Cache Efficient Parallel Partition

Alek Westover
June 26, 2019

Abstract

Quicksort is a very important problem to which a fast solution is desirable. By utilizing parallel processing it is possible to perform quicksort much faster than a serial implementation could. Quicksort consists of partitioning an array according to some pivot value such that the elements with value at most the pivot value occur before the elements with value greater than the pivot value, and then recursively sorting each of these parts of the array. Thus, parallel partition is the important algorithm to make fast in order to make quicksort fast. Parallel partition is also an important problem in its own right, which further motivates exploration of parallel algorithms for performing a partition. In this paper we present an algorithm for parallel partition that has low span, is fast in practice, incurs only $n(1 + o(1))$ cache misses, is in place, and has theoretical guarantees for arbitrary inputs.

1 Introduction

Serial Partition Problem. We describe the serial partition problem and the solution which is used extensively in parallel partition algorithms. The serial partition problem takes in an array, and then reorders the array according to a decider function. The serial partition algorithm, which runs fully in-place, has the end result of the array being partitioned such that all elements labelled predecessors by the decider function occur before the elements labelled successors by the decider function. In the case of the decider function "$A[i] \leq pivotValue$", 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, upon recursively sorting the predecessors and the successors the entire sorting problem is solved. 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 $O(n)$.

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
```

Further Preliminaries. Here 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 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 in conjunction 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 which would be quite adversarial.

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. Brent's theorem states that

$$T_p = \Theta\left(T_\infty + \frac{T_1}{p}\right)$$

which shows why optimizing the algorithms span and work 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, 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 $1 \cdot n$ 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 actual speed is 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 Algorithm With Desirable Guarantees On Random Inputs

There is an algorithm for performing 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 random inputs. We now present a proof of the following theorem regarding the algorithm.

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.*

Definitions. 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 $n \bmod t \not\equiv 0$ 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. We proceed to 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 deviation 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 1.

Proof of Theorem 1.

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 fully satisfactory that this algorithm only has these theoretical guarantees for the restrictive case of random inputs. 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.

4 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.

Now we prove the following lemma to simultaneously bound $|\mu - \mu_y|$ for all y.

Lemma 2. *Given n, μ, For our choice of b, δ, if $s = \Theta(\frac{\log n}{\delta^2})$ and $g = \frac{n}{b \cdot s}$ then the Grouped Partition Algorithm forms g such that, with high probability in n, for all y, $|\mu - \mu_y| < \delta$.*

Proof of Lemma 2.
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

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

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:

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

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
```

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

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

We are looking for the probability that

$$\left| \mu_y - \mu \right| \geq \delta.$$

Note that μ_y is the average of the s independent random variables

$$\frac{1}{b} \text{pred}(G_y[i]) \in [0, 1]$$

by Equation (1), so we can apply Hoeffding's inequality. Let M_y be the event that $|\mu_y - \mu| \geq \delta$. Hoeffding's inequality gives us that the probability of μ_y deviating from its expected value, which was calculated to be μ, by more than δ is

$$\Pr(M_y) \leq \exp(-2s\delta^2).$$

We have found this bound for each group G_y by themselves, but we need the bound to apply to all groups at once. To extend the bound to apply to all groups at once we use the idea of the union bound, which says that if a collection of events each individually do not happen with high probability, then the union of all of these events also does not occur with high probability. We wish to make

$$\Pr\left(\bigcup_{y=0}^{g-1} M_y \right) < \epsilon.$$

Using the fact that

$$\Pr\left(\bigcup_{y=0}^{g-1} M_y \right) \leq \sum_{y \in [0, g-1]} \Pr(M_y),$$

we see that by making each

$$\Pr(M_y) < \epsilon / g,$$

we can make the probability of any of the events $M_0, \ldots, M_{g-1}$ occurring less than ϵ.

