Cache-Efficient Parallel Partition

Alek Westover

MIT PRIMES

2019

Introduction To The Partition Problem

Partitioning an array means moving all elements that satisfy a certain condition to the front of the array. In many applications there is a pivot value, and you would like to sort an array into a set where the values are greater than the pivot value, and a set where the values are less than the pivot value. One very useful algorithm that uses a partition step is quicksort.

Serial Solution To The Partition Problem

There is a very standard method for solving the problem in serial. The method involves creating 2 indexes on either side of the array, and moving them inwards towards each other until they collide. Here is pseudocode that illustrates the algorithm

```
$A[N]$
$pivotValue \leftarrow A[rand()\%N]$
$low \leftarrow 0; high \leftarrow N - 1$
**while** $low < high$ **do**
    **while** $A[low] \leq pivotValue$ **do**
        $low \leftarrow low + 1$
    **end while**
    **while** $A[high] > pivotValue$ **do**
        $high \leftarrow high - 1$
    **end while**
    $tmp \leftarrow A[low]$
    $A[low] \leftarrow A[high]$
    $A[high] \leftarrow tmp$
**end while**
**if** $A[low] \leq pivotValue$ **then**
    $low \leftarrow low + 1$
**end if**
```

Introducing Parallel Algorithms

This is a good algorithm in serial, but in parallel it is possible to do much better. Parallel algorithms run on multiple processors at the same time, and therefore achieve a speed up over the serial program. There are some things that can be done in parallel, and there are some things that depend on other results and therefore cannot be done until the other things have already been done. It is also important to avoid data races, when multiple processors try to overwrite a single value at the same time, which often results in non-deterministic behavior, when making a parallel algorithm. Here are some of the most important ideas in parallel algorithms.

Definition

The work of an program, T_1, is the amount of time the program takes on a single processor.

Definition

The span of an program, T_∞, is the amount of time the program takes on an infinite amount of processors.

Definition

The amount of time a program takes to run on p processors is written T_p.

There are relationships between these variables. First, you cannot expect a speed up of more than p by having p processors versus having 1 processor, so

$$T_p \geq \frac{T_1}{p}.$$

Also, we know that using ∞ processors will make a program faster than merely using p processors so.

$$T_p \geq T_\infty.$$

Also Brent's theorem tells us that

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

Here is a proof
Looking at the graph of the algorithm, we see that T_∞ is the height of the graph, and at each level i let there be x_i operations that must be done. Then,

$$T_p = \sum_{i=1}^{N} \left\lceil \frac{x_i}{p} \right\rceil \leq \sum_{i=1}^{N} \left(\frac{x_i}{p} + 1 \right).$$

Thus

$$T_p \leq \frac{1}{p} \sum_{i=1}^{N} x_i + N.$$

As already stated $T_\infty = N$, and we see that the total sum of all operations that must be done is the work, i.e. $\sum_{i=1}^{N} x_i = T_1$

Thus,

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

This is Brent's theorem. Putting this together with our other results, we have that

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

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

Adding the equations and dividing by 2,

$$\frac{1}{2}(T_\infty + \frac{T_1}{p}) \leq T_p \leq T_\infty + \frac{T_1}{p}.$$

This is a very strong bound on T_p which shows that computing span and work of an algorithm tells you about how it will run on p processors.
Parallelism is defined as $\frac{T_1}{T_\infty} \leq p$.

Parallel prefix sum

An important algorithm is parallel prefix sum. This can be done in parallel with span $O(\log N)$ as follows

```
function PARALLEL_PREFIX_SUM(A)
    for $i \leftarrow 1$ to $N/2$ do
        $B[i] \leftarrow A[2 \cdot i] + A[2 \cdot i - 1]$
    end for
    $B \leftarrow parallel\_prefix\_sum(B)$
    $C[1] \leftarrow A[1]$
    for $i \leftarrow 2$ to $N$ do
        if $i \mod 2 \equiv 0$ then
            $C[i] \leftarrow B[i/2]$
        else
            $C[i] \leftarrow B[i/2] + A[i]$
        end if
    end for
    return $C$
end function
```

Old solution to parallel partition

Note that in the algorithm mentioned for parallel prefix sum the for loops can be parallelized, making it so that the top level of recursion takes span $O(1)$, and by recursing we get $T(N) = T(N/2) + O(1)$, which means that $T(N) = \log(N)$. Also, clearly the work is $O(N)$. We can use this to make a parallel partition program.

