6.6050 Algorithm Engineering
Problem Set 1

Natasha K. Hirt
nhirt@mit.edu

October 6, 2025

1 Parallel Integer Sorting

1.1 Efficient algorithms to compute all of the *number(v,s)* and *serial(i)* values as described above. Analyze the total work and span of your algorithms in terms of r and n

.

For the following, I assume that input array $B[1 \dots n]$ is the concatenated array of individually sorted subsets of array A.

The purpose of `number(v, s)` is to identify how many elements in subarray `s` share the same value as `v`. If one were to perform this for all values $[0, \dots, r-1]$ that can exist in our array and stored them as a `numbers` array it might look as follows:

0	0	0	1	1
0	3	4	4	4
0	1	2	3	4
0	0	1	2	3

B as a 2D array

3	2	0	0	0
1	0	0	1	3
1	1	1	1	1
2	1	1	1	0

numbers

Table 1: Example of a 2D representation of an input array B (left) and the corresponding `numbers` array (right) for all values $[0, \dots, r-1]$. Subarrays are shown as rows.

For example, because there are three 0's in the first row of B, numbers$(0, 0) = 3$.

The algorithm assumes that variables B and r exist in the global namespace and takes v, s as input.

We have two options for finding the number of values in the subarray. The first takes the form of a simple (serial) counter. Since each segment is scanned once, a single iteration of this algorithm would be performed in $O(r)$ per call. One could make a faster version of this using binary search to find the left and the right bounds of the section of B (this is possible because B has been pre-sorted for us). I'd maybe argue that the serial version could give us more opportunities for parallelization since we can perform the operation for every unique value between $[0, \dots, r-1]$ together, but for individual calls the binary search version $O(\log r)$ is superior in terms of work ($O(\log n)$ in the worst case. The convention of using $O(\log n)$ in place of $O(\log r)$ will be continued for the rest of this

homework assignment, even if technically the list we are iterating over is length r). Unlike in the table above, B is represented as a 1D array in the following algorithm. The span is theoretically $O(1)$ for the serial version because we can make all comparisons in parallel (realistically $O(\log n)$, though, to prevent read/write conflicts), and $O(\log n)$ for the binary search version because the binary search is a critical path.

Algorithm 1 `number(v, s)`

Require: Array $B[0 \dots n-1]$ (sorted), integer v, segment index s, segment size r
Ensure: Returns the number of times v appears in segment s of B

```
left ← s · r                                              // start index of row
right ← (s + 1) · r − 1                                   // end index of row
first_v ← binary_search(B, v, left, right, first)
last_v ← binary_search(B, v, left, right, last)
if first_v = −1 then
    return 0                                              // no v in subarray
end if
return last_v − first_v + 1
```

The key with `serial(i)` is to recognize that, thanks to `numbers(s,v)`, we already have the information needed to place a tight bound on where i is in the subarray (based on its value $B[i]$), the subarray's position in the larger array ($s \cdot r$), and within that smaller subset of values can quickly find what rank i has in its subarray. This algorithm performs $O(r)$ work in $O(\log n)$ span (we are summing an array of length r and these are the exponential bounds of parallel sum).

Algorithm 2 `serial(i)`

Require: Array $B[0 \dots n-1]$ (sorted), integer i, Array numbers
Ensure: Returns the number of values equal to $B[i]$ that appear before i in its subarray.

```
s ← ⌊i/r⌋
lessthan ← Σ_{x=0}^{B[i]−1} numbers[row, x]
left ← s · r + lessthan                                   // start index of binary search
return i − lessthan + 1
```

1.2 Computing the global rank offset for parallel output

Now we can think about using these two functions to calculate the global rank offset for any given index i. This is the final position of every index in the sorted array. This function is an equation that takes the outputs of the `number` and `serial` functions we've defined and returns a global rank.

