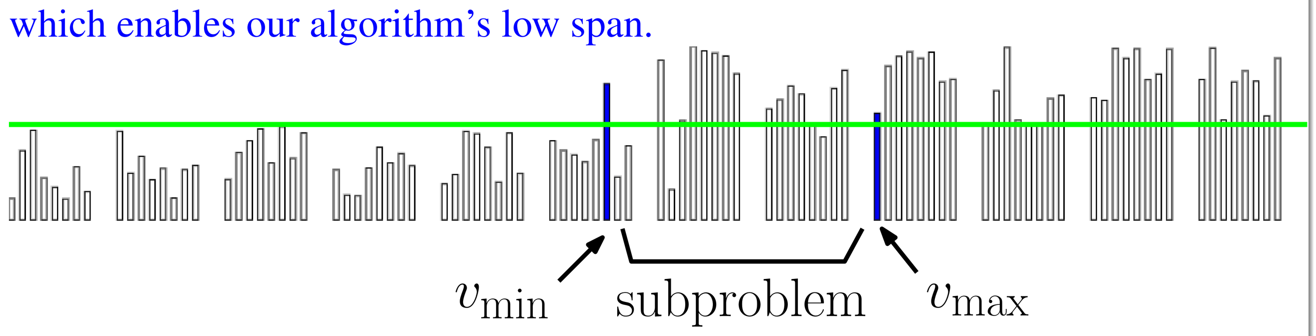
Define v_i = index of first element greater than the pivot in U_i.



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



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

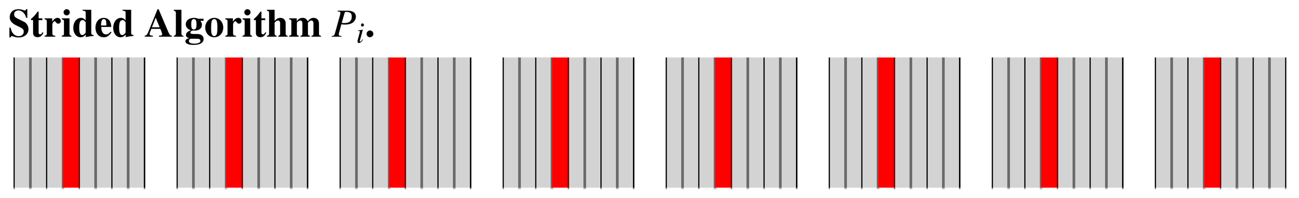


A Key Challenge

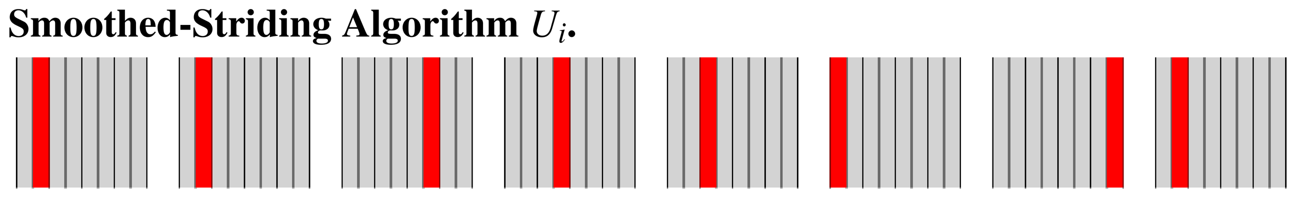
How do we store the U_i's if they are all random?

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

Strided Algorithm P_i.



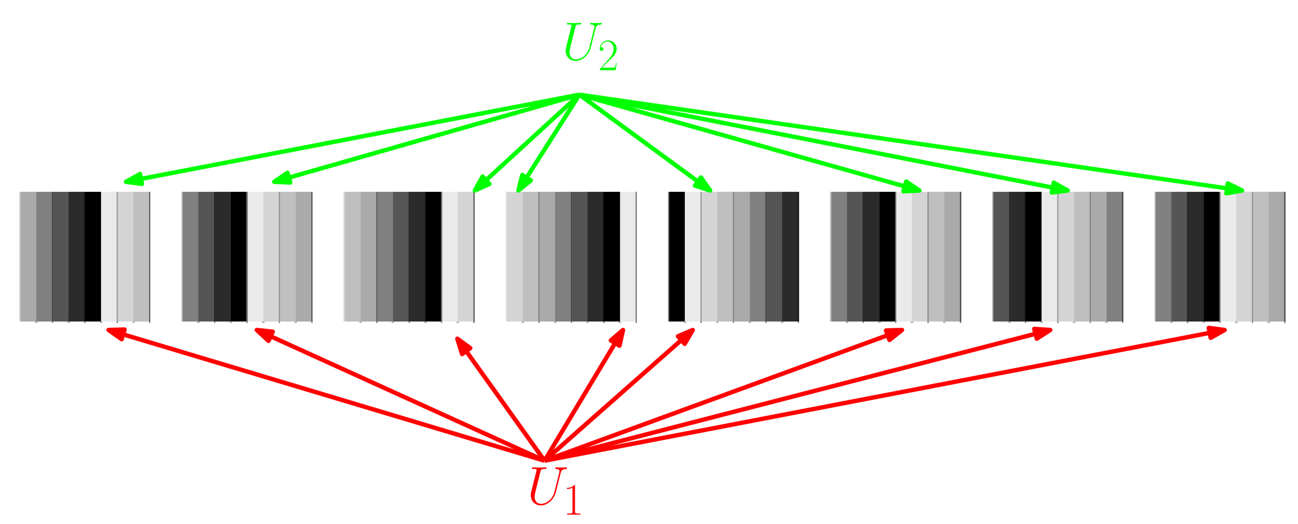
Smoothed-Striding Algorithm U_i.



