Cache Efficient Parallel Partition

Alek Westover

June 26, 2019

Abstract

We present a cache-efficient algorithm for parallel partition that has work $O(n)$ and span $O(\log n \log\log n)$. This imediately gives us a cache-efficient quicksort algorithm with work $O(n \log n)$ and span $O(\log^2 n \log\log n)$.

We also present an exceptionally simple cache-efficient algorithm for parallel partition that has work $O(n)$ and span $O(\log^2 n)$. Despite its slightly worse span, we show that this algorithm performs well in practice.

1 Introduction

Serial Partition Problem. We breifly describe the serial partition problem and the solution which is used extensively in our parallel partition algorithms. The serial partition problem takes in an array, and then reorders the array according to a decider function and returns the number of predecessors in the array. Upon completion of the serial partition algorithm, which runs fully in-place, if k is the number of predecessors in the array then for all $i < k$, $A[i]$ is a predecessor, and for all $i \geq k$, $A[i]$ is a successor. In the case of the decider function "$A[i] \leq$ pivot value", as is the case in the partition used for quicksort to sort an array in ascending order, this corresponds to placing all elements with values less than or equal to the pivot value before the elements with values greater than or equal to the pivot value. Thus, by recursively sorting the predecessors and the successors, the entire array is sorted. This is because the predecessor portion of the array is sorted, the successor portion of the array is sorted, and the predecessors occur before the successors, so the whole array is sorted. The serial partition algorithm works as follows:

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[low] is a successor
3. Decrement high until A[high] is a predecessor
4. Swap values A[low] and A[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[low] is a predecessor increment A[low] by 1 so that A[low] is the first successor in A, which is now partitioned

See Algorithm 1 for a pseudocode implementation of this algorithm. Note that the serial partition described makes a single pass over the n elements in the array meaning that it incurs $O(n)$ cache misses, and also it has running time (i.e. work) $O(n)$.

Further Preliminaries. We introduce some key ideas in analysis of the algorithms presented in this article. Many of the algorithms we present have worst case performances that are highly unlikely to be realized. For instance, if in quicksort a random selection of a pivot value results in only 1 element being labelled a successor, then the recursive subproblems are very

Algorithm 1 Serial Partition

```
low ← 0; high ← n − 1
while low < high do
    while A[low] ≤ pivotValue do
        low ← low + 1
    end while
    while A[high] > pivotValue do
        high ← high − 1
    end while
    Swap A[low] and A[high]
end while
if A[low] ≤ pivotValue then
    low ← low + 1
end if
```

unevenly sized. If the size of the problems is $n, n-1, n-2, \ldots, 2$ then quicksort performs at its worst case performance, which is $O(n^2)$. This however is very unlikely. We can formalize this with the notion of saying that the event does not happen ***with high probability***. An event is said to happen with high probability in n if the probability of the event occurring is $1 - n^{-c}$ for some $c > 0$. Intuitively, if we can show that our algorithm has a certain span with high probability, then we are confident that the algorithm achieves this span.

We define throughout our analysis μ to be the fraction of predecessors in the array A. In implementing the partition algorithm we do not have control over the pivot value, it is an input to our problem. The pivot value together with the array A and the decider function determine the fraction μ. Throughout our analysis we assume that μ is a constant not dependent on n.

Factors to Consider. We now describe the desirable characteristics of a parallel partition algorithm. The ***work*** of an algorithm, T_1, is the running time of the algorithm in serial. The ***span*** of an algorithm, T_∞ is the running time of the algorithm when run on an infinite number of processors. The running time of the algorithm on p processors, T_p, is determined by its work and span. This is formally expressed in Brent's theorem which asserts that

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

Thus optimizing the work and span of an algorithm will optimize its performance on p processors. Another important consideration is auxiliary memory usage. It is desirable to be as close to in-place as possible because a user might not have any extra space that our algorithm can use if the array to be partitioned takes up nearly all of their machine's memory. Finally, it is important to analyze whether or not the algorithm is cache efficient, which dependends on if the algorithm is in-place, because this determines the ***memory bandwidth bound*** on the algorithm. A ***cache miss*** is when the algorithm must access a value that is not in the cache already. Doing this takes time, and the memory banks on a computer can only service a limited number of memory access attempts per interval of time, so if there are lots of requests to the memory bank happening then the algorithm slows down, and we say it is memory bandwidth bound. For the partition problem each element must be read at least once, so there are at least n cache misses. We consider all of these properties in our development of a parallel partition algorithm, and our algorithm does well according to all of these metrics.

Prior Work. Prior work has been done on the parallel partition problem. William Kuszmaul developed an in-place parallel partition algorithm. Unfortunately, despite theoretical

guarantees, his algorithm was outperformed by an optimized but higher span algorithm. The problem with this parallel partition algorithm in terms of speed in practice is its memory bandwidth bound. The reason for the memory bandwidth bound is the number of cache misses that the algorithm incurs. Thus reducing the number of cache misses is a promising place to look in order to speed up parallel partition, which motivates our algorithm.

2 Cache Efficient Parallel Partition for Random Inputs

We now present an algorithm for parallel partition developed by Francis and Pannan, which we call the ***Strided Algorithm***. The Strided Algorithm is thought to use very few passes over the array, although we could not find any accurate theoretical analysis of the algorithm. Having a low number of cache misses is desirable because it makes the algorithm run much faster, overcoming the problem of memory bandwidth bound. The Strided Algorithm does not achieve a small number of cache misses for all inputs, but it does achieve this when used on some inputs, for example random inputs. We introduce our model of cache misses and then prove that the Strided Algorithm incurs very few cache misses for random inputs.

Strided Algorithm Description. Now we describe the algorithm. The algorithm is given an input array $A[0], A[1], \ldots, A[n-1]$ where for all i, $A[i] \in \{0, 1\}$. The values in the array A were determined independently, and randomly with an equal probability of being either 0 or 1. The Strided Algorithm first logically partitions A into parts $P_0, P_1, \ldots, P_{t-1}$ where

$$P_i = \{A[i], A[i+t], \ldots, A[i + (\frac{n}{t} - 1)t]\}.$$

Note that we do not need to expend any extra memory to store each part P_i because the part is determined by a formula that can be used to access elements in P_i. The length of $P_i \approx n/t$. This is approximate because if $n \mod t \not\equiv 0$ then some of the P_is have 1 less element than others.

