Theoretical Questions

Nitsan BenHanoch 208585927

I WAS ALLOWED TO SUBMIT THIS WORK LATE

Lior approved this because she is very nice <3

Part I - A

> Determine whether the parentheses in a given mathematical equation are balanced.

Pseudo Code (C style)

```
char parenthesis_sign(char ch) {
  switch (ch) {
    case '(': return +1;
    case ')': return -1;
    default : return  0;
  }
}

bool are_parentheses_balanced(char expr[], int N) {
  int signs[N], balance[N];

  for (int i = 0; i < N; i++) do in parallel {
    signs[i] = parenthesis_sign(expr[i]);
  }

  balance = parallel_prefix_sum(signs);

  bool never_negative = true;

  for (int i = 0; i < N; i++) do in parallel {
    if (balance[i] < 0) {
      never_negative = false;
    }
  }

  return never_negative && balance[N - 1] == 0;
}
```

Explanation

The classic sequential algorithm scans the input left to right, incrementing a balance on every `(`, and decrementing it on every `)`.

It's already pretty much a prefix-sum, if we treat `(` as `+1` and `)` as `-1`; We just need to check that at no point the balance is negative, and that the final balance is zero.

Here we use parallel-prefix-sum as a black box, then verify (in parallel) that at no point the balance is negative.

Note that different threads assigning `never_negative = false` in parallel is *not* a race condition (there are no reads).

Correctness If the expression **is not balanced**, then **either** * there's a `)` with no a preceeding `(`, or * there's a `(` with no succeeding `)`.

In the first case, the balance at that `)` will be negative, and the algorithm will return false; In the second case, the *final* balance will be positive, and the algorithm will return false.

If the expression **is balanced**, then **both** * every `(` will increase the balance by 1, but it'll come with exactly one succeeding `)` that will decrease it by 1; so the final balance will be zero. * no `)` will come before its matcing `(`, so at no point will the balance go negative.

Hence, the algorithm will return true.

Complexity In class we saw that `parallel_prefix_sum` can be done in `O(log n)` time.

My additions can be done in `O(1)` (given `N` cores).

Hence `are_parentheses_balanced` can be run in `O(log n)` time.

(By the way, for the same reasons, it's also `O(n)` work).

Part `I - B`

> Find the number of continuous sub arrays with a given sum.

In the forum, Saar said we can assume all values are positive.

Pseudo Code (C style)

```
bool binary_search(int arr[], int N, int x) {
  // returns whether arr contains x
  // using classic sequential binary search
}
```

```
int continuous_sub_arr_sum(int arr[], int N, int target) {
  int arr_cumsum[N], results[N], results_cumsum[N];

  results[0] = (arr[0] == target) ? 1 : 0;

  arr_cumsum = parallel_prefix_sum(arr);

  for (int i = 1; i < N; i++) do in parallel {
    int complement = arr[i] - target;
    bool found = binary_search(arr, i, complement);
    results[i] = found ? 1 : 0;
  }

  results_cumsum = parallel_prefix_sum(results);

  return results_cumsum[N - 1];
}
```

Explanation

As an example, consider * arr = [3, 2, 4, 5, 1, 9, 8, 1, 2, 6] * target = 11

After prefix-sum we get * arr_cumsum = [3, 5, 9, 14, 15, 24, 32, 33, 35, 41]

Notice 14 - 3 = 11, and 35 - 24 = 11.

We'll say that y is x's *complement*, if x - y == target.

Each element in arr_cumsum can determine whether it has a preceding complement using binary search. Note that since i arr[i] > 0, arr_cumsum is strictly monotonous.

The first element is a special case (it has no preceding values), so we add a check for it.

But we still need to tackle **parallel mutual counting**!

- Letting threads increment a shared counter is a race condition.
- Protecting the counter with a mutex gives O(n) runtime.

Solution: We create an array of results. Each thread sets their result to either 0 or 1. Afterwards, we sum the results (in parallel) in O(log n) time.

Complexity Assuming N threads, `parallel_prefix_sum(arr)` takes `O(log n)` time.

The binary search takes another `O(log n)` for each thread in parallel.

The last `parallel_prefix_sum(results)` again adds `O(log n)` time.