First, sum over all `number(v, s)` for all values of $v < B[i]$ to get the offset of that value v overall (how many numbers are there less than v?) This has work $O(r \cdot \frac{n}{r}) = O(n)$, and span $O(r)$ if we precompute the `Numbers` array (else we add a factor of $\log n$). Then sum over all `Numbers(v, row)` for all values of $row < s - 1$ to see how many elements share the same value prior to the occurence of our $B[i] = v$ in earlier subarrays. This does not change the exponential work but does have a span of $O(r)$. Finally, for the specific subarray, get `serial(i)`, which has work $O(r)$ and span $O(\log n)$.

The total work is consequently $O(r)$. The span is $O(n + n/r + r)$.

Algorithm 3 `rank(i)`

Require: Array $B[0 \dots n-1]$ (sorted), integer i, segment size r, 2D array `Numbers` storing output of `Numbers(s,v) for all combinations of` s `and` v.
Ensure: Returns the global rank of $B[i]$ in the output

1: $s \leftarrow \lfloor \frac{i}{r} \rfloor$ // segment (row) index (0-based)
2: $rank \leftarrow 0$
3: **if** $B[i] > 0$ **then**
4: **for** $v = 0$ **to** $B[i] - 1$ **parallel do**
5: $rank \leftarrow rank + \sum_{s=0}^{n/r-1} \text{Numbers}[s, v]$ // sum over all columns for values less than $B[i]$
6: **end for**
7: **end if**
8: **for** $row\_idx = 0$ **to** $s - 1$ **parallel do**
9: $rank \leftarrow rank + \text{Numbers}[row\_idx, B[i]]$ // sum over all previous v
10: **end for**
11: $rank \leftarrow rank +$ `serial`(i) // add segment offset
12: **return** $rank$

Some thoughts on improving this algorithm in its in-place form, since it would be impractical to be dependent on a pre-calculated array Numbers that would take up n memory. We can improve the span of the algorithm by using parallel algorithms as introduced in lecture. If we keep the within-row accumulation on line 5 serial (span $O(r)$), we can replace the accumulations across columns for elements with the same value in span $O(\log(n/r))$ (parallel sum over the number of rows in each column). For the values less than the desired element but of the same value we can also perform a column reduction ($O(\log(n/r))$). Finally, the serial component has span $O(\log n)$. In sum, we get a per-query span of $O(r + \log(n/r) + \log(n/r) + \log n)$ which can reasonably be simplified to $O(r + \log(n))$ (even if we save memory by performing these calculations on the fly using the functions we provided above, and do not pre-calculate our Numbers array!)

1.3 Overall work and span complexities

Section	Work	Span
`number(v, s)` (serial scan)	$O(r)$	$O(\log n)$
`number(v, s)` (binary search)	$O(\log r)$	$O(\log n)$
`serial(i)`	$O(r)$	$O(\log n)$
`rank(i)`	$O(r)$	$O(r + \log n)$
`rank(0...n-1)`	$O(nr)$	$O(n(r + \log n))$

Table 2: Work and span complexities for each section and approach.

1.4 Is it stable?

Yes, this algorithm is stable because the relative order of individual elements is retained in the final sorted array A'. This is enforced by the ordered nature of passing through individual subarrays

and the serial function. Each unique location of $B[i]$ maps onto a unique location A'.

1.5 Is it in-place?

If you use an on-the-fly approach, yes. The only extra storage is used for temporary counting variables ($O(1)$ memory complexity). If you precompute Numbers, you gain a lot of speed (and do a lot less work because everything is done up-front) but you also need to store an extra array `numbers` of the same size as the input array.

1.6 How to do stable sorting in O(kn) work and O(k(pow(r, 1/k) + log n)) span

Well, going off the clue, individual digits will be important to this. Radix sort works by breaking down numbers into individual digits and iteratively sorting by significant digit (e.g. sorting the most significant digit, then the second most significant, ... etc. to the least significant digit), essentially turning a bigger sorting problem into a series of small ones.