Define x_i to be the number of predecessors in P_i. The algorithm performs a serial partition of each P_i in parallel (on all the $P_i s$ at once). Note that this serial partition is slightly different than a typical serial partition because the elements in P_i are not adjacent in the array A, so to go from one value in P_i to the next we must add t rather than simply 1, and in decrementing we decrement by t instead of by 1. Partitioning P_i makes the first x_i elements of P_i predecessors. Define $v_i = t \cdot x_i + i \approx t \cdot x_i$. The index v_i is the location in A of the first successor in group P_i, because by construction P_i consists of elements that are t apart in the array, so by scaling the index relative to the part P_i by t (and shifting by i so that the first element is aligned) we get the index into A. Define $v_{min} = \min_i v_i$ and $v_{max} = \max_i v_i$. Note that after the partitioning all elements in the array A with index less than v_{min} are predecessors, and all elements after and including v_{max} are successors, so to finish the partitioning of the array A we only need to partition a subarray of size $v_{max} - v_{min}$. Note that we cannot recursively apply the Strided Algorithm to solve the sub problem, because it would behave very poorly on the subarray, and also because, as we demonstrate, the size of the subarray should be very small relative to n. Also note that if we can show that $v_{max} - v_{min}$ is small relative to n then

the partition algorithm uses very few passes over the array.

Strided Algorithm Analysis. We now prove the following theorem about the Strided Algorithm's performance:

Theorem 1. *Given an array* $A[0], A[1], \ldots, A[n-1]$ *where each* $A[i]$ *is chosen randomly independently to be either* 0 *or* 1*, and a decider function that labels each element with value* 0 *a predecessor, and each element with the value* 1 *a successor, the Strided Algorithm partitions the array in span that is, with high probability in* n*,* $O(\frac{n}{t} + \sqrt{n \cdot t \cdot \log n})$ *where* t *is a parameter of our choice indicating the number of parts that we break* A *into, incurring* $n \cdot (1 + o(1))$ *cache misses.*

To do this, we first prove the following lemma:

Lemma 1. *With high probability in* n*, the size of the recursive subproblem,* $v_{max} - v_{min}$*, is* $O(\sqrt{t \cdot n \log n})$.

Proof of Lemma 1.

A Single P_i. To prove this, we use Chernoff Bounds, which give bounds on the probability of a deviation of a random variable from its mean (expected value) μ by more than a certain amount. We first bound the number of predecessors x_i in each part P_i, which allows us to bound v_i, the index in the array A of the first successor in P_i after P_i has been partitioned.

The Multiplicative Chernoff bound bounds the probability of a random variable such as x_i deviating by more than $\delta \cdot \mu$ from its mean. Specifically it says,

$$P(x_i > \mu(1+\delta)) = O(e^{-\mu \cdot \delta^2/3}).$$

We want to know what value of δ makes $x_i \leq \mu(1+\delta)$ with high probability. To find the smallest δ satisfying this, we must solve

$$O(e^{-\mu \cdot \delta^2/3}) = n^{-c},$$

where the right hand side comes from the definition of an event being false with high probability, and the left hand side comes from the probability guaranteed by the Chernoff bound. Solving, we obtain

$$\log e^{-\mu \cdot \delta^2/3} = \log n^{-c},$$

which upon isolating δ yields,

$$\delta = O\left(\sqrt{\frac{\log n}{\mu}}\right).$$

Because x_i is the number of predecessors in P_i, and $|P_i| \approx \frac{n}{t}$, the expectation of x_i is $\mu = \frac{n}{2t}$ because the values in A were generated independently at random, with $\frac{1}{2}$ probability of being predecessors. Thus we can assert that

$$\delta = O\left(\sqrt{\frac{t \log n}{n}}\right).$$

By a nearly identical argument with the Chernoff bound for the probability of x_i deviating below the mean μ by more than $\mu \cdot \delta$ we get the same result: that deviation by more than $\mu \cdot \delta$ does not happen with high probability.

Thus we have that with high probability

$$|x_i - \mu| = O\left(\sqrt{\frac{t \log n}{n}}\right) \cdot \mu.$$

Once again we can substitute in $\mu = \frac{n}{2t}$ to get,

$$|x_i - \mu| = O\left(\sqrt{\frac{t \log n}{n}}\right) \cdot \frac{n}{2t}.$$

This simplifies to,

$$\left|x_i - \frac{n}{2t}\right| = O\left(\sqrt{\frac{n \log n}{t}}\right).$$

To convert this index to an index into the array A rather than an index in P_i, we multiply by t (the gap between the indices into the array A of elements in P_i) obtaining,

$$\left|t \cdot x_i - \frac{n}{2}\right| = O(\sqrt{t \cdot n \log n}).$$

Equivalently, we can use our name v_i for the index into the array A to write this as

$$\left|v_i - \frac{n}{2}\right| = O(\sqrt{t \cdot n \log n}).$$

We now need to move the discussion from just looking at P_i to looking at all P_is.

Union Bound over all P_i. Now, we apply the union bound for with high probability events. It states that the union of a set of events that are false with high probability is still false with high probability. Using the union bound allows us to go from saying

for all i, with high probability
$$|v_i - \frac{n}{2}| = O(\sqrt{t \cdot n \log n}),$$

to saying

with high probability, for all i
$$|v_i - \frac{n}{2}| = O(\sqrt{t \cdot n \log n}).$$

This establishes that, with high probability,

$$v_{max} - v_{min} = O(\sqrt{t \cdot n \log n}).$$

□

Using this lemma we can now prove Theorem 2.

Proof of Theorem 2.
Span. There are 2 contributions to the span of our algorithm. First, there is a contribution of $\frac{n}{t}$ for performing serial partitions on each P_i in parallel because serial partition has a running time that is on the order of the input array, and $|P_i| = \frac{n}{t}$. Second, there is a contribution from performing a serial partition on the subarray of size $v_{min} - v_{max}$ which is $O(\sqrt{t \cdot n \log n})$ by Lemma 1. Thus the total span is

$$T_\infty = O(n/t + \sqrt{t \cdot n \log n}).$$

We can choose t in order to minimize span. To minimize span, ignore the log factor and set

$$n/t = \sqrt{t \cdot n} \implies t = n^{1/3}.$$

This yields the minimum span, because if either term could be decreased, then the function would not be at a minimum, so the terms must be equal. This yields a span of $O(n^{2/3})$.