Total: `O(log n)` time.

(Work is `O(n log n)`, because N threads peform logarithmic binary search).

Part `II`

Find the k-th ancestor of each node in an up-pointing tree.

```
Input: k, up-pointing tree (encoded as an array)

    k = 2

    indices: 0, 1, 2, 3, 4, 5, 6, 7, 8, 9
    array = [0, 0, 0, 1, 1, 2, 3, 5, 5, 5]

       0
      / \
     1   2
    / \   \
   3   4   5
   |      /|\
   6     7 8 9


Output: k-th ancestor of each node, or -1 if no such

    [-1, -1, -1, 0, 0, 0, 1, 2, 2, 2]
```

Pseudo Code (C style)

```
void parallel_copy_into(int dst[], int src[], int N) {
  for (int i = 0; i < N; i++) do in parallel {
    dst[i] = src[i];
  }
}

void deterministic_pointer_jumping(int arr[], int map[], int N) {
  // a deterministic version of pointer jumping,
  // working on a copy of arr, to avoid race conditions.

  int temp[N];

  for (int i = 0; i < N; i++) do in parallel {
    if (arr[i] > -1) {
      temp[i] = map[arr[i]];
    }
```

```
  }

  parallel_copy_into(arr, temp, N);
}

void each_node_kth_ancestor(int arr[], int results[], int N) {

  //////////////////////
  // step 1: find root //
  //////////////////////

  int root;

  // copy arr into results
  parallel_copy_into(results, arr, N);

  // one round of pointer jumping is enough
  deterministic_pointer_jumping(results, results, N);

  // root will be the only node pointing to itself
  for (int i = 0; i < N; i++) do in parallel {
    if (results[i] == i) {
      root = i;
    }
  }

  ////////////////////////////////////
  // step 2: make root point nowhere //
  ////////////////////////////////////

  arr[root] = -1;

  //////////////////////
  // step 3: money time //
  //////////////////////

  // copy arr into results
  parallel_copy_into(results, arr, N);

  while (k) {
    if (k & 0x1) {
      // pointer jumping in results based on arr
      deterministic_pointer_jumping(results, arr, N);
    }
    // pointer jumping in arr
    deterministic_pointer_jumping(arr, arr, N);
```

```
    k = k >> 1;
  }
}
```

Explanation

My approach takes inspiration from the famous `Exponentiation by Squaring` algorithm.

```
double fast_pow(double x, unsigned int n) {
  double result = 1.0;
  while (n) {
    if (n & 0x1) {
      result *= x;
    }
    x *= x;
    n >>= 1;
  }
  return result;
}
```

We perform regular pointer jumping in **`arr` on every iteration**. * At iteration `i`, `arr` has links of distance `2 ** i`.

We perform *guided* pointer jumping in **`results` on some iterations**. * We only jump if the i'th bit of k is set. * We advance each result by a distance of `2 ** i` (using `arr`).

Example Let's say `k = 43`; its binary representation is `0b101011`, accounting to the fact that `43 = 32 + 0 + 8 + 0 + 2 + 1`.

`i`	`arr`	`results`
0	jumps of 1	jump += 1
1	jumps of 2	jump += 2
2	jumps of 4	
3	jumps of 8	jump += 8
4	jumps of 16	
5	jumps of 32	jump += 32

Correctness Normal pointer jumping (as in class) corresponds with finding ancestors at distance of **at least** k.

However, in our case we need to find ancestors at distance of **exactly** k. Also, nodes without a k'th ancestor should point to `-1`.

In this algorithm, we start by we making the root point to `-1`. We also treat `-1` as a black whole (once a node gets to `-1`, it'll stay there forever).

The trick of selectively using powers of 2, with `-1` being a black whole, guarantees we'll indeed find **exact** k'th ancestors.

Complexity Assuming N cores.

`parallel_copy_into` takes `O(1)` time.

`deterministic_pointer_jumping` takes `O(1)` time.

Step 1 does Pointer-Jumping once, so it's `O(1)` time.

Step 2 is `O(1)` time.

Step 3 does Pointer-Jumping for every bit of k, so it's `O(log k)`.

Total: `O(log k)` time.