Recursive Strategies

The smoothed striding algorithm allows for the subproblems to be solved with recursion. There are two algorithms that can be used in recursion:

- In the *Recursive Smoothed Striding Algorithm* we recurse with the smoothed striding algorithm. This gets span $O(\log^2 n)$ which is not great, but this algorithm is very simple to implement, while maintaining optimal cache behavior.
- We can also recurse with a Cache-Ineficient In-Place Parallel-Partition algorithm, that was developed concurrently to our algorithm. Doing so we achieve span $O(\log n \log\log n)$ and optimal cache behavior.

Formal Theoretical Results

Proposition:
Let $\varepsilon \in (0, 1/2)$ and $\delta \in (0, 1/2)$ such that $\varepsilon \geq \frac{1}{\text{poly}(n)}$ and $\delta \geq \frac{1}{\text{polylog}(n)}$. 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

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

We have the following corrolaries based on the two recursive strategies:
Corrolarry:
Suppose $b \leq o(\log\log n)$. Then the Cache-Efficient Full-Partition Algorithm using $\delta = \Theta\left(\sqrt{b/\log\log n}\right)$, achieves work $O(n)$, and with high probability in n, achieves span $O(\log n \log\log n)$ and incurs fewer than $(n + o(n))/b$ cache misses.
Corrolarry:
With high probability in n, the Recursive Smoothed Striding Algorithm using parameter $\delta = 1/\sqrt{\log n}$: achieves work $O(n)$, attains span $O(b \log^2 n)$, and incurs $n/b \cdot (1 + O(1/\sqrt{\log n}))$ cache misses.

Analysis Overview

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.

- NOPEEach 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 $1 - \mu, \mu$.
- $|U_i| = \text{polylog}\, n$, 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.

Pseudocode

Figure: The Smoothed Striding 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
```