Cache Misses. The number of cache misses in a serial partition is the input size, so the total number of cache misses in this program is $t \cdot \frac{n}{t}$ from the serial partitioning of the P_is in parallel, plus $O(\sqrt{t \cdot n \log n})$ for when the algorithm performs a serial partition on the subarray. So the total number of cache misses is

$$n + O(\sqrt{t \cdot n \log n}).$$

To show that the second term in this sum becomes insignificant compared to n as n grows, we take

$$\lim_{n \to \infty} \frac{\sqrt{t \cdot n \log n}}{n} = \sqrt{\lim_{n \to \infty} \frac{t \cdot \log n}{n}}.$$

We can't choose $t > n$, and making $t = o(\frac{n}{\log n})$ is a reasonable constraint which guarantees that,

$$\lim_{n\to\infty} \frac{t}{\left(\frac{n}{\log n}\right)} = 0.$$

Thus, the algorithm only incurs $n+o(n) = n(1+o(1))$ cache misses. □

Spatial locality. The Strided Algorithm as described above is good because of the low number of cache misses, but it is not good in that there is no spatial locality in referencing elements of each P_i. A way to increase spatial cache friendliness of the algorithm is to redefine the parts P_is using blocks of adjacent elements instead of individual elements each separated by t indices for each P_i. Let b be the ***block size***.
To compute v_i, the index into A of the first successor in the partitioned P_i, we must now compute which block the index x_i into P_i falls in, denoted by w_i and where within the block x_i falls, denoted m_i as follows:

$$w_i = \left\lfloor \frac{x_i}{b} \right\rfloor,\ m_i = x_i \mod b.$$

Once we have w_i and m_i we compute v_i as follows:

$$v_i = t \cdot b \cdot w_i + (i-1)b + m_i.$$

But we don't need that much detail. More simply this is,

$$v_i = O(t \cdot b \cdot w_i) = O(x_i \cdot t).$$

Where the absorption into the big-O notation is reasonable because $m_i < b$, $i < t$ so the other terms get absorbed in the big-O notation, and we can also drop the floor and cancel the b. Note that this is the same expression we got for the conversion from x_i to v_i as last time (in big-O notation). So we once again get the same bound on $v_{max} - v_{min}$, which doesn't actually depend on b, that is, with high probability

$$v_{max} - v_{min} = O(\sqrt{t \cdot n \log n}).$$

Summary. It is not satisfactory that this algorithm does not have theoretical guarantees for arbitrary arrays. We use the ideas of this algorithm to create a new algorithm that has have theoretical guarantees on arbitrary inputs.

3 Key Algorithmic Ideas

We add randomization to the Strided Algorithm to make a new algorithm that has theoretical guarantees for arbitrary inputs while remaining cache efficient and in-place. To accomplish this the algorithm forms groups $G_0, G_1, \ldots, G_{g-1}$ which are each a collection of indices to parts P_j. We will refer to the set that is the union of all parts P_j where $j \in G_y$ as U_y. That is,

$$U_y = \bigcup_{j \in G_y} P_j.$$

Then the algorithm partitions each U_y in serial. Define μ_y to be the fraction of U_y that is composed of predecessors. It is important that $|\mu_y - \mu|$ is small for all y with high probability in n so that the algorithm will have a small subproblem to recurse on. We present 3 versions of the algorithm which are differentiated by their methods of forming groups.

Version 1. In this version of the algorithm, we form groups $G_0, G_1, \ldots, G_{g-1}$ of indices for certain P_js, which are defined in the same way that they were defined in the Strided algorithm, as follows:

- Permute the numbers $0, \ldots, t-1$ into g groups of size $O(\text{polylog}\, n)$.
- Sort each group G_y of indices in serial.
- For every group G_y perform a serial partition on U_y the union of all P_j where $j \in G_y$. To accomplish this, the algorithm must find the next or previous element in a group starting from the index of some other element in the group. This is accomplished by storing both the index into whichever part P_j the algorithm is currently on, and the index into the group G_y specifying which part P_j the algorithm is currently on. Then to, for example, find the next element the algorithm increments through the part P_j that it is currently in, until it reaches the end of the part, after which the algorithm increments its counter into G_y and starts at the beginning of the next part P_j.
- Partition $A[v_{min}], \ldots, A[v_{max} - 1]$ where $v_{min} = \min v_j, v_{max} = \max v_j$, and v_j is defined, as in the Strided algorithm, to be the index of the first successor in P_j. As before, this guarantees that all elements of A with indices $k < v_{min}$ are predecessors, and all elements with indices $k \geq v_{max}$ are successors.

This algorithm has the desirable property that, because we randomly group $O(\text{polylog}\, n)$ P_js together, the fraction μ_y of U_y that is composed of predecessors should cluster closely around the fraction μ of A that is composed of predecessors for all y. Also, we can make the algorithm have small span by performing some of the steps in parallel. The permutation step can be done in parallel. The algorithm can also perform the serial partition of all collections U_y in parallel. The unsatisfactory property of this version of the algorithm is the auxiliary memory that is required by the algorithm. The algorithm stores g groups of size $O(\text{polylog}\, n)$ where g is chosen to be $O(n/\text{polylog}\, n)$, so the algorithm uses $O(n/\text{polylog}\, n)$ extra space and is not in-place.

Version 2. In this version of the algorithm we again assign the indices of parts $P_0, \ldots, P_{t-1}$ to groups $G_0, \ldots, G_{g-1}$. However, in this version the groups are not independent of one another. Because of this dependence, the algorithm only needs $O(\text{polylog}\, n)$ auxiliary space to store all of the groups. The algorithm creates an array X of size $O(\text{polylog}\, n)$ and sets each $X[i]$ to be an integer chosen uniformly at random from $[0, g-1]$. The value $X[i]$ is used to determine a unique index $j \in [i \cdot g, (i+1) \cdot g - 1]$ for each group indicating which part P_j belongs to the group G_y from this set of indices. The index for the part from this set of indices that belongs to G_y is

$$i \cdot g + ((X[i] + y) \bmod g).$$

This method of assigning parts to groups assigns all P_js to exactly 1 group. Having independent random variables for each group is not necessary because we apply a union bound over all groups in proving that $|\mu - \mu_y|$ is small for all y with high probability in n, and the union bound still applies for random variables that are not independent. This version of the algorithm is an improvement over the previous algorithm because it does not use excessive auxiliary memory and it has small span, similarly to that of the previous algorithm. An undesirable property of this algorithm however is that it has some unnecessary overhead.

Version 3. In this version of the algorithm we eliminate the use of parts P_j in the algorithm.

