Cache-Efficient Parallel Partition

Alek Westover

MIT PRIMES

2019

Introduction

- What is the partition problem?
- Why is it interesting to solve?

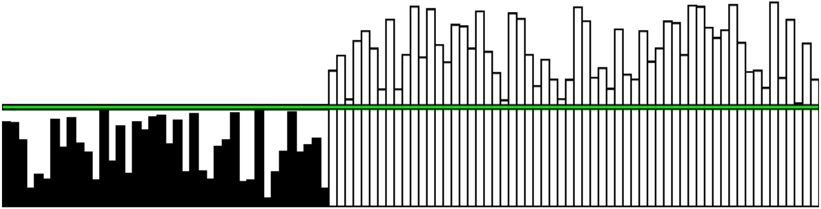

Figure: A depiction of a partitioned array where rectangle heights represent values in the array, and the green bar represents the value the elements are partitioned relative to. Note that the elements colored black are called predecessors and the elements colored white are called successors.

The Partition Problem

- Given an array A of size n and a decider function that labels each $A[i]$ as either a predecessor or a successor, a partition algorithm must reorder the array such that for all i such that $A[i]$ is a predecessor and for all j such that $A[j]$ is a successor $i < j$.
- We will use a decider funciton that partitions the array relative to some "pivot value". That is, we use a decider function that, given some pivot value, labels $A[i]$ a predecessor if and only if $A[i] \leq$ pivot value.

Parallel Partition Motivation

- Parrallel partition plays a central role in Parallel Quicksort
- Parallel partition is interesting in its own right
- Parallel partitions are used in performing filter opperations

Preliminaries

- Serial Partition
- What is parallel processing?
- Standard Parallel Partition (not in-place)
- Work, Span, and Brent's theorem
- With High Probability
- Bill's alg and bandwidth bound
- Cache Misses

Serial Partition

1. Initialize **low** to point at the beginning of the array, and initialize **high** to point at the end of the array
2. Increment low until $A[\text{low}]$ is a successor
3. Decrement high until $A[\text{high}]$ is a predecessor
4. Swap values $A[\text{low}]$ and $A[\text{high}]$ in the array
5. Repeat steps 3-5 until high $\geq$ low which means that all elements in the array have been processed
6. If $A[\text{low}]$ is a predecessor increment $A[\text{low}]$ by 1 so that $A[\text{low}]$ is the first successor in A, which is now partitioned

This has work $O(n)$, is in-place, and incurs $n + O(1)$ cache misses.

Standard Parallel Partition (not in-place)

- Parallel prefix sum

Work, Span, Brent's Theorem

- The **work** of an algorithm, denoted T_1 is its running time with a single processor.
- The **span** of an algorithm, denoted T_∞ is its running time with an infinite number of processors.
- Clearly $T_p \geq \frac{T_1}{p}$ and $T_p \geq T_\infty$.
- Brent's theorem gives an upper bound for an algorithm's running time on p processors, denoted T_p, from the alghrithms work and span.

Theorem (Brent's theorem)

$$T_p \leq \frac{T_1}{p} + T_\infty.$$

With High Probability

whp means

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

for c of our choice.

Bill's alg and Memory Bandwidth Bound

Memory bandwidth bound is bad. Thus minimizing cache misses is good.

Cache Misses

defn

The Strided Algorithm

- Description
- Guarantees

Description

- **The Partial Partition Step.** Let $g \in \mathbb{N}$ be a parameter, and assume for simplicity that $gb \mid n$. Partition the array A into $\frac{n}{gb}$ chunks $C_1, \ldots, C_{n/gb}$, each consisting of g cache lines of size b. For $i \in \{1, 2, \ldots, g\}$, define P_i to consist of the i-th cache line from each of the chunks $C_1, \ldots, C_{n/gb}$. One can think of the P_i's as forming a strided partition of array A, since consecutive cache lines in P_i are always separated by a fixed stride of $g - 1$ other cache lines.
  The first step of the algorithm is to perform an in-place serial partition on each of the P_is, rearranging the elements within the P_i so that the predecessors come first.
- **The Serial Cleanup Step.** For each P_i, define the **splitting position** v_i to be the position in A of the final predecessor in (the already partitioned) P_i. Define $v_{\min} = \min\{v_1, \ldots, v_g\}$ and define $v_{\max} = \max\{v_1, \ldots, v_g\}$. Then the second step of the algorithm is to perform a serial partition on the sub-array $A[v_{\min}], \ldots, A[v_{\max} - 1]$. This completes the full partition.

Guarantees

Note that the partial partition step has parallelism, and requires work $\Theta(n)$ and span $\Theta(n/g)$. Note that the Cleanup Step of the Strided Algorithm has no parallelism, and thus has span $\Theta(v_{\max} - v_{\min})$. In general, this results in an algorithm with linear-span (i.e., no parallelism guarantee). When the number of predecessors in each of the P_i's is close to equal, however, the quantity $v_{\max} - v_{\min}$ can be much smaller than $O(n)$. For example, if $b = 1$, and if each element of A is selected independently from some distribution, then one can use Chernoff bounds to prove that with high probability in n, $v_{\max} - v_{\min} \leq O(\sqrt{n \cdot g \cdot \log n})$. The full span of the algorithm is then $\tilde{O}(n/g + \sqrt{n \cdot g})$, which optimizes at $g = n^{1/3}$ to $\tilde{O}(n^{2/3})$. Since the Partial Partition Step incurs only n/b cache misses, the full algorithm incurs $n + \tilde{O}(n^{2/3})$ cache misses on a random array A.

