Cache-Efficient Parallel Partition Algorithms

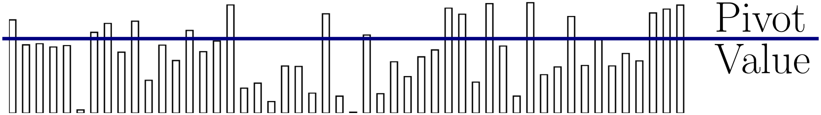
Alek Westover

MIT PRIMES

October 20, 2019

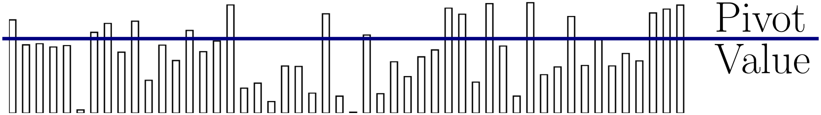
The Partition Problem

An unpartitioned array:

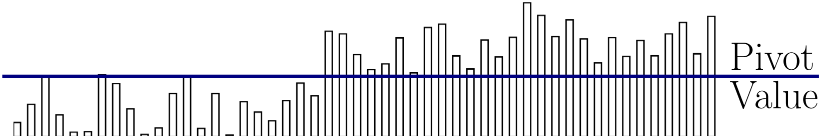


The Partition Problem

An unpartitioned array:



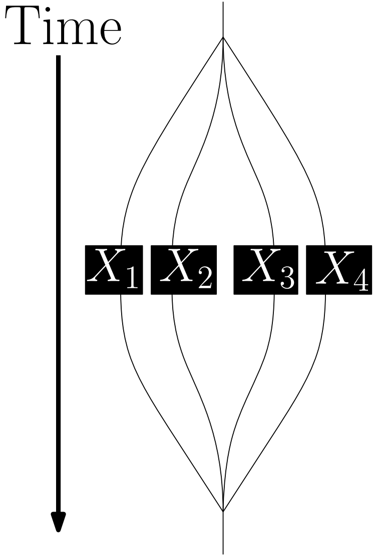
An array partitioned relative to a pivot value:



What is a Parallel Algorithm?

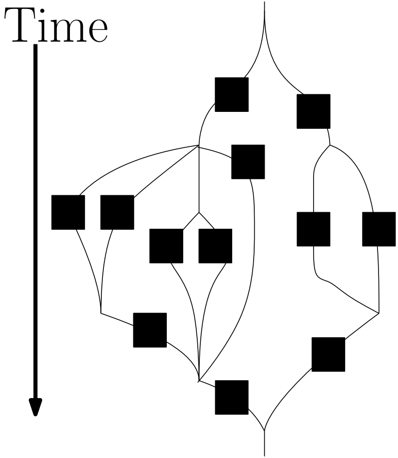
Fundamental primitive:
Parallel for loop

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

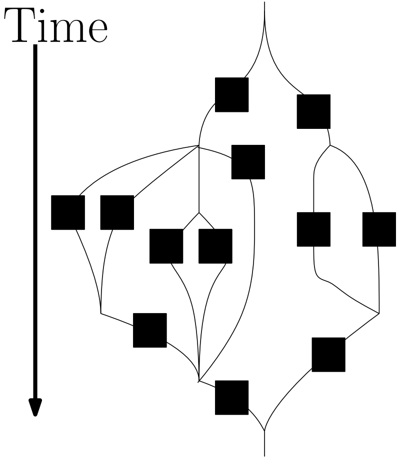


What is a Parallel Algorithm?