The set U_y is a union of parts P_j which each contain multiple elements. The motivation for having P_j contain multiple elements was that in the Strided Algorithm the P_js were the only place where we combined elements into a set to partition in serial in the algorithm. We need to have collections of elements to make each collection have approximately the same fraction of predecessors as the entire array A. Now however, we are combining elements on 2 levels: we group elements into P_js, and we group P_js together into G_ys. We only need one source of grouping elements though, so we can set $|P_j| = 1$ and the collections $U_0, \ldots, U_{g-1}$ will still contain a collection of elements. Thus the groups can still be expected to have close to the same fraction μ_y of predecessors in them as the fraction μ in the entire array A. Setting $|P_j| = 1$ makes $P_j = \{A[j]\}$, demonstrating that P_j is no longer needed because it refers to a single element of A. When the cache block size is b instead of 1 it is convenient to still have parts P_js and say that each is composed of a single cache block. Thus, we now refer to P_j as a block, and it is defined to contain elements $A[b \cdot j], \ldots, A[b \cdot (j+1) - 1]$. Changing the role of the P_js is desirable because it adds simplicity to the algorithm and reduces overhead without sacrificing any desirable characteristics of the algorithm.

Algorithm Concept. We now present the ***Cache-Efficient Partial-Partition Algorithm*** which, like the parallel step of the Strided Algorithm, partitions subsets of the array in parallel such that partitioning the subsets and then a small subarray results in a fully partitioned array. However, unlike the parallel step of the Strided Algorithm, the Cache-Efficient Partial-Partition Algorithm has high probability guarantees on the size of the unpartitioned subarray for arbitrary inputs. The Strided Algorithm has guarantees on some inputs, for instance randomly ordered inputs, but the Cache-Efficient Partial-Partition Algorithm uses randomization in the algorithm to obviate the need for any specific type of input.

Interestingly, because the Cache-Efficient Partial-Partition Algorithm has guarantees on arbitrary inputs, the Cache-Efficient Partial-Partition Algorithm can be used to partition the subarray that it generates, unlike in the Strided Algorithm where recursing with the parallel step of the Strided Algorithm would not successfully partition the subarray. This is because the subarray generated in the parallel step of the Strided Algorithm is non-random in a problematic way: subsequences composed of every t-th element in the subarray would already be partitioned, so the parallel step of the Strided Algorithm would not change the array as it partitions these subsequences that are already partitioned, and thus could not finish the partitioning of the array.

The Cache-Efficient Partial-Partition Algorithm forms collections U_y, which are similar to the Strided Algorithm's P_js, and performs a serial partition of each U_y in parallel. We cannot explicitly store each U_y, because this would require $O(n)$ memory (or $O(n/b)$ if we use indices of blocks of the array) which would make the algorithm not in-place, and eliminate the algorithm's desirable cache behavior. However, we can represent all U_ys with very little space by making each U_y random, but not independent of other $U_{y'}$s.

Our construction of U_ys makes it so that the fraction of predecessors in each U_y will cluster closely around the fraction of predecessors in A. The elements in each U_y are spread out in A so that each section of a certain size in the array will contain the same number of elements from U_y.

Define v_y to be the index of the first successor in U_y–this is similar to the definition of the splitting position in the Strided Algorithm. Because of this uniformity in how the elements of U_y are spread out, and the guarantee that the fraction of successors in each U_y will cluster closely around the fraction of successors in A, the index v_y of the first successor in collection U_y will be close to all other indices $v_{y'}$. This means that after each U_y is partitioned, A will be partially partitioned, in the sense that $A[i]$ is a predecessor for all $i < \min_y v_y$, and $A[i]$ is a successor for all $i \geq \max_y v_y$. Furthermore, the size of the unpartitioned subarray $A[\min_y v_y], \ldots, A[\max_y v_y - 1]$ will be very small relative to n.

4 Cache-Efficient Parallel Partition for Arbitrary Inputs

Algorithm Description. We now describe the algorithm in more detail. Let b be the size of a cache line. Let X be an array such that $X[i] \in \{0, 1, \ldots, g-1\}$ determines which chunk of size b from the array belongs to any group G_y from section i of size $g \cdot b$ in the array. Let $s = |X|$. Then g, the number of groups G_y the algorithm will form, is determined by $g = \frac{n}{b \cdot s}$ [1]. The algorithm performs the following procedure:

- Logically partition the array A into blocks P_j each of b adjacent elements, i.e.

  $$P_j = \{A[b \cdot j], A[b \cdot j + 1], \ldots, A[b \cdot (j+1) - 1]\}.$$

- Form the array X, where each element $X[i]$ is an integer chosen randomly at uniform from $[0, g-1]$.

- The values in X determine g groups G_y each of s indices to P_js. The index for block i of group G_y is determined by

  $$G_y[i] = (X[i] + y) \bmod g + i \cdot g.$$

  Note that we do not need to store the indices of the members of each group even though the groups are random, because the constituent blocks of a group are determined by the group index and the array X. This means that as long as $|X|$ is made small, which it will be, we do not use significant extra space in creating this array.

- Define U_y to be the union of all blocks that belong to group G_y. That is,

  $$U_y = \bigcup_{j \in G_y} P_j.$$

  The algorithm performs a serial partition on each U_y in parallel.

- Define v_y to be the index of the first successor in U_y, and define $v_{\min} = \min_y v_y$, $v_{\max} = \max_y v_y$ [2]. Note that the array is partially partitioned, i.e. $A[i]$ is a predecessor for all $i < v_{\min}$, and $A[i]$ is a successor for all $i \geq v_{\max}$.

[1] If $n \not\equiv 0 \mod b \cdot s$, then we define g as $\left\lfloor \frac{n}{b \cdot s} \right\rfloor$. This means that $g \cdot b \cdot s < n$. We handle the array of size $n - g \cdot b \cdot s$ separately from the array of size $g \cdot b \cdot s$ that is the focus of our analysis. Thus for simplicity in the analysis we say that $n = g \cdot b \cdot s$. It is straightforward to deal with the $n - g \cdot b \cdot s$ elements. This is done by swapping the extra elements with the first $n - g \cdot b \cdot s$ successors, and then increasing the size of the subproblem to include these new elements. Because $n - g \cdot b \cdot s$ is very small, this step is not relevant to the analysis, but this is an important detail to handle in order to make the algorithm work.

