Assignment 3

Christos Tsakiroglou
Ramkumar Rajagopalan
Niklas Semmler
Jiannan Guo

- Task 1
  - a) Analyze existing pseudo code
    - Assumptions
    - Example
    - Work Complexity & Step Complexity#
    - Required time
  - b) Devise new SCAN
    - Description
    - Pseudo code
    - Analysis
  - c) Recursive scan
    - Description
    - Assumption
    - Pseudo code
    - Analysis
- Task 2
    - Operator
    - Explanation
    - Proof of associativity
    - Example
- Task 3

Task 1

a) Analyze existing pseudo code

Assumptions

1. sequence = [x_1 , x_2 , ... x_n], hence length(sequence) == n
2. the "≥" in the pseudo code is a typo and should have been ">"
   - (the algorithm does not work otherwise)

Example

sequence = [x_1, x_2, x_3, x_4]

- First step j = 1
  - k = 1; $1 > 2^0 == 1 > 1 \Rightarrow FALSE\ x'_1 = x_1$
  - $k = 2;\ 2 > 2^0 == 2 > 1 \Rightarrow x'_2 = x_1 \oplus x_2$
  - $k = 3;\ 3 > 2^0 == 3 > 1 \Rightarrow x'_3 = x_2 \oplus x_3$
  - $k = 4;\ 4 > 2^0 == 4 > 1 \Rightarrow x'_4 = x_3 \oplus x_4$
- Second step j = 2 == $\log_2$ n:
  - k = 1; $1 > 2^1 == 1 > 2 \Rightarrow FALSE\ x''_1 = x'_1$
  - k = 2; $2 > 2^1 == 2 > 2 \Rightarrow FALSE\ x''_2 = x'_2$
  - k = 3; $3 > 2^1 == 3 > 2 \Rightarrow x''_3 = x'_1 \oplus x'_3 == x''_3 = x_1 \oplus x_2 \oplus x_3$
  - k = 4; $4 > 2^1 == 4 > 2 \Rightarrow x''_4 = x'_2 \oplus x'_4 == x''_4 = x_1 \oplus x_2 \oplus x_3 \oplus x_4$

Work Complexity & Step Complexity[1]

work complexity:

$$\Theta = (n - 2^0) + (n - 2^1) + \ldots + (n - 2^{(logn)-1})$$
$$= log\ n \bullet n - ((2^0) + (2^1) + \ldots + (2^{(logn)-1}))$$
$$= n\ log\ n - \frac{1-2^{logn}}{1-2}$$
$$= n\ log\ n - n + 1$$

step complexity:

$$\Theta = log\ n$$

Required time

The required time of the algorithm depends on the numbers of processors available. If the number of processors is greater than or equals to n, the step complexity gives the best-case latency for the algorithm, which is exact $log\ n$. Although, usually we only have a finite number of processors, we might need to serialize some of the available parallelism. This serialization converts some of the work complexity into latency.[2]

---

[1] Note that "log" in this documentation is "base 2 logarithm" if no explicit explain provided.
[2] Parallel Pattern 8: Scan Michael McCool
http://software.intel.com/en-us/blogs/2009/09/15/parallel-pattern-8-scan/

b) Devise new SCAN

Description

Make all processors use a sequential version of SCAN over a segment of the list. Consequently they make their last sum visible (put it into the hold_element). Then do the parallel SCAN over all of these visible numbers. In the end you need to update all segments by adding the difference of the last element after and the last element before the parallel SCAN to all elements.

Pseudo code

```
do_local_scan(my_segment):
        // local scan + make hold_element visible
        cumulative_segment = seq_SCAN(my_segment)
        self.hold_element = my_segment[-1] // last element of my_segment

divide_list_to_segments(list, num_processes):
        // similar to scatter
        // returns segments in form [[x_0, x_1, …, x_i], [x_{i+1}, x_{i+2}, …, x_j], …]
        // a list containing num_processes lists with an even share of elements
        …
        return segments

finalize_segment(my_segment):
        // add the difference between last element after SCAN
        //  and last element before SCAN to all elements
        add = self.hold_element - my_segment[-1]
        for elem in my_segment:
                elem += add

start(p, func, var):
        // starts function(variables) on the processor p

num_processors = 5 // some constant
list = [x_0, x_1, …, x_n]
processors = [Y_0, Y_1, …, Y_m]

segments = divide_list(list, num_processes);
```

```
p = 0

for segment in segments:
        start(processors[p], do_local_scan, segment)
        p += 1

parallel_SCAN(processor[p].hold_elements for p in [0, 1, …, num_processors])

p = 0
for segment in segments:
        start(processors[p], finalize_segment, segment)
```

Analysis

The algorithm requires $n \div p$ steps per processor for the inner sequential SCAN, as there are *p* processors this means $p \bullet (n \div p) \Leftrightarrow n$ steps. Plus an additional n steps to perform the outer SCAN. Finally all processors will need to do an additional $n \div p + 1$ steps. Hence the algorithm takes 3n + p steps.