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

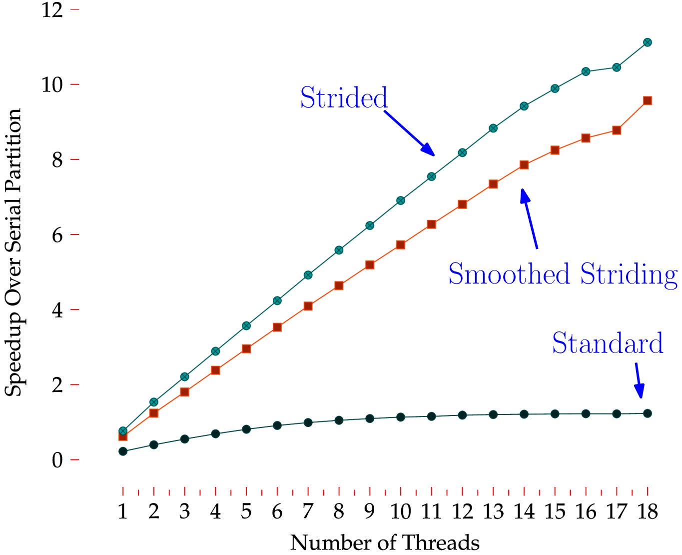
[Francis and Pannan, 92; Frias and Petit, 08]

- 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

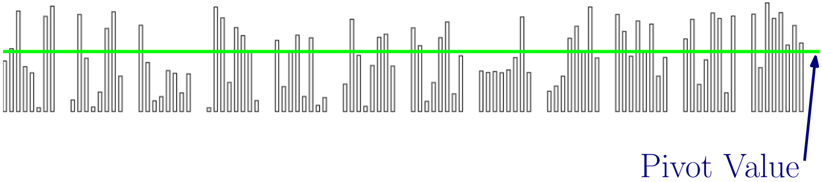
Smoothed-Striding Algorithm's performance



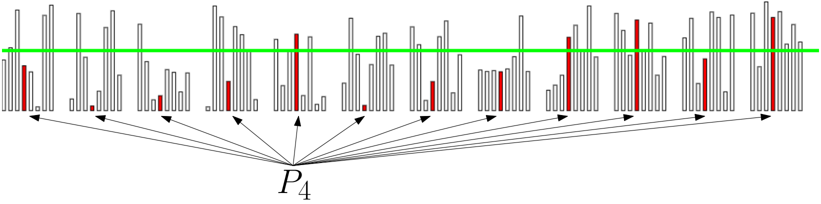
The Strided Algorithm

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

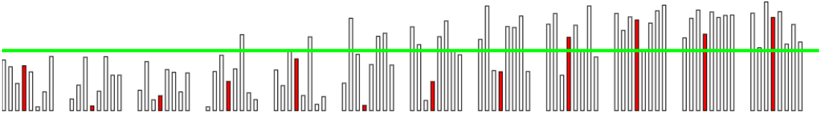
Logically partition the array into chunks of adjacent elements



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

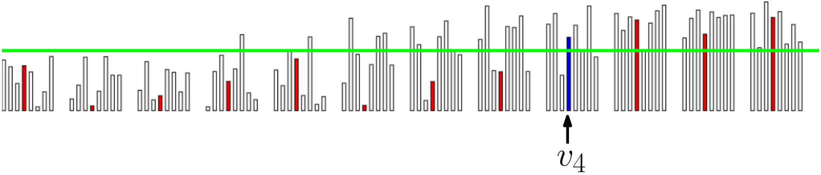


Perform serial partitions on each P_i in parallel over the P_i's

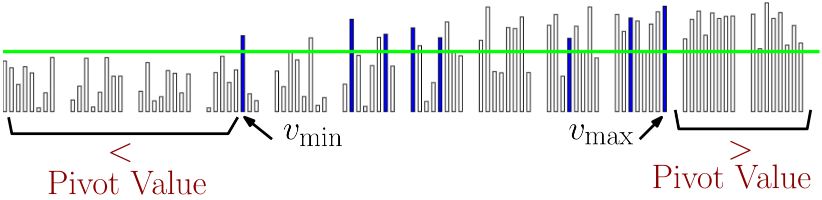


This step is highly parallel.