- Create a binary array where the values represent whether or not an element in the array is a predecessor or successor. (Span contribution: $O(1)$)
- Parallel prefix sum this array. (Span contribution: $O(\log(N))$)
- Using the output of parallel prefix sum as indexes write the values in the array to a new array. (Span contribution: $O(1)$)

By using this method of parallel partitioning at each of the $\log(N)$ levels (with high probability) of recursion for quicksort, we get an algorithm that has span

$$T(N) = O(\log N) + T(N/2)$$

$$T(N) = O(\log^2 N)$$

And work:

$$T(N) = O(N) + T(N/2)$$

$$T(N) = O(N \log(N))$$

With high probability

The idea of "with high probability" is that the probability is $1 - n^{-c}$ for constant c of our choice. This is a very useful concept, especially because it allows us to union bound.

Bill's in place algorithm

$\boxed{\text{The above algorithm is not in place however, which is not great.}}$

Solution with the Ps to parallel partition

Initial algorithm: P_is are chosen as $A[i], A[i+t], \ldots$ and then partitioned. Strenghts: Given the assumption that the elements in the list are all randomly with $1/2$ probability greater than or less than the pivot then with high probability the size of the problem to recurse on in partitioning is much smaller than the size of the original input. Weakness: This only has theoretical guarantees if every element in the input array is randomly greater than or less than the pivot. This is not desirable, because a non-random input could potentially make this algorithm perform very poorly.

Grouped P algorithm

Join P_is. In this algorithm once the P_js are made, we join together polylogarithmically many of them into a group G_i. These groups now with high probability contain close to $1/2$ predecessors, thus making the size of the problem to recurse on small. The groups are generated by randomly permuting the numbers $0 \ldots t - 1$ and assigning a chunk of the permuted array to each group. Strengths: We don't need to make incredibly restrictive assumptions about the input. Weaknesses: It uses a lot of extra space.

Grouped P less space algorithm

Store group information in a single small array. If we decide to instead of randomly permuting the array, use a more restrictive scheme for generating groups, then we can represent groups with less space. We divide the numbers $0, 1, \dots t-1$ into chunks, and choose a single number within each chunk. These numbers are stored. We then can use the single number per chunk to encode all the groups, by adding the group index in the modulus of the chunk size to the value stored for the value chosen on a specific chunk. Strengths: Removed the extra space! Weaknesses: The P_js are now unnecessary, and should be taken out (by making $|P_j| = 1$).

Final algorithm

We logically partition the array $A = A[0], A[1] \dots A[n-1]$ into blocks P_j each of b **adjacent** elements. There are $t = \frac{n}{b}$ of these blocks. We then generate a random array $X = X[0], X[1], \dots, X[s-1]$ where each element in X contains an integer chosen randomly at uniform from 0 to $\frac{t}{s} - 1$. The values in X will encode $\frac{n}{s \cdot b}$ groups of P_js. The first group $G_0 = P_{X[0]} \cup P_{X[1]+t/s} \cup P_{X[2]+2t/s} \cup \dots$ and similarly $G_i = P_{(X[0]+i) \mod (t/s)} \cup P_{(X[1]+i) \mod (t/s)+t/s} \cup \dots$. Once we have generated X, we can perform a serial partition on each group in parallel. While doing this we keep track of the middle of each array. Denote by v_{min} the lowest index of a middle, and v_{max} the highest index of a middle. Everything before v_{min} is a predecessor and everything after v_{max} is a successor. Thus, by recursing on v_{min} to v_{max} we complete the partitioning.