As for the time, the inner SCAN will take $n \div p$, and the outer SCAN will be performed in parallel so will take exactly one time step. Finally the last part will need $n \div p + 1$ time steps. Thereby the algorithm takes i

c) Recursive scan

Description

Here we write a scan function that recursively splits the array to two sub-arrays until the leaf level is reached, where the operation is executed between the two elements. While going up the tree, the cumulative result of the previous segment is added to every number of the current segment.

Assumption

- Number of elements in sequence is a power of 2

Pseudo code

```
split_in_two(range_seq):
        last = len(range_seq)
        half = last / 2
        return (range_seq[1:half], range_seq[half+1:last])
```

```
do_recursive_SCAN(seq):
        range_seq = range(1, len(sequence))
        do_recursive_SCAN2(range_seq, seq)

do_recursive_SCAN2(range_seq, seq):
        if len(range_seq) > 2:
        // going down, intermediate nodes => splitting
                (new_range_seq1, new_range_seq2) = split_in_two(range_seq)
                start_another_process(do_recursive_SCAN2(new_range_seq1, seq))
                do_recursive_SCAN2(new_range_seq2, seq)
        else:
        // leaf nodes => increase second element by first
                seq[range_seq[1]] ⊕= seq[range_seq[0]]
                return

        for i in new_range_seq2:
        // going up, intermediate nodes =>
        //   add the cumulative result of previous segment to next segment
                seq[i] ⊕= seq[new_range_seq1[-1]]
        return
```

sequence = [x_1,x_1,...,x_n]

do_recursive_SCAN(sequence)

Analysis

Number of Operations:

Case leaf nodes: 1 step

Case other nodes: n/(2^(depth of tree starting from 1))

In every layer of the tree n/2 steps are executed.

There are log n layers of the tree.

The number of steps is:

$$\frac{n}{2} \log n$$

By comparing the total number of operations between algorithms (a) and (c), we observe that they are equal for n = 2, while the recursive algorithm needs less operations for any n>2.

Task 2

According to the algorithm provided, we devised operator as below:

Operator

$(A_n,\ F_n) \oplus (A_m,\ F_m) = (A_n \bullet \neg(F_{n+1} \vee F_{n+2} \vee ... \vee F_{m-1} \vee F_m) + A_m,\ F_m),\ n < m,\ n,\ m \in N^*$

Explanation

The operation is to detect 1 between two operands, see the scenario below. If there's at least one "1" existing between them, A_n will not have impact on A_m, hence the result would be A_m.

A [… A_n … … … A_m …....], $n < m$
F [… … … 1 … … … …]

On the other hand, if there's no "1" between two operands, which means all zero in between, A_n will have direct impact on A_m, and the result would be $A_n + A_m$.
Therefore, to parallelize scan for this operation, each processor will need to have the full knowledge of F and local segment of A.

Proof of associativity

$[(A_s, F_s) \oplus (A_q, F_q)] \oplus (A_p, F_p)$
$= (A_s \bullet \neg(F_{s+1} \vee ... \vee F_q) + A_q) \bullet \neg(F_{q+1} \vee ... \vee F_p) + A_p,\ F_p)$
$= (A_s \bullet \neg(F_{s+1} \vee ... \vee F_q) \bullet \neg(F_{q+1} \vee ... \vee F_p) + A_q \bullet \neg(F_{q+1} \vee ... \vee F_p) + A_p,\ F_p)$
$= (A_s \bullet \neg(F_{s+1} \vee ... \vee F_q \vee F_{q+1} \vee ... \vee F_p) + A_q \bullet \neg(F_{q+1} \vee ... \vee F_p) + A_p,\ F_p)$
$= (A_s, F_s) \oplus [(A_q, F_q) \oplus (A_p, F_p)]$
$(s < q < p,\ s, q, p \in N^*)$

Example

A=[1,4,2,3,1,6,5,2,1,1]
F=[1,0,0,1,0,1,1,0,0,0]
Index starts from 1 to 10.
$(A_1,\ F_1) \oplus (A_2,\ F_2) = (1,\ 1) \oplus (4,\ 0) = (1 \bullet \neg(0) + 4,\ 0) = (5,\ 0)$
$(A_1,\ F_1) \oplus (A_3,\ F_3) = (1,\ 1) \oplus (2,\ 0) = (1 \bullet \neg(0 \vee 0) + 2,\ 0) = (3,\ 0)$
$(A_2,\ F_2) \oplus (A_5,\ F_5) = (4,\ 0) \oplus (1,\ 0) = (4 \bullet \neg(0 \vee 1 \vee 0) + 1,\ 0) = (1,\ 0)$

Task 3

- distributed memory systems
  - requires messaging

    - in our case this would be closest to 1b)
- shared memory systems
    - basically every algorithm is possible
    - in that case the best algorithm appears to be the tree 1c)
- GPU-like coprocessor
    - in a GPU you have to make use of the topology so the different layers of processing
    - basically a combination of segmentation and loop unrolling does the trick

http://research.nvidia.com/sites/default/files/publications/nvr-2008-003.pdf