[2] When calculating $v_{\max}$ and $v_{\min}$ we do not want to store the v_ys because this would require $O(g)$ memory. In order to avoid storing the v_y, we compute $v_{\max}$ and $v_{\min}$ while partitioning each U_y in parallel. To facilitate this, we implement a parallel for loop with a recursive divide-and-conquer strategy. We have a function that takes in

Algorithm Analysis. We now prove the following general proposition about the Cache-Efficient Partial-Partition Algorithm:

Proposition 1. *Let A be an array of size n; Let μ be the fraction of predecessors in A; Let b, the size of a cache line, be $O(\text{polylog}\, n)$; Let $\epsilon \in (0,1)$ be a failure probability; Let $\delta \in (0,1)$; Let $s \in \Theta\left(\frac{\log(n/\epsilon)}{\delta^2}\right)$ satisfying $s > \frac{\log(2n)-\log(b\epsilon)}{2\delta^2}$.*

The Cache-Efficient Partial-Partition Algorithm, when executed on array A: achieves work $O(n)$; achieves span $O(b \cdot s)$; incurs $\frac{s+n}{b} + O(1)$ cache misses (assuming the array X is pinned to cache); and leaves an unpartitioned subarray $A[v_{min}], \ldots, A[v_{max} - 1]$ which, with probability $1 - \epsilon$, has size

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

Proof. The algorithm accesses each element of the array once, resulting in work $O(n)$. Recall that by the Resource Augmentation Theorem we can assume small arrays and values that are repeatedly used to be pinned to cache. The contribution to the algorithm's cache misses are then: n/b from accessing each block of A, s/b for instantiating the array X, and $O(1)$ for performing various other tasks that require a constant number of cache misses. This results in the total number of cache misses being

$$\frac{n+s}{b} + O(1).$$

The algorithm has span

$$O(n/g) = O(b \cdot s)$$

because of the g serial partitions of groups that it performs in parallel on arrays of size $b \cdot s$. To show that with probability $1 - \epsilon$ the size of the unpartitioned subarray is bounded above by $4n\delta$, we consider the fraction of elements in each collection U_y that are predecessors, which we denote by μ_y. Note that each μ_y is the average of s independent random variables, where random variable i is the fraction of $P_{G_y[i]}$ that is composed of predecessors, a random variable in $[0,1]$. The groups are constructed in a symmetric way such that for each group, the probability of the group getting any specific block is the same as the probability of any other group getting that block. Because of this symmetry, $\mathbb{E}[\mu_y]$–a characteristic on the group–is the same for all y. Furthermore, $\mathbb{E}[\mu_y] = \mu$ for all $y \in \{0, \ldots, g-1\}$ because the average of the fraction of predecessors in each group is the fraction of predecessors in the entire array. Thus we can apply Hoeffding's inequality (i.e. A Chernoff Bound for a random variable on $[0,1]$ rather than on $\{0,1\}$) to each μ_y to show that they are concentrated around their shared expected value μ, i.e.

$$\Pr[|\mu_y - \mu| \geq \delta] < 2\exp(-2s\delta^2).$$

Substituting in s which was chosen $s > \frac{\log(2n)-\log(b\epsilon)}{2\delta^2}$, we find that for all $y \in \{0, \ldots, g-1\}$,

$$\Pr[|\mu_y - \mu| \geq \delta] < 2\exp\left(-2\frac{\log(2n/(b\epsilon))}{2\delta^2}\delta^2\right) = \frac{\epsilon}{n/b} < \frac{\epsilon}{g}.$$

We use this bound on the probability of any individual group U_y failing to meet the condition $|\mu - \mu_y| < \delta$ to bound the probability that at least one of the groups $U_0, \ldots, U_{g-1}$ fails to meet the

a subset of the U_y, partitions these U_y in parallel, and then returns the minimum and maximum v_y among the U_y that the function partitioned. This function operates by divide-and-conquer, i.e. it spawns processes that in parallel recursively apply this function to 2 disjoint, approximately equal sized, subsets of the subset of U_ys that the function was given, and then the function computes its $v_{\min}, v_{\max}$ as the min of the $v_{\min}$s reported by the subprocesses and the max of the $v_{\max}$s reported by the subprocesses.

condition. Note that the probability of at least one group failing is:

$$\Pr\Big[\bigvee_{y=0}^{g-1} |\mu_y - \mu| \geq \delta\Big].$$

This is bounded by

$$\Pr\Big[\bigvee_{y=0}^{g-1} |\mu_y - \mu| \geq \delta\Big] \leq \sum_{y=0}^{g-1} \Pr[|\mu_y - \mu| \geq \delta] < \epsilon.$$

Thus the event occurs with probability bounded above by ϵ, the specified failure probability. To complete the proof we will show that the occurrence of the event that all y simultaneously satisfy $|\mu - \mu_y| < \delta$ implies that the size of the unpartitioned subarray is bounded above by $4n\delta$.

Intuitively, because the elements in each group are distributed evenly throughout the array, $\mu_y \cdot n$ approximately determines v_y, so we can bound $v_y - \mu \cdot n$ for all y using our bound on $|\mu - \mu_y|$. The index for the block of group G_y that contains v_y is

$$G_y[\mu_y \cdot s] = (X[\mu_y \cdot s] + y) \bmod g + \mu_y \cdot s \cdot g.$$

Because the entries of X are random, we chose to bound $G_y[\mu_y \cdot s]$ without referencing X, only referencing parameters and variables of the problem. This bound is,

$$\mu_y \cdot s \cdot g \leq G_y[\mu_y \cdot s] \leq \mu_y \cdot s \cdot g + g - 1.$$

Then, because block $P_{G_y[i]} = \{A[b \cdot i], \ldots, A[b \cdot i + b - 1]\}$, we can bound v_y as

$$\mu_y \cdot s \cdot g \cdot b \leq v_y \leq (\mu_y \cdot s \cdot g + g - 1) \cdot b + b - 1.$$

Note that $n = s \cdot g \cdot b$. Rearranging, we have

$$0 \leq v_y - \mu_y \cdot n < g \cdot b.$$

We desire to bound $v_y - \mu \cdot n$, which can be done using our bound on $|\mu_y - \mu|$ as follows:

$$-\delta \cdot n \leq v_y - \mu \cdot n < \delta \cdot n + g \cdot b.$$

Thus, the concentration of all v_ys implies that with probability $1 - \epsilon$,

$$v_{\max} - v_{\min} < n(2\delta) + g \cdot b.$$

We express this in terms of the algorithm's parameters, using the fact that $g \cdot b = n/s$,

$$v_{\max} - v_{\min} < n\left(2\delta + \frac{2\delta^2}{\log(2n) - \log(b\epsilon)}\right) < 4n \cdot \delta.$$

□

We can use this result to analyze two Parallel full Partition algorithms that use the Cache-Efficient Partial-Partition Algorithm.