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:     for all y ∈ {0, 1, ..., g − 1} in parallel do
8:         – Now we perform a serial partition on U_y
9:         – Initialize ALowIdx to be the index of the first element in U_y
10:        ALowIdx ← ((X[0] + y) mod g) · b
11:        – Initialize AHighIdx to be the index of the last element in U_y
12:        AHighIdx ← n − g · b + ((X[s − 1] + y) mod g) · b + b − 1
13:        while ALowIdx < AHighIdx do
14:            while A[ALowIdx] ≤ pivotValue do
15:                ALowIdx ← ALowIdx+1
16:                if ALowIdx on block boundary then
17:                    – We perform a block increment
18:                    i ← # of block increments so far (including this one)
19:                    – Increase ALowIdx to start of block i of G_y
20:                    ALowIdx ← ((X[i] + y) mod g) · b + i · b · g
21:                end if
22:            end while
23:            while A[high] > pivotValue do
24:                AHighIdx ← AHighIdx−1
25:                if AHighIdx on block boundary then
26:                    – We perform a block decrement
27:                    i ← # of block decrements so far (including this one)
28:                    – Decrease AHighIdx to end of block s − 1 − i of G_y
29:                    AHighIdx ← ((X[s − 1 − i] + y) mod g) · b + i · b · g + b − 1
30:                end if
31:            end while
32:            Swap A[ALowIdx] and A[AHighIdx]
33:        end while
34:    end for
35:    Recurse on A[v_min], ..., A[v_max − 1]
36: end if
```

We established an upper bound for $\Pr(M_y)$ above with Hoeffding's inequality, so to ensure that $\Pr(M_y)$ is less than ϵ/g it suffices to make the upper bound less than ϵ/g. That is,

$$\Pr(M_y) < \exp(-2s\delta^2) \leq \frac{\epsilon}{g}.$$

Solving for the necessary size of s we find that

$$s \geq \frac{\ln(g/\epsilon)}{2\delta^2}.$$

If we want all events M_y to not occur with high probability in n, then we must set $\epsilon = 1/n^c$. Thus,

$$s \geq \frac{\ln(g \cdot n^c)}{2\delta^2}.$$

Larger values of s hurt our span however, so we will chose δ to be just big enough to ensure that M_y all happen with high probability in n. Thus we chose

$$s = \Theta\left(\frac{\log n}{\delta^2}\right)$$

to ensure that with high probability in n $|\mu - \mu_y| < \delta$ for all y. □

Grouped Partition Subproblem Analysis. Once the Grouped Partition algorithm has made a single pass over the array there is a subproblem left to solve. Lemma 2 can be used to find out how much smaller we can expect the subarray to be relative to the original array, from which we can derive the span and cache efficiency of the Grouped Partition Algorithm. There are two options for solving the recursive subproblem, and we calculate the span of the algorithm given each approach.

Proposition 1. *By first applying the Grouped Partition algorithm to the top level of recursion for the partition problem, and then solving the remaining subproblem problem with an already existing algorithm for parallel partition, the algorithm achieves span*

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

and makes $(1+\delta)\cdot n$ *cache misses.*

Proof. From previous work we know that it is possible to perform a parallel partition of an array of size $\delta \cdot n$ with span $O(\log(n\delta)\log\log(n\delta))$. With this method our algorithm consists of first sorting each of the g groups in serial, which will take time $O(s\cdot b)$ for each group because serial partition is has linear run time in the input size and $|U_y| = |G_y|\cdot|P_j| = s\cdot b$. After this the algorithm will have, with high probability in n, a subarray of size $n\cdot\delta$ to partition, which it does in parallel with the other algorithm. In computing the span of our algorithm, all of the groups can be partitioned in serial at the same time. This makes the span of our algorithm, with this choice of a recursive technique,

$$T_\infty(n) = O(s\cdot b + \log(n\delta)\log\log(n\delta)).$$

By Lemma 2 we choose $s = \Theta(\frac{\log n}{\delta^2})$, so the span becomes

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

Note that $\log(n\delta) = \log(n) - \log(\frac{1}{\delta}) < \log n$ so we can simplify the span to

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