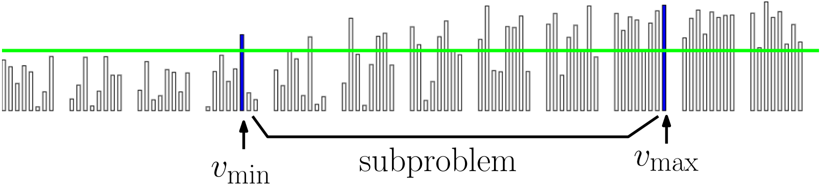
Define v_i = index of first element greater than the pivot in P_i



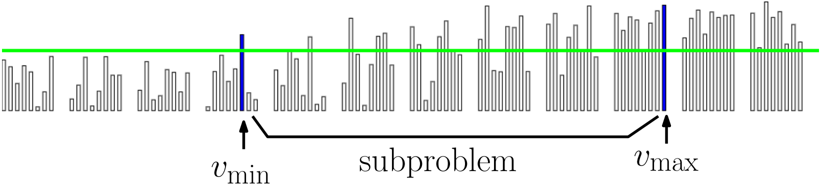
Identify leftmost and rightmost v_i



Final step: Recursively partition the subarray



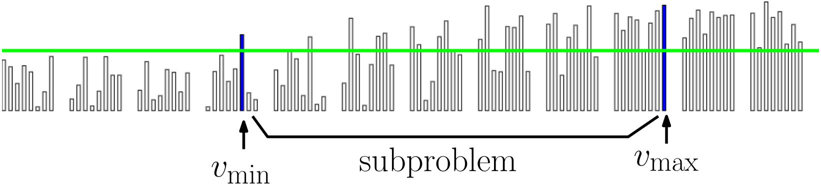
~~**Final step:** Recursively partition the subarray~~



- 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~~

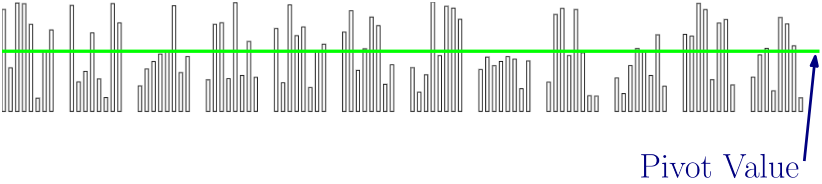


- Recursion is impossible!
- **Final Step:** Partition the subarray *in serial*.

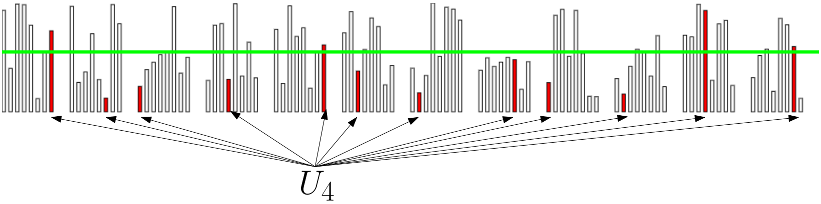
Subproblem Span $T_\infty \approx v_{\max} - v_{\min} \longleftarrow$ n in worst case.

The Smoothed-Striding Algorithm

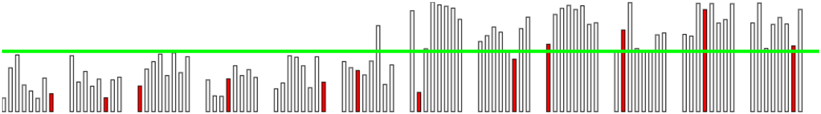
Logically partition the array into chunks of adjacent elements



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

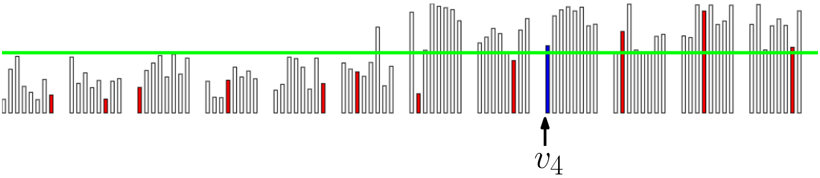


Perform serial partitions on each U_i in parallel over the U_i's

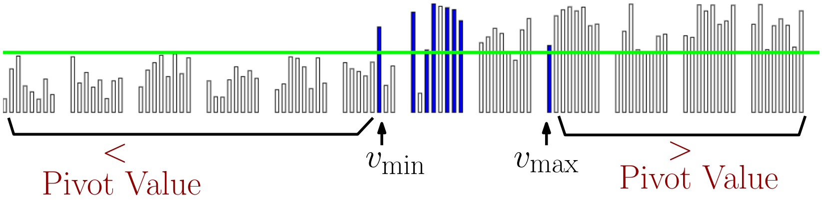


This step is highly parallel.