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).



Partial Partition Step

The *Partial Partition Step* of the Smoothed Striding algorithm guarantees that all elements at index $i < v_{\min}$ have value $A[i] \leq$ pivot value, and all elements at index $i > v_{\max}$ have value $A[i] >$ pivot value. Thus, to completely partition the array a subarray of size $v_{\max} - v_{\min}$ must be partitioned. We prove the following proposition that bounds the size of this subarray:

Proposition:

Let $\varepsilon \in (0, 1/2)$ and $\delta \in (0, 1/2)$ such that $\varepsilon \geq \frac{1}{\mathrm{poly}(n)}$ and $\delta \geq \frac{1}{\mathrm{polylog}(n)}$. Suppose $s > \frac{\ln(n/\varepsilon)}{\delta^2}$. Finally, suppose that each processor has a cache of size at least $s + c$ for a sufficiently large constant c.

Then the Partial-Partition Algorithm achieves work $O(n)$; achieves span $O(b \cdot s)$; incurs $\frac{s+n}{b} + O(1)$ cache misses; and guarantees with probability $1 - \varepsilon$ that

$$v_{\max} - v_{\min} < 4n\delta.$$

Recursive Strategies

We propose two algorithms for solving the recursive subproblem:

- In the *Hybrid Smoothed Striding Algorithm* we recurse with a (Cache-Inneficient) In-Place Parallel-Partition algorithm, that has span $O(\log n \log \log n)$. With this recursive strategy we achieve span $O(\log n \log \log n)$ overall – which is within a $\log \log n$ factor of optimal – and incur fewer than $(n + o(n))/b$ cache misses – which is optimal up to low order terms– for appropriate parameter choices, with high probability in n.
- In the *Recursive Smoothed Striding Algorithm* we recurse with the Smoothed Striding algorithm. This algorithm achieves span $O(\log^2 n)$ which is worse than the other approahc, but this algorithm has the major benefit of simplicity to implement, while maintaining optimal cache behavior of $(n + o(n))/b$ for appropriate parameter choices, with high probability in n.

Analysis Overview

The proof of our proposition about the Parallel Partition Step proceeds along these lines:

- Let μ be the faction of elements of the array that are less than the pivot, and μ_i be the fraction of elements of U_i that are less than the pivot.
- All the μ_i have identical probability distributions, because any given element of the array is equally likely to be assigned to any U_i. Hence $\mathbb{E}[\mu_i] = \mu$.
- $|U_i| = \mathrm{polylog}\, n$, so a Hoeffding Bound (Chernoff Bound for random variable on $[0, 1]$ instead of on $\{0, 1\}$) guarantees that all U_i's will have μ_i's concentrated around μ with high probability in n.
- The concentration of μ_i's induces a concentration of v_i's.
- This guarantees that $v_{\max} - v_{\min}$ is small.

Pseudocode For the Algorithm

Recall:
A is the array to be partitioned, of length n.
We break A into chunks, each consisting of g cache lines of size b.
We create g groups $U_1, \ldots, U_g$ that each contain a single cache line from each chunk,
U_i's j-th cache line is the $(X[j] + i \bmod g + 1)$-th cache line in the j-th chunk of A.

```
procedure GET BLOCK START INDEX(X, g, b, i, j)    ▷ This procedure returns the index in A of the start of U′_i s j-th block.
    return b · ((X[j] + i mod g) + (j − 1) · g) + 1
end procedure

procedure PARALLELPARTITION(A, n, g, b)
    if g < 2 then
        serial partition A
    else
        for j ∈ {1, 2, ..., n/(gb)} do
            X[j] ← a random integer from [1, g]
        end for
        for all i ∈ {1, 2, ..., g} in parallel do          ▷ We perform a serial partition on all U_i's in parallel
            low ← GetBlockStartIndex(X,g,b,i,1)              ▷ low ← index of the first element in U_i
            high ← GetBlockStartIndex(X,g,b,i,n/(gb)) + b − 1   ▷ high ← index of the last element in U_i
            while low < high do
                while A[low] ≤ pivotValue do
                    low ← low+1
                    if low mod b ≡ 0 then     ▷ Perform a block increment once low reaches the end of a block
                        k ← number of block increments so far (including this one)
                        low ← GetBlockStartIndex(X,g,b,i,k)    ▷ Increase low to start of block k of G_i
                    end if
                end while
                while A[high] > pivotValue do
                    high ← high−1
                    if high mod b ≡ 1 then    ▷ Perform a block decrement once high reaches the beginning of a block
                        k ← number of block decrements so far (including this one)
                        k′ ← n/(gb) − k
                        high ← GetBlockStartIndex(X, g,b,i,k′) +b − 1    ▷ Decrease high to end of block k′ of G_i
                    end if
                end while
                Swap A[low] and A[high]
            end while
        end for
        Recurse on A[v_min], ..., A[v_max − 1]
    end if
end procedure
```