Using Hoeffding's Inequality in place of Chernoff bounds, one can obtain analogous bounds for larger values of b; in particular for $b \in \text{polylog}(n)$, the optimal span remains $\tilde{O}(n^{2/3})$ and the number of cache misses becomes $n/b + \tilde{O}(n^{2/3}/b)$ on an array A.

The Smoothed Striding Algorithm

- Partial Partition Description
- Partial Partition Analysis
- From Partial Partition to Full Partition
- Hybrid Smoothed Striding Algorithm
  - Theorem
  - Corollary
- Recursive Smoothed Striding Algorithm
  - Theorem
  - Corollary

Partial Partition Description

- Set each of $X[1], \ldots, X[s]$ to be uniformly random and independently selected elements of $\{1, 2, \ldots, g\}$. For $i \in \{1, 2, \ldots, g\}$, and for each $j \in \{1, 2, \ldots, s\}$, define

  $$G_i(j) = (X[j] + i \pmod g) + (j-1)g + 1.$$

  Using this terminology, we define each U_i for $i \in \{1, \ldots, g\}$ to contain the $G_i(j)$-th cache line of A for each $j \in \{1, 2, \ldots, s\}$. That is, $G_i(j)$ denotes the index of the j-th cache line from array A to be contained in U_i.
  Note that, to compute the index of the j-th cache line in U_i, one needs only the value of $X[j]$. Thus the only metadata needed by the algorithm to determine the $U_1, \ldots, U_g$ is the array X. If $|X| = s = \frac{n}{gb} \leq \text{polylog}(n)$, then the algorithm is in place.
- The algorithm performs an in-place (serial) partition on each U_i (and performs these partitions in parallel with one another). In doing so, the algorithm, also collects $v_{\min} = \min_i v_i$, $v_{\max} = \max_i v_i$, where each v_i with

Partial Partition Step Analysis

Let $\epsilon \in (0, 1/2)$ and $\delta \in (0, 1/2)$ such that $\epsilon \geq \frac{1}{\text{poly}(n)}$ and $\delta \geq \frac{1}{\text{polylog}(n)}$. Suppose $s > \frac{\ln(n/\epsilon)}{\delta^2}$. 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 $1 - \epsilon$ that

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

From Partial Partition to Full Partition

We will use Proposition 16 as a tool to analyze the Recursive and the Hybrid Smoothed Striding Algorithms.
Rather than parameterizing the Partial Partition step in each algorithm by s, Proposition 16 suggests that it is more natural to parameterize by ϵ and δ, which then determine s.
We will assume that both the hybrid and the recursive algorithms use $\epsilon = 1/n^c$ for c of our choice (i.e. with high probability in n).
Moreover, the Recursive Smoothed Striding Algorithm continues to use the same value of ϵ within recursive subproblems (i.e., the ϵ is chosen based on the size of the first subproblem in the recursion), that way the entire algorithm succeeds with high probability in n.
For both algorithms, the choice of δ results in a tradeoff between cache misses and span. For the Recursive algorithm, we allow for δ to be chosen arbitrarily at the top level of recursion, and then fix $\delta = \Theta(1)$ to be a sufficiently small constant at all levels of recursion after the first; this guarantees that we at least halve the size of the problem size between recursive iterations[3]. Optimizing δ further (after the first level of recursion) would only affect the

Hybrid Algorithm Analysis - General Theorem

Theorem

The Hybrid Smoothed Striding Algorithm algorithm using parameter $\delta \in (0, 1/2)$ *satisfying* $\delta \geq 1/\operatorname{polylog}(n)$*: has work* $O(n)$*; achieves span*

$$O\left(\log n \log \log n + \frac{b \log n}{\delta^2}\right),$$

with high probability in n; and incurs fewer than

$$(n + O(n\delta))/b$$

cache misses with high probability in n.

Hybrid Algorithm Analysis - Corollary for specific parameter settins

An interesting corollary of the above theorem concerns what happens when b is small (e.g., constant) and we choose δ to optimize span.

Corollary (Corollary of Theorem 1)

Suppose $b \leq o(\log\log n)$. Then the Cache-Efficient Full-Partition Algorithm 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.

Recursive Algorithm Analysis - General Theorem

Theorem

With high probability in n, the Recursive Smoothed Striding algorithm using parameter $\delta \in (0, 1/2)$ satisfying $\delta \geq 1/\operatorname{polylog}(n)$: achieves work $O(n)$, attains span

$$O\left(b\left(\log^2 n + \frac{\log n}{\delta^2}\right)\right),$$

and incurs $(n + O(n\delta))/b$ cache misses.

Recursive Algorithm Analysis - Corollary for specific parameter settings

A particularly natural parameter setting for the Recursive algorithm occurs at $\delta = 1/\sqrt{\log n}$.

Corollary (Corollary of Theorem 3)

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.

Expiriments

- Strided Algorithm vs Smoothed Striding algorithm
- Cache misses

Acknowledgments

I would like to thank

- The MIT PRIMES program
- William Kuszmaul, my PRIMES mentor
- My parents