1. The ***Cache-Efficient Full-Partition Algorithm*** algorithm achieves the best theoretical span and cache efficiency for appropriate parameter choice.

2. The ***Grouped Partition Algorithm*** has slightly higher span, but has a very small number of cache misses, and is exceptionally simple to implement.

Both algorithms start by using the Cache-Efficient Partial-Partition algorithm with parameters $\epsilon = 1/n^c$ for c of our choice (i.e. with high probability in n) and parameter δ which can be toggled to achieve a trade off between span and cache misses. The algorithms then partition the unpartitioned subarray with their own methods.

The Cache-Efficient Full-Partition Algorithm uses the algorithm given by Theorem **??** which

has span $O(\log\log n \log n)$ to partition the subarray. This approach is motivated by the observation that once the problem size has been reduced cache behavior is less important, so we can switch to the algorithm that optimizes span rather than cache misses to solve the subproblem.

The Grouped Partition Algorithm partitions the subarray by repeatedly applying the Cache-Efficient Partial-Partition algorithm. The recursive Cache-Efficient Partial-Partition algorithms use the same parameter ϵ as the top-level, in order to still guarantee success with high probability in n, and use $\delta = 1/8$ in order to at least halve the size of the problem size at each iteration.

We now establish the following theorem about the Cache-Efficient Full-Partition Algorithm algorithm in general and a corollary about an interesting setting of its parameters.

Theorem 2. *The Cache-Efficient Full-Partition Algorithm algorithm using parameter $\delta \in (0,1)$ satisfying $\delta \geq \Omega(n^{-1/4})$: has work $O(n)$, and with high probability in n, achieves span*

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

and incurs fewer than

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

cache misses.

An interesting corollary of the above theorem concerns what happens when we chose δ to optimize span. This can be done with an extreme setting of δ. This is interesting because it shows that it is possible to achieve low span along with a small number of cache misses. [Corollary of Theorem 3] The Cache-Efficient Full-Partition Algorithm algorithm using $\delta = \Theta\big(\sqrt{b/\log\log n}\big)$, where b is chosen to satisfy $\sqrt{b/\log\log n} = o(1)$: 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.

Proof of Theorem 3. Using Proposition 1, we find the relevant statistics for the step of this algorithm where we apply the Cache-Efficient Partial-Partition Algorithm. Note that by our choice of ϵ, $s = O\left(\frac{\log n}{\delta^2}\right)$. The top layer has work $O(n)$, span $O\Big(\frac{b\log n}{\delta^2}\Big)$, and incurs fewer than

$$\frac{n}{b} + O\left(\frac{\log n}{b\delta^2}\right) + O(1)$$

cache misses. The subproblem has size less than $4n\delta$, so using the algorithm described by Theorem **??**, we achieve span

$$O(\log n\delta \log\log n\delta) = O(\log n \log\log n)$$

and work $O(n\delta) \leq O(n)$ in solving the subproblem. Thus the total work is $O(n)$, and the total span is

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

In solving the subproblem the number of cache misses that we incur is linearly dependent on the size of the subarray, so the total number of cache misses is

$$\frac{n}{b} + O\left(\frac{\log n}{b\delta^2} + \frac{n\delta}{b}\right) + O(1).$$

Note that $\delta > \Omega(n^{-1/4})$ implies

$$O\left(\frac{\log n}{\delta^2}\right) < O\left(\frac{n^{1/8}}{\delta^2}\right) < O\left(n^{1/2+1/8}\right) < O\left(n^{3/4}\right) < O(n\delta)$$

so the number of cache misses simplifies to

$$(n + O(n\delta))/b + O(1).$$

Recall that $b \leq O(\text{polylog}\, n)$, so $n\delta/b \geq \Omega(1)$, which makes the total number of cache misses

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

□

Proof of Corollary 4. We use $\delta = \sqrt{b/\log\log n}$ in the result proved in Theorem 3.
First note that the assumptions of Theorem 3 are satisfied because

$$O(\sqrt{b/\log\log n}) > \Omega(n^{-1/4}).$$

The algorithm achieves work $O(n)$. With high probability in n the algorithm achieves span

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

With high probability in n the algorithm incurs fewer than

$$(n + O(n\delta))/b = (n + O(n\sqrt{b/\log\log n}))/b$$

cache misses. By assumption $\sqrt{b/\log\log n} = o(1)$, so this reduces to

$$(n + o(n))/b$$

cache misses. □

Theorem 3. *The Grouped Partition Algorithm using parameter $\delta_0 \in (0,1)$ satisfying $\delta_0 \geq \Omega(n^{-1/4})$: achieves work $O(n)$, and with high probability in n, attains span*

$$O\left(b\left(\log^2 n + \frac{\log n}{\delta_0^2}\right)\right)$$

and incurs $(n + O(n\delta_0))/b$ cache misses.

[Corollary of Theorem 4] The Grouped Partition Algorithm using parameter $\delta_0 = 1/\sqrt{\log n}$: achieves work $O(n)$, and with high probability in n, attains span $O(b\log^2 n)$ and incurs $(n + o(n))/b$ cache misses.

Proof of Theorem 4. By Theorem 1, the top level of the algorithm has work $O(n)$, span $O\left(b\frac{\log n}{\delta_0^2}\right)$, and incurs

$$\frac{s+n}{b} + O(1)$$

cache misses. On the top layer of recursion the algorithm reduces the problem size by at least a factor of 4δ. On lower layers of recursion with high probability in n the algorithm reduces the problem size by at least a factor of $1/2$ at each iteration by our choice of $\delta' = 1/8$ as the parameter for lower levels of the Cache-Efficient Partial-Partition algorithm. The size of the problem on lower levels thus is bounded above by

$$2^2 n\delta_0, 2^1 n\delta_0, 2^0 n\delta_0, 2^{-1} n\delta_0, \ldots$$

This sequence of sizes terminates, but the sum of the terms in the infinite sequence with terms of the form $\{2^{2-i}n\delta\}_{i=0}^{\infty}$ is an upper bound for the sum of the sizes of the lower levels of recursion. The bound is

$$\sum_{i=0}^{\infty} 2^{2-i} n\delta_0 = 8n\delta_0.$$

This means that the total work of the algorithm is bounded above by $O(8n\delta_0)+O(n)$, so the total work is $O(n)$.