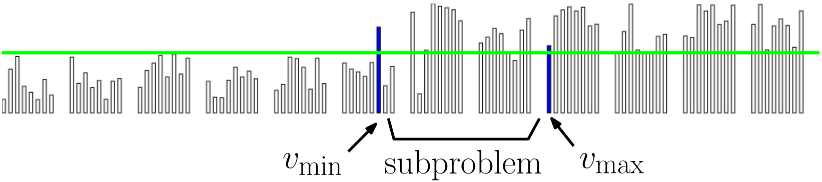
Define v_i = index of first element greater than the pivot in U_i



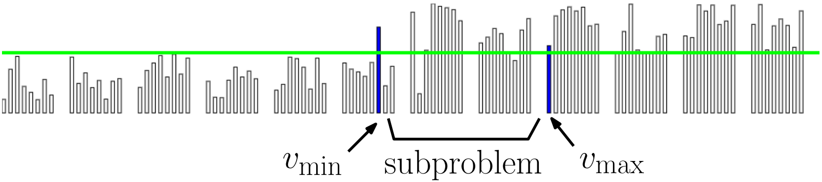
Identify leftmost and rightmost v_i



Final step: Recursively partition the subarray



Final step: Recursively partition the subarray



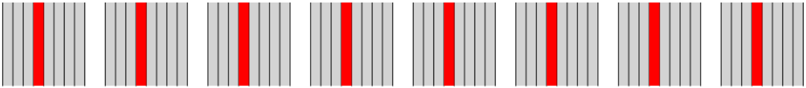
- Recursion is now possible!
- Randomness guarantees that $v_{\max} - v_{\min}$ is small

A Key Challenge

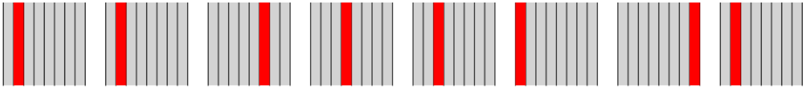
How do we store the U_i's if they are all random?

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

Strided Algorithm P_i.



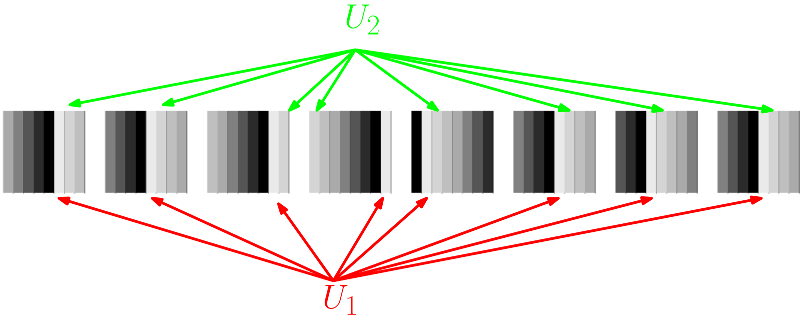
Smoothed-Striding Algorithm U_i.



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).



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

Why are atomic variables and Locks "undesirable"?

Algorithms with locks and atomic variables can be very fast.

However,

- They are often hardware specific
  (e.g. 128-bit fetch-and-add)
- Their scaling is unpredictable because threads have to queue when competing for access to the same variable.

Why are Atomic Variables and Locks "Undesirable"

An example that illustrates why their scaling is unpredictable: Consider the problem of summing n integers.

- With our model of parallelism a simple divide and conquer strategy achieves the optimal span $O(\log n)$.
- With atomic-fetch-and-add, span $O(1)$ seems achievable by having each of n processors fetch-and-add an element to the sum, but this really takes time $O(n)$ because only a single thread can have access to the sum variable at a time.