Now we compute the number of cache misses incurred. On the top level, the algorithm incurs n cache misses for the serial partitions of all the groups. When recursing on an array of size $\delta \cdot n$ the algorithm incurs $\delta \cdot n$ cache misses. Thus, in total the algorithm incurs $(\delta + 1) \cdot n$ cache misses. □

Recursing With the Same Algorithm. We could instead recurse with the Grouped Partition algorithm. Note that with Francis and Pannan's algorithm which only performs well on random inputs we could not do this because their algorithm works poorly on the subarray generated, but our algorithm's performance is guaranteed for arbitrary inputs so we can recurse with our algorithm. There are two ways we can do this. We could applying special treatment to the top level of recursion because it is by far the biggest problem, or we could treat all levels the same.

Proposition 2. *If we use the same allowable deviation δ for $|\mu_y - \mu|$ at all levels of recursion then the algorithm has span*

$$T_\infty(n) = \Theta(b \log^2 n),$$

and incurs approximately $2n$ cache misses.

Proof. If we choose some constant δ, then by Lemma 2 choosing $s = \Theta(\log n)$ makes $|\mu - \mu_y| < \delta$ for all y with high probability in n. Then,

$$\mu - \delta < \mu_y < \mu + \delta$$

for all y, so

$$\max \mu_y - \min(\mu_y) < 2\delta.$$

We can compute v_y from μ_y, because $\lfloor \mu_y \cdot s \rfloor$ gives the number of parts P_j where $j \in G_y$ that are completely filled with predecessors once the collection U_y is partitioned. There are s portions of size $g \cdot b$ in the array, so to find the index of the start of the portion of index $\lfloor \mu_y \cdot s \rfloor$, we compute

$$v_y = \Theta((\mu_y \cdot s) \cdot (g \cdot b)).$$

But this is just

$$v_y = \Theta(\mu_y \cdot n).$$

Thus, we can compute the difference $v_{max} - v_{min}$ as

$$v_{max} - v_{min} = \Theta(n \cdot (\max \mu_y - \min \mu_y))$$

which is

$$v_{max} - v_{min} = \Theta(\delta \cdot n)$$

Thus for an appropriate constant choice of δ the algorithm could be made to reduce the problem size by half at each step. We can compute the span of the algorithm given this choice of δ

$$T_\infty(n) = \Theta(b \cdot s) + T_\infty(n/2).$$

Note that the $\Theta(b \cdot s)$ term incurred from the serial partitioning of the groups in parallel can be re-expressed as $\Theta(b \log n)$ because we chose $s = \Theta(\log n)$. Thus we can recursively express the span as

$$T_\infty(n) = \Theta(b \cdot \log n) + T_\infty(n/2).$$

The depth of the recursion is $\Theta(\log n)$ because the input size is reduced by half at each

level. Thus, by adding up the $\log n$ terms of the form

$$b \cdot \log(n/2^i) = b \cdot (\log n - i),$$

we get the span of the algorithm, which is

$$T_\infty(n) = \Theta(b \log^2 n).$$

The number of cache misses is approximately

$$n + n/2^1 + n/2^2 + \cdots = 2n.$$

□

Proposition 3. *If we take $\delta = O(1/\sqrt{\log n})$ on the top level of recursion and then use a constant δ on lower levels of recursion, then the algorithm has span $\Theta(b \log^2 n)$.*

Proof. We could potentially achieve better performance by using a smaller δ initially, and then using a larger δ at lower levels of recursion. This means that the algorithm will reduce the size of the problem significantly on the top level of recursion and then use a δ that does not hurt its span on lower levels where it is not critical that the size gets reduced a lot. We can compute the span of the algorithm when run on the initial input n_0,

$$T_\infty(n_0) = b \cdot \Theta(\frac{\log n_0}{\delta^2}) + T_\infty(\delta n_0)$$

and for all $n < n_0$, where we use $\delta_{fake} = \frac{1}{2}$

$$T_\infty(n) = b \cdot \Theta(\log n_0) + T_\infty(\delta_{fake} n).$$

From the previous part we have shown that once we are using δ_{fake} we get span $\Theta(b \log^2 n)$, so we can have $\delta = \frac{1}{\sqrt{\log n}}$ to get span $\Theta(b \log^2 n)$, but this is much better than before. The improvement can be seen in the cache behavior analysis. The algorithm incurs n cache misses on the top layer, and then at lower layers the algorithm incurs less than

$$\frac{1}{\sqrt{\log n}} + \frac{1}{2\sqrt{\log n}} + \cdots = \frac{2}{\sqrt{\log n}}.$$

Thus, the total number of cache misses is

$$(1 + o(1))n$$

because $\frac{2}{\log n} \in o(1)$.

□

5 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 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.