In order to compute the span of the algorithm, we apply Proposition 1 to the subarray, which has size bounded above by $2^{2-i}n\delta_0$, and has parameter $\delta' = 1/8$. Thus, level i contributes

$$O(b{\cdot}s) = O\Big(b\frac{\log(2^{-i}n\delta_0/b)}{(\delta')^2}\Big) = O(b(\log(n\delta_0/b)-i))$$

to the span. Now note that because the algorithm at least cuts the problem size in half each time we will reduce the problem size to the base case in at most $\log(4n\delta_0)$ levels. Thus, to get the total span on the lower levels of recursion, we must add together $O(\log(n\delta_0))$ terms of the form $O(b(\log(n\delta_0/b) - i))$. This makes the total span on lower levels of recursion $O(b\log^2(n\delta_0)) = O(b\log^2 n)$, and the total span for the algorithm

$$O\left(b\left(\log^2 n + \frac{\log n}{\delta_0^2}\right)\right).$$

To compute the total number of cache misses of the algorithm, we add together $(n+s)/b+O(1)$ for the top level, and then at most

$$\sum_{0\le i<\log 4n\delta} \frac{1}{b}\cdot O\left(2^{2-i}n\delta_0 + \frac{\log 2^{2-i}n\delta_0/b}{(1/8)^2}\right)$$

for lower levels. The first term is the sum of a geometric series and thus is $O(n\delta_0)/b$. We can bound the second term by $O(\log^2 n)/b$ by taking a larger term at each index in the sum. Then the total number of cache misses for the algorithm is

$$\frac{1}{b}\left(n + \frac{\log n}{\delta_0^2}\right)+O(1)+O(\log^2 n)/b+O(n\delta_0)/b = (n+O(n\delta_0))/b$$

□

Proof of Corollary 4. We use $\delta_0 = 1/\sqrt{\log n}$ in the result proved in Theorem 4. First note that the assumptions of Theorem 4 are satisfied because

$$\delta_0 = 1/\sqrt{\log n} \ge \Omega(n^{-1/4}).$$

The algorithm has work $O(n)$. With high probability in n the algorithm has span

$$O\left(b\left(\log^2 n + \frac{\log n}{\delta_0^2}\right)\right) = O(\log^2 n),$$

and incurs

$$(n+O(n\delta_0))/b = (n+O(n/\sqrt{\log n}))/b = (n+o(n))/b$$

cache misses. □

5 Analysis of Grouped Partition

In this section we give a more detailed description of our final algorithm, which we call the ***Grouped Partition Algorithm*** and prove some key results regarding its performance and number of caches misses.

Algorithm Overview. We logically divide the array $A = A[0], A[1], \ldots, A[n-1]$ into blocks P_j each of b adjacent elements, where b is the ***block size***. That is,

$$P_j = \{A[b\cdot j], A[b\cdot j+1], \ldots, A[b\cdot(j+1)-1]\}.$$

Note that the P_js are defined in a different way then in the Strided algorithm when P_j contained elements spaced throughout the array. This is equivalent to setting $|P_j| = 1$ for the P_js in the Strided Algorithm and then making P_j a collection of cache blocks. Define **pred(j)** to be the number of predecessors in P_j. There are n/b blocks P_j.

The algorithm will form g groups of these P_js by the following procedure. The algorithm generates a random array $X = X[0], X[1], \ldots, X[s-1]$ where each element in X is an integer chosen randomly at uniform from $[0, g-1]$. The values in X determine groups $G_0, G_1, \ldots, G_{g-1}$ of P_js. The first group is

$$G_0 = \{X[0], X[1] + g, X[2] + 2 \cdot g, \\ \ldots, X[s-1] + (s-1) \cdot g\}.$$

This means that group G_0 is a collection of indices for specific parts P_j, indicating that these parts belong to group G_0. Similarly, group G_y is defined as

$$G_y = \{(X[0] + y) \bmod g, (X[1] + y) \bmod g + g, \\ \ldots, (X[s-1] + y) \bmod g + (s-1) \cdot g\}.$$

Intuitively this means that, for group G_y, on each chunk of size g of the array, take $X[j]$ and add the group's index i (wrapping around if there is overflow beyond the number of groups) to get to the index of the P_j that belongs to group G_y from this chunk of the array. Note that we do not need to store the indices of each P_j that belongs to a group because X and the group index is enough information to determine which P_js belong to a group. We call our algorithm in-place because s (recall $s = |X|$) is made $O(\text{polylog}(n))$.

Define U_y to be the union of all parts that belong to group G_y. That is,

$$U_y = \bigcup_{j \in G_y} P_j.$$

Define μ_y to be the number of predecessors in U_y divided by the number of elements in U_y. This is analogous to the definition of μ: the number of predecessors in A divided by n.

Once the algorithm has generated X, it performs a serial partition on each collection U_y in parallel. The algorithm performs the partitions in parallel by splitting the tasks using a recursive method of calling the function repeatedly. While performing the serial partitions of each U_y, the algorithm computes the indices v_{max}, v_{min} where $v_{max} = \max v_y, v_{min} = \min v_y$ and v_y is the index in A of the first successor in U_y. This is roughly outlined in Figure 1.

Then, because for all y

$$v_{min} \leq v_y \leq v_{max},$$

all elements of A with index less than v_{min} are predecessors and all elements of A with index greater or equal to v_{max} are successors. Thus, by recursing on the subarray $A[v_{min}], \ldots, A[v_{max} - 1]$, we complete the partitioning of the array. We recursively apply the Grouped Partition algorithm to the subproblem. The base case for the recursion is that when the algorithm can no longer make a substantial number of groups, in which case it partitions the input array in serial.

See Figure 2 for a pseudocode implementation of the algorithm.