Chernoff Bound

Chernoff Bound: "If you flip polylog(n) fair coins then, with high probability in n, the number of heads will be tightly concentrated around $\frac{1}{2}\operatorname{polylog} n$".

More formally: The probability of a random variable X deviating from its mean μ by more than a constant multiple δ of μ is exponentially small in $\delta^2\mu$. That is,

$$P(|X-\mu| \leq \delta\mu) \geq 1 - 2 \cdot e^{-\delta^2\mu/2}$$

Big Picture of the Analysis

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.

- Each U_i has a random element from each chunk of the array, so each element of each U_i is randomly either greater than or less than the pivot, with probabilities $1 - \mu, \mu$.
- $|U_i| = \operatorname{polylog} n$, so a Chernoff Bound guarantees that all U_i's will have μ_i's similar to $\mathbb{E}[\mu_i] = \mu$ 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.

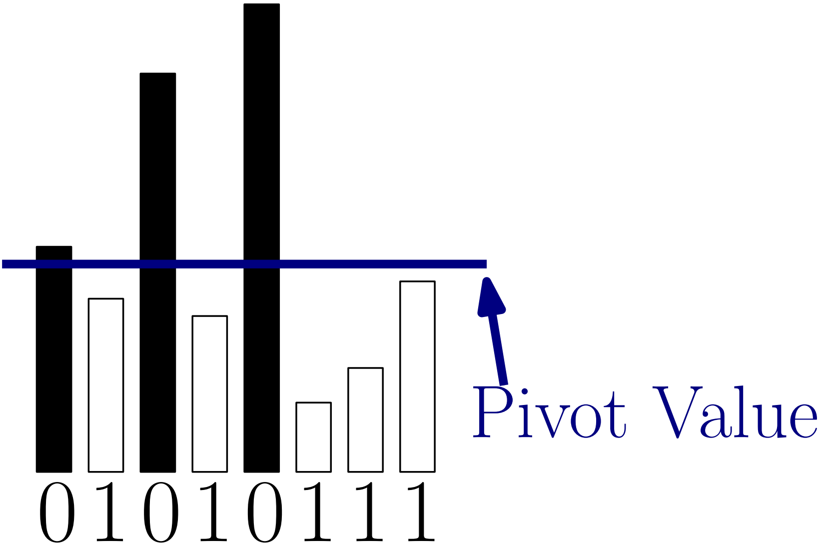
With High Probability in n

With probability

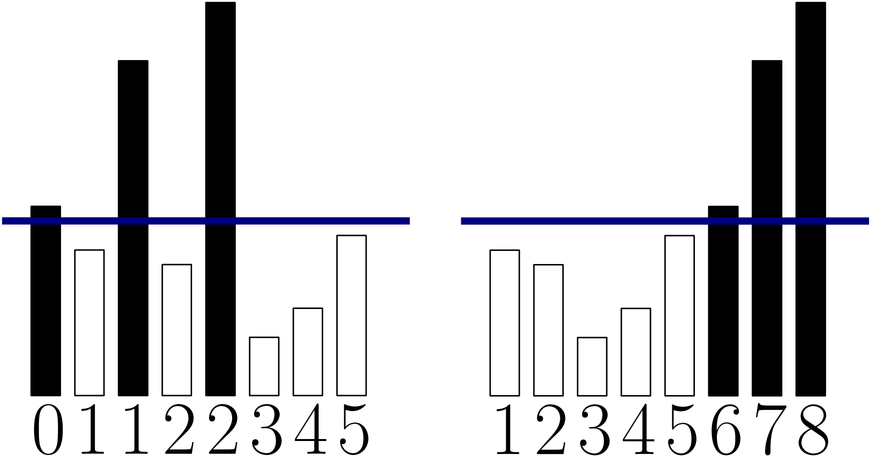
$$1 - \frac{1}{n^c}$$

for c of our choice.

