Homework 1

Parallel Scan(Or)

José Javier González Ortiz

September 14, 2015

Note: It has been considered as the worst case scenario in this problem the case (t,f,f...,f) which will have the solution (t,t,t...,t) and therefore will require $O(p)$ operations between all the processors to make everything t. Depending on how are we able to distribute this operations between the processors we will arrive to a better or worse lower bound

1 Completely Connected Computer

Given a fully connected computer, we can think of a lower bound by intuition using a Divide and Conquer approach. Since we are limited to p processors we can expect a $\Omega(\log p)$ lower bound. However, since the final result is not a sum of all the elements, but a whole array of results, we cannot use a traditional Divide and Conquer algorithm.

The proposed algorithm is based on the following idea: in every step each processor sends its value v_j to the one 2^i next to him (if it exists) and receives a value from the one 2^i previous to him (again, if it exists). After receiving the value, each processor makes the logical or of that value and its own and storing the result as its new v_j. From this description we can see that the algorithm will take $\Theta(\log p)$ steps. Since receiving and sending do not get blocked in this case, they will both take $\Theta(1)$ time. It is also worth noticing that p does not need to be a power of two since the algorithm will omit the messages to processors with *ids* greater or equal to p.

1.1 Algorithm

The proposed algorithm is described below in Algorithm 1.

Algorithm 1 Fully Connected Parallel Scan(or)

Require: Each processor has the common value p and the local values id and v

```
1: procedure ScanOr
2:     $i \leftarrow 0$
3:     while $2^i < p$ do
4:         if $id + 2^i < p$ then
5:             $send(id + 2^i, v)$
6:         if $id - 2^i \geq 0$ then
7:             $receive(id - 2^i, w)$
8:             $v \leftarrow w \vee v$
9:         $i \leftarrow i + 1$
10:    return $v$
```

1.2 Correctness

For correctness, we can observe that given any input the solution will be

$$(\underbrace{f, f \dots, f,}_{a \text{ times}} \underbrace{t, t \dots, t}_{p-a \text{ times}}) \tag{1.1}$$

where p is the size of the array and a the index of the first t. Therefore, only the first t matters and we want to make sure that it spreads through all the array. Given the worst case, where $a = 0$, processor 0 will send to processor 1 the t during the first iteration, processor 0 will send to processor 2 and processor 1 will send to processor 3 and so on, until we go beyond p where the processors will stop sending their values. At most we will use $\Theta(\log p)$ iterations.

Given any other case where $a > 0$, by induction we can reduce the problem to the worse case scenario. Processors below a will only send f, which has no effect, and form processor a will reach every processor $b > a$ in the same manner as stated before. It will take less time to arrive to the desired result, however in total time the algorithm will take the same amount of time since during the last iterations the processors will be sending useless f values. It is not easy to get rid of those operations since a processor cannot predict when it will receive a t value and the *receive* operation is *blocking.*

To prove correctness we can refer to equation 1.1 and claim that our algorithm will send a $t \quad \forall b > a$ with a being the index of the first t, therefore arriving to the desired solution. $\forall b \leq a$ the values are correct from the start and the *send* operation will only send messages to processors with bigger ids. For processors with ids $b > a$ we can prove by contradiction that the algorithm is correct.

Lets suppose that there exists one processor $c > a$ that after the algorithm remains f. Then we can calculate the difference $c - a = d$ and express it in binary

$$d = \sum_{n=0}^{\lfloor \log p \rfloor} d_n 2^n \qquad k = id + d = id + \sum_{n=0}^{\lfloor \log p \rfloor} d_n 2^n \tag{1.2}$$

However, we can see by the Equation 1.2 that starting at id there exists a binary sequence that lets us arrive at k. Therefore, for each iteration i we can follow the messages, if $d_i = 0$ we look at the same processor and wait for the next iteration and if $d_i = 1$ then we move to the processor we are sending the message to. Therefore, we will always arrive at the processor k and the t value will propagate to it. $\square$

1.3 Running time

Finally, as we have been reasoning, the running time is $\Theta(\log p)$ since there are $\Theta(\log p)$ iterations and each iteration involves $\Theta(1)$ operations: comparisons, assignments, *send*, and *receive.* Note that the *receive* step does not get blocked since the processors are perfectly paired during this step, otherwise, we could not guarantee constant time *reception.*

2 2-dimensional mesh computer

In this case we can also reason by intuition a lower bound for an ideal algorithm given the communication restraints. Since each processor is only connected to its neighbors and in the worst case we will have to propagate a t from one corner to the whole grid, we can see that even the fastest algorithm has a lower bound $\Omega(\sqrt{p})$ to propagate the t from one corner to another.