Proof of Lemma **??**.
First we use Hoeffding's inequality (i.e. Chernoff Bounds for random variables on $[0, 1]$ rather than on $\{0, 1\}$) to bound the probability that $|\mu - \mu_y| \geq \delta$. We can express μ_y as the sum of the number of predecessors in all the blocks of the group G_y divided by $|G_y| \cdot |P_j|$. Note that G_y is composed of s blocks of b elements each, so $|G_y| \cdot |P_j| = s \cdot b$. Thus we can express μ_y as

Figure 1: Recursive Spawning Implementation of Parallel For Loop

Input: first group index, number of groups to process
Output: v_{min}, v_{max} **for** $U_y, \ldots, U_{y+n-1}$

```
1: procedure PARTITIONGROUPS(y, n)
2:     if n = 1 then
3:         perform serial partition of U_y
4:         return v_y, v_y
5:     else
6:         leftvs ← PARTITIONGROUPS(y, n/2)
7:         rightvs ← PARTITIONGROUPS(y + n/2, n/2 + n&2)
8:         return min(leftvs.v_min, rightvs.v_min), max(leftvs.v_max, rightvs.v_max)
9:     end if
10: end procedure
```

$$\begin{aligned}\mu_y &= \frac{1}{|G_y| \cdot |P_j|} \sum_{j \in G_y} \text{pred}(j) \\ &= \frac{1}{b \cdot s} \sum_{i=0}^{s-1} \text{pred}(G_y[i]).\end{aligned} \tag{1}$$

We are interested in calculating that the expected value of μ_y, or equivalently calculating the average fraction of predecessors in group G_y over all allowable assignments of blocks to the group. This is,

$$\mathbb{E}(\mu_y) = \mathbb{E}\left(\frac{1}{s \cdot b} \sum_{i=0}^{s-1} \text{pred}(G_y[i])\right).$$

By linearity of expectation this is

$$\mathbb{E}(\mu_y) = \frac{1}{s \cdot b} \sum_{i=0}^{s-1} \mathbb{E}(\text{pred}(G_y[i])).$$

Because $G_y[i]$ is an integer random variable which assumes an index uniformly at random from $[g \cdot i, g \cdot (i+1) - 1]$ we can expand the expectation to:

$$\mathbb{E}(\mu_y) = \frac{1}{s \cdot b} \sum_{i=0}^{s-1} \sum_{k=0}^{g-1} \frac{1}{g} \text{pred}(k + i \cdot g).$$

But the argument to pred assumes all possible values from $[0, s \cdot g - 1]$, so we can rewrite the sum as

$$\mathbb{E}(\mu_y) = \frac{1}{b \cdot s \cdot g} \sum_{j=0}^{s \cdot g - 1} \text{pred}(j) = \frac{n \cdot \mu}{b \cdot s \cdot g} = \mu.$$

6 Systems Performance

Handling Non-divisible Array Size. In the analysis above we have assumed that n is divisible by our parameters g, b, s. However, this simplifying assumption is incredibly restrictive and unnecessary. In our algorithm b and s parameters, and then g is determined as

$$g = \left\lfloor \frac{n}{b \cdot s} \right\rfloor.$$

If $n \bmod (b \cdot s) \not\equiv 0$, then $g \cdot b \cdot s < n$. To deal with this problem, we first

Figure 2: Parallel Partition

```
1: if g < 2 then
2:     serialPartition A
3: else
4:     for i ∈ {0, 1, ..., s − 1} do
5:         X[i] ← a random integer from [0, g − 1]
6:     end for
7:     – We implement this parallel for-loop with a sequence of recursive spawns, and which
       facilitates computing v_min, v_max without storing v_ys
8:     for all y ∈ {0, 1, ..., g − 1} in parallel do
9:         – Now we perform a serial partition on U_y
10:        – Initialize ALowIdx to be the index of the first element in U_y
11:        ALowIdx ← ((X[0] + y) mod g) · b
12:        – Initialize AHighIdx to be the index of the last element in U_y
13:        AHighIdx ← n − g · b + ((X[s − 1] + y) mod g) · b + b − 1
14:        while ALowIdx < AHighIdx do
15:            while A[ALowIdx] ≤ pivotValue do
16:                ALowIdx ← ALowIdx+1
17:                if ALowIdx on block boundary then
18:                    – We perform a block increment
19:                    i ← # of block increments so far (including this one)
20:                    – Increase ALowIdx to start of block i of G_y
21:                    ALowIdx ← ((X[i] + y) mod g) · b + i · b · g
22:                end if
23:            end while
24:            while A[high] > pivotValue do
25:                AHighIdx ← AHighIdx−1
26:                if AHighIdx on block boundary then
27:                    – We perform a block decrement
28:                    i ← # of block decrements so far (including this one)
29:                    – Decrease AHighIdx to end of block s − 1 − i of G_y
30:                    AHighIdx ← ((X[s − 1 − i] + y) mod g) · b + i · b · g + b − 1
31:                end if
32:            end while
33:            Swap A[ALowIdx] and A[AHighIdx]
34:        end while
35:    end for
36:    Recurse on A[v_min], ..., A[v_max − 1]
37: end if
```

apply our partial partition algorithm to $A[0], \ldots, A[g \cdot s \cdot b - 1]$. After doing this there is a subarray $A[v_{min}], \ldots, A[v_{max} - 1]$ left to partition, and there is also the part of the array $A[g \cdot s \cdot b], \ldots, A[n-1]$ that we have ignored. The algorithm deals with the tail of the array by either swapping the tail with a portion of the array of equal size which is the earliest successors in the array, or, in the rare case that there are more elements in the tail than successors– this may realistically happen on a very low layer of recursion when solving the partition problem with repeated applications of our algorithm, or if the pivot value is chosen adversarially such that there are very few successors–then the entire section of the array which is composed of successors is swapped with the last portion of the tail which is of the same length as the length of the section of the array composed of successors and appropriately incrementing v_{max}. This step adds very little to the size of the recursive subproblem because

$$n - b \cdot s \left\lfloor \frac{n}{b \cdot s} \right\rfloor < b \cdot s.$$

This does not affect the algorithm's work, span, or cache-efficiency.

Optimizations. The Grouped Partition algorithm must perform a serial partition on each group G_y. The serial partition is complicated by the fact that the groups do not contain a single contiguous portion of the array, but rather they contain s blocks P_j of size b each. In the serial partition the algorithm must repeatedly find the next or previous element in the group. Although the algorithm could utilize modular arithmetic and floor division of a counter to find the index into the array A, division and modulo are very expensive operations to perform, so we use the better approach of checking proceeding within a block while checking if we have hit the boundary of the block, and then at that point jumping to the next block. It is possible with this method to only use addition and subtraction and bit shifting by $\log b$, which is numerically equivalent to multiplication by b but faster and possible because we choose b to be a power of 2. This makes the determination of next and previous elements in the serial partition step much faster. Our algorithm spends most of its time on the serial partition step at the top level of recursion. This makes it so that the algorithm gets high parallelism because by increasing the number of groups that the algorithm can process in parallel the amount of time that the algorithm must spend on the time consuming step of serial partitioning is reduced. The overhead introduced in the serial partition step of the algorithm slows it down initially, but is overcome when the number of processors increases.

Performance. This new algorithm is very cache efficient, and operates very close to the memory bandwidth bound. Our implementation of the algorithm is available on github: https://github.com/awestover/Parallel-Partition.