Noah Wiley
CSCI 176 - Parallel Processing
Professor Park
02/10/26

CSCI 176 - Homework 1

Question 1 - Devise formulas for my_first_i & my_last_i

my_first_i & my_last_i formulas should ensure that each processor is assigned roughly the same number of data elements, handling cases where the total number of elements n is not evenly divisible by the number of processors p.

The base number of elements per core, when n is divided by p, is [n/p]. The remainder, which needs to be distributed among the first few cores, is n (mod p).

Cores with a my_rank less than the remainder will be assigned one extra element to ensure balanced distribution.

The starting index for the current core is the base number of elements multiplied by its rank, plus an offset for the extra element assigned to all preceding cores.

my_first _i = (my_rank * [n/p] + min(my_rank, n (mod p))

The ending index is the start index of the next core

my_last_i = ((my_rank + 1) * [n/p]) + min(my_rank + 1, n (mod p))

The formulas for my_first_i and my_last_i in pseudocode are;

```
// Assume n, p, and my_rank are defined and available to each core

base_elements = floor(n / p);
Remainder = n % p;

if (my_rank < remainder) {
        my_first_i = my_rank * (base_elements + 1);
        my_last_i = my_first_i + (base_elements + 1);
} else {
        my_first_i = my_rank * base_elements + remainder;
        my_last_i = my_first_i + base elements'
}
```

Question 2 - Formula for the load on each processor with cyclic assignment

The formula to compute the load on each processor with cyclic assignment involves summing the times for specific data elements assigned to that processor.
The time for index i = k is (k + 1) * T clocks.

Under cyclic assignment with p processors, a processor with rank r is assigned all data elements with indices i such that i = r (mod p0. These indices are:

$$i \in \{r, r + p, r + 2p, \ldots, r + kp, \ldots\}$$

subject to the constraint i < n.

The problem states that the computation time for index i is:

$$Cost(i) = (i + 1) * T$$

The total load L_r for processor r is the sum of costs for all its assigned indices. Let k_{max} be the largest integer such that r + k_{max} p < n. This value is $k_{max} = \left[\frac{n-1-r}{r}\right]$.

The formula for the total load (in clocks) is:

$$L_r = \sum_{k=0}^{\left[\frac{n-1-r}{p}\right]} (r + kp + 1) * T$$

By factoring out T and using the arithmetic series formula, the load can be expressed more directly:

$$L_r = T * \left[(r + 1)(k_{max} + 1) + p\frac{k_{max}(k_{max}+1)}{2}\right]$$

Where $k_{max} = [\frac{n-1-r}{p}]$

The total load on processor r is:

$$L_r = \sum_{k=0}^{\left[\frac{n-1-r}{p}\right]} (r + kp + 1)$$

Question 3 - Pseudo code for tree structured global sum

The pseudo code uses a loop structure where processors communicate in pairs, doubling the communication distance (core_difference) and testing condition (divisor) in each iteration, until only one processor holds the final sum.

```
function tree_structured_global_sum(my_sum, my_rank, p):
        divisor= 2
        core _difference = 1

        while (divisor ≤ p) {
                if  (my_rank % divisor == 0) {
                        // receiver
                        partner = my_rank + core_difference
                        receive(partner, temp)
                        my_sum = my_sum + temp
                }
                else if (my_rank % divisor == core difference) {
                        // sender
                        partner = my_rank - core_difference
                        send(partner, my_sum)
                        break
                }

                divisor =  divisor * 2
                core_difference = core_difference * 2
        }
```

In each round, half the processors send
Half receive and accumulate
After $log_2(p)$ steps, core 0 has total sum

Time complexity: O(log p)

Question 4 - What if p is NOT a Power of Two?

If the tree-structured reduction pseudocode from Question 3 is executed when the number of processors is not a power of two, some processors may attempt to communicate with a partner whose rank does not exist. Since the original algorithm assumes perfect pairing at every stage, a processor might try to send to or receive from a core with a rank greater than or equal to p, which would cause incorrect behavior or deadlock. To fix this, we simply add a condition to check whether the computed partner rank is valid before performing send or receive operations. This small modification allows the algorithm to work correctly for any number of processors.

```
divisor = 2
core_difference = 1

while (core_difference < p) {
        if (my_rank % divisor == 0 {
                partner = my_rank + core_difference

                if (partner < p) {
                        receiver(partner, temp)
                        my_sum = my_sum + temp
                }
        }
        else if (my_rank % divisor == core_difference) {
                partner = my_rank - core_difference
                if (partner >= 0) {
                        send(partner, my_sum)
                }
                Break
        }

        divisor = divisor * 2
        core_difference = core_difference * 2
}
```

Question 5 - Is Tree-Structured Global Sum Data or Task Parallelism?

The Tree-Structured Global Sum is an example of task parallelism rather than data parallelism. In this phase, processors are not performing the same operation on different pieces of data at the same time. Instead, different processors take on different roles during each stage of the reduction process. The processors send their partial sums, while others receive values and add them to their own sums. These roles change from one iteration to the next as the divisor and core difference increase. Because the processors are coordinating distinct tasks rather than uniformly applying the same computation across independent data elements, this reduction phase is best classified as task parallelism.