Therefore a good approach involves applying the *ScanOr* in parallel to each row from left to right and then communicating the accumulated last row values down. Finally, we propagate the accumulated last column value from right to left, being careful to propagate only if the t came from the upper column and not from the current column.

We can apply this approach since firstly we are calculating the *ScanOr* for each row. Once we have finished the first step, the first row will have the correct values. Subsequent rows will be correct if and only if there are no t values in any of the cells in the last column above that row. Otherwise, the whole row will have to turn to t and subsequently the rows below.

It is worth noting that the communication between elements in the same row happens in a serial fashion. This is because there is no faster way of communicating the v_j values in a line. Therefore, since we are doing three steps of this type we can expect a $\Theta(\sqrt{p})$ running time.

All of this behavior is reflected in the Algorithm 2.

2.1 Algorithm

Algorithm 2 2D Mesh Parallel Scan(or)

Require: Each processor has the common value p and the local values id and v

```
procedure ScanOr
    row ← $\lfloor id/\sqrt{p} \rfloor$
    column ← $id - \text{row}\sqrt{p}$

    if column > 0 then                          The processors receive, apply the or and send to the right
        $receive(id-1,\ w)$
        $v \leftarrow w \vee v$
    if column < $\sqrt{p}-1$ then
        $send(id+1,\ v)$

    if column = $\sqrt{p}-1$ then
        if row > 0 then
            $receive(id-\sqrt{p},\ u)$          The processors in the last column receive from the one
            $send(id-1,\ u)$                    above, send the untouched value of the previous column to
            $v \leftarrow u \vee v$             the left, apply the or, and send their value downwards
        if row < $\sqrt{p}-1$ then
            $send(id+\sqrt{p},\ v)$

    else if row > 0 then                        The processors in the rows greater than 0
        $receive(id+1,\ u)$                     receive the value from the right, apply the or
        $v \leftarrow u \vee v$                 and send left what was received
        if $column$ > 0 then
            $send(id-1,\ u)$

    return $v$
```

2.2 Correctness

For correctness first we focus in each row. Following the serial communication fashion we can see that by definition after the first step every row will have the *ScanOr* of that particular row. The row = 0 will be already correct now. Using the notation v'_i for the intermediate results and v_i for the original values of each processor i.

$$r_i = \lfloor i/\sqrt{p} \rfloor \tag{2.1}$$
$$v'_i = \vee\{v_j : r_i\sqrt{p} \leq j \leq i\} \tag{2.2}$$

Now, for every row > 0, we have to check if there is any t above. If there is then all the rows below will have to be filled with t. By cascading the t through the last row and at the same time propagating them to the left, we are guaranteeing this condition. Note that the row left to right propagation is different from the right to left. In the former, we send the accumulated or, namely the v'_i value, whilst in the latter we send the received value u_i. Otherwise, we could propagate t values backwards getting a wrong result. Namely as we can see in the Equation 2.3, the final results v''_i are what we wanted to calculate.

$$r_i = \lfloor i/\sqrt{p} \rfloor \tag{2.3}$$
$$u_i = \vee\{v'_{(j+1)\cdot\sqrt{p}-1} : 0 \leq j < r_i\} = \vee\{v_j : 0 \leq j < r_i\sqrt{p}\} \tag{2.4}$$
$$v''_i = v'_i \vee u_i = \vee\{v_j : 0 \leq j \leq i\} \tag{2.5}$$

$\square$

2.3 Running time

Finally, as we can see from the Algorithm 2, there are no loops. Assignments, comparisons and *send* operations are $\Theta(1)$ so the most majority of the time happens during the *receive* operations since they are *blocking* operations.

As we reasoned before, for a single row ScanOr it will take time $\Theta(\sqrt{p})$ since the value has to cross all the processors in that row. The last column propagation will also take $\Theta(\sqrt{p})$. Therefore by adding the two parallel row propagations and the column propagation we get that the parallel algorithm runs in $\Theta(\sqrt{p})$ time using p processors.