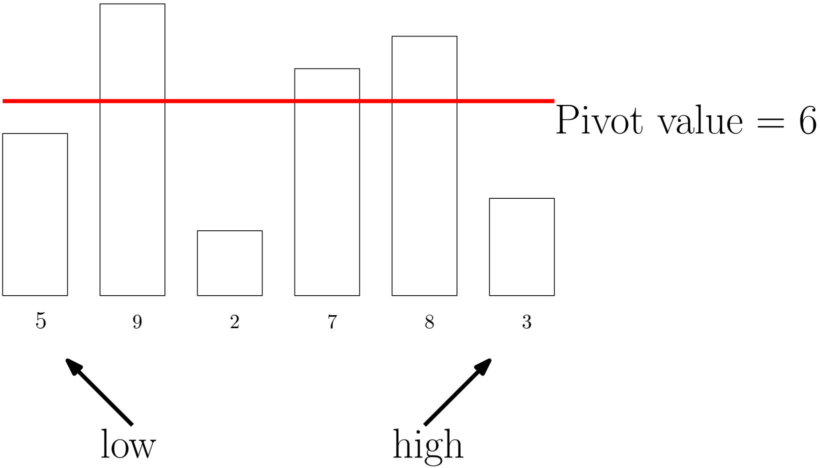
The Standard Parallel Partition Algorithm



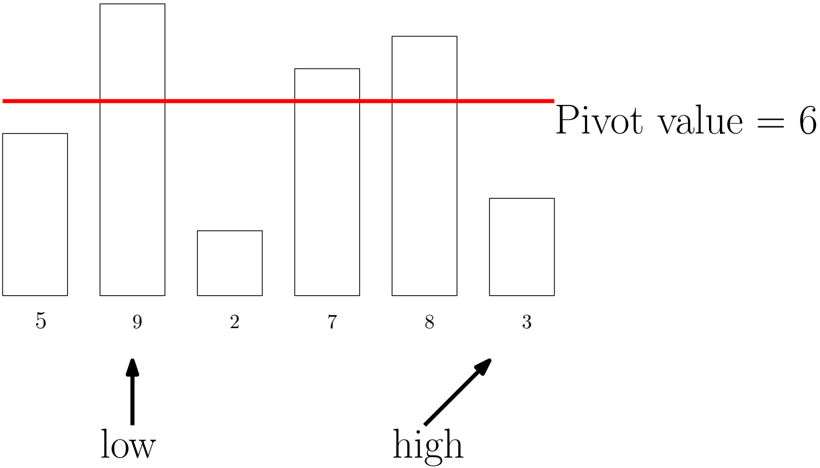
The Standard Parallel Partition Algorithm



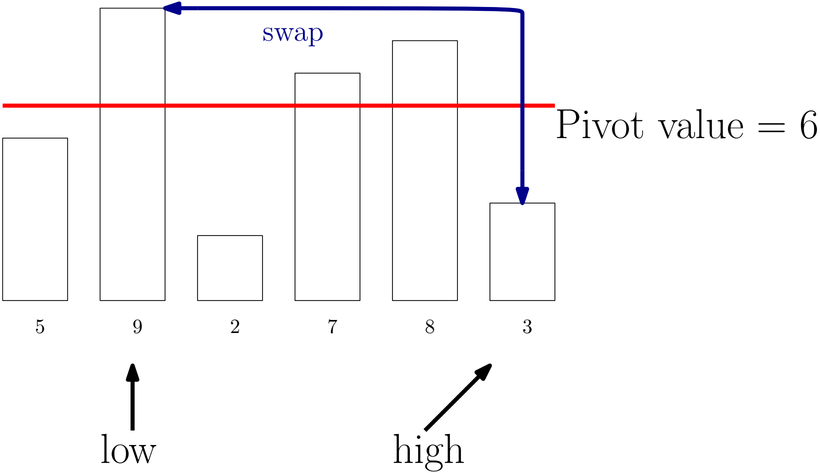
The Serial Partition Algorithm



The Serial Partition Algorithm



The Serial Partition Algorithm



The Serial Partition Algorithm