Say we imagine our sorting algorithm as the subroutine that does the sorting of each level of significant digit. We can represent each of our values in base $r^{1/k}$, giving us k subsorting problems (one for each significant digit). For each level of significant digit, n integers in range $[0, r^{1/k} - 1]$ are sorted. Thanks to our choice of base, there are k levels to sort k times using our subroutine, so the total work done is $O(kn)$ instead of $O(rn)$.

If we break down the desired span, $O(k(r^{1/k} + \log n))$, we see a similar multiplicative factor of k owing to the number of passes for each digit. This leaves span $r^{1/k} + \log n$ for each pass of the sorting algorithm, which, if we did one pass ($k = 1$, as in the original setup) would reduce to $r + \log n$. This is exactly the result we have for the regular algorithm evaluated above.

2 Tree Properties

2.1 Efficient algorithm to compute DFS number of each node in the tree

The way I thought about this problem was as follows. We begin with a set of nodes as a linked list where we can readily ($O(1)$) access the node's parents. We also have the Euler tour, which is also given as a linked list. We need to find the order of the first occurences of every element in the Euler tour, which is nonobvious from the linked list structure but will give us the DFS order (since the Euler tour is essentially a DFS where the retreat edges are made explicit.)

In lecture, we learned that we can find the distance of any node in a linked list from the end using pointer jumping. Since we can access each first occurence in the Euler list easily ($O(1)$), let us frame this problem as one which counts the distance to the end but only in terms of first occurences. The first occurence of any value i is accessed via the Euler tour ($euler\_tour[i]$).

Since the output will be the number of elements to the end of the Euler tour we have to subtract the final "distance" value from k to get the DFS value.

The work of pointer jumping over n values in the Euler tour is $O(n \log n)$ and the span is $\log n$. To iterate over the k nodes in the tree we use $O(k \log k)$ work and $\log k$ span. In the worst case, $k = n$ and we have $O(n \log n)$ work and $\log n$ span overall.

Algorithm 4 Compute DFS order of nodes in an Euler tour

Require: Euler tour $curr$, length of Euler tour n, number of nodes k
Ensure: Array DFS where $DFS[i]$ is the DFS order of node v

```
1: Initialize array distance of length n with all entries set to 0
2: Initialize array DFS of length k with all entires set to 0
3: for i = 0 to k − 1 parallel do
4:    distance[euler_tour[i]] ← 1                    // k first elements in euler tour get rank 1
5: end for
6: for j = 0 to ⌈(log n)⌉ − 1 do
7:    for i = 0 to n − 1 parallel do
8:       temp ← distance[i.next]
9:       temp_parent ← i.next.next
10:   end for
11:   for i = 0 to n − 1 parallel do
12:      distance[i] ← distance[i] + temp
13:      i.next ← temp_parent
14:   end for
15: end for
16: for i = 0 to k − 1 parallel do
17:   DFS[i] = k − distance[euler_tour[i]]           // reverse the order
18: end for
19: return DFS
```

2.2 Efficient algorithm to compute the rank of each node in the tree

To find the depth of a node in the list we can pointer jump over the parents (since we can readily access those as well) in $O(k \log n)$ work and $O(\log n)$ span (not $O(n \log n)$ work because we're not looking at every value in the list n. The list of starting nodes is only k long, and work scales with the active nodes we're investigating). This gives us the rank list for each node.

Algorithm 5 Compute rank order of nodes in an Euler tour

Require: Euler tour *curr*, length of adjacency list k, parent array P length k
Ensure: Array *rank* where $rank[v]$ is the rank of node v

```
1: Initialize array rank of length k with all entries set to 1
2: rank[0] ← 0
3: for j = 0 to ⌈(log k)⌉ − 1 do
4:    for i = 0 to k − 1 parallel do
5:       temp ← rank[P[i]]
6:       temp_parent ← P[P[i]]
7:    end for
8:    for i = 0 to k − 1 parallel do
9:       rank[i] ← rank[i] + temp
10:      P[i] ← temp_parent
11:   end for
12: end for
13: return rank
```