Proof of Correctness with high probability

First, we use Hoeffding bounds (a slight generalization of Chernoff bounds which applies to random variables that take values from $[0, 1]$ rather than only binary random variables which take values from $\{0, 1\}$) to analyze the relationship between our results in terms of size reduction as we recurse and the probability of the reduction occurring. We denote by $f(j)$ the number of predecessors in P_j. Then the number of predecessors in G_0 for example, is

$$\sum_{i=0}^{s-1} f(X[i] + i \cdot t/s).$$

We are interested in the mean of this quantity given the assumption that half of the array A is composed of predecessors. First note that the maximum possible value of the sum is $s \cdot b$, if every element in every P_j as part of the group is completely filled with predecessors, and the minimum value is 0, when the group is filled with successors. Now we compute

Proof of Correctness with high probability

$$\mathsf{E}(\sum_{i=0}^{s-1} f(X[i] + i \cdot t/s)).$$

By linearity of expectation this is

$$\sum_{i=0}^{s-1} \mathsf{E}(f(X[i] + i \cdot t/s)).$$

Expanding the expectation (remember that $X[i]$) is a random variable uniformly distributed from 0 to $\frac{n}{b \cdot s}$ this is

$$\sum_{i=0}^{s-1} \sum_{k=0}^{k=\frac{t}{s}-1} \frac{s}{t}(f(k + i \cdot t/s)).$$

Rewriting the sum, we have

$$\frac{s}{t}\sum_{p=0}^{t-1} f(p) = \frac{s}{t} \cdot \frac{t \cdot b}{2} = \frac{b \cdot s}{2}.$$

Proof of Correctness with high probability

Next we must normalize the sum,

$$\mathrm{E}(\frac{1}{s}\sum_{i=0}^{s-1}\frac{1}{b}\cdot f(X[i]+i\cdot t/s))=\frac{1}{2}.$$

We are looking for the probability that

$$\left|\frac{1}{s}\sum_{i=0}^{s-1}\frac{1}{b}\cdot f(X[i]+i\cdot t/s)-\frac{1}{2}\right|>\delta.$$

Hoeffding's inequality gives us that the probability of this occurring is

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

Note that for now we have only been looking at a single group, G_0. By symmetry the probability of deviation of more than δ from $\frac{1}{2}$ of the fraction of predecessors in any other group is the same as in G_0. In order to apply the union bound, we want each of the probabilities to be at most

Proof of Correctness with high probability

$$\frac{s \cdot b}{n} \cdot \epsilon$$

so we can set $\exp(-2(s)\delta^2) = \frac{s \cdot b}{n} \cdot \epsilon$. Solving, $s = \frac{\ln(n) - \ln(sb\epsilon)}{2\delta^2}$ but we can neglect the $\ln(sb\epsilon)/(2\delta^2)$ getting $s = \Theta(\frac{\ln(n)}{\delta^2})$.

Tradeoffs - Recursing With Other Algorithm

From previous work we know that for any δ we can already do a parallel partition of $\delta \cdot n$ things with span $O(\log(n\delta)\log\log(n\delta))$. This makes the span of our algorithm, with this choice of a recursive technique,

$$T(n) = b \cdot s + O(\log(n\delta)\log\log(n\delta))$$

Because the serial partition step has a span of $s \cdot b$, which using our expression for s is

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

We don't need the δ in the $n\delta$ for the asymptotics since we are taking the log $n\delta$, so we can simplify the expression to,

$$T(n) = b \cdot \Theta(\frac{\log n}{\delta^2}) + O(\log(n)\log\log(n)).$$

Tradeoffs - Recursing With the Same Algorithm - Constant δ

We can instead actually recurse with our algorithm.
There are 2 ways we could do this. Let us use $\delta = \frac{1}{2}$ on all levels of recursion (even the top).

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

$\delta = \frac{1}{2}$ so the problem size is repeatedly cut in half, so the depth is $\log n$. By adding up the terms we get

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

Tradeoffs - Recursing With the Same Algorithm - Smaller δ initially

We could achieve much better performance by using a smaller δ initially, and then using $\delta = \frac{1}{2}$. This allows us to quickly cut down the size of the problem, and then use a δ that does not hurt our span. For span we are going to get,

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

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

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

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 it is much better than before.

Acknowledgments

I would like to thank

- The MIT PRIMES program
- Bill
- My parents