ITCS 6145

Parallel Computing
Professor : Jinzhen Wang

Name : Aravinda Reddy Gangalakunta
Student Id : 801393674

In Class Quiz : Extracting Parallelism

Prefix Sum :

Pre-calculates the sum of all parts of the list, so you don't have to add up numbers again and again. With the prefix sum array, you can find the sum of any section of the list quickly by subtracting two values from this helper list.

For example of list [2, 4, 1, 3, 5], we calculate:

- The first entry is just the first number: 2.
- The second entry is the sum of the first two numbers: 2 + 4 = 6.
- The third entry is the sum of the first three numbers: 2 + 4 + 1 = 7.
- The fourth entry is the sum of the first four numbers: 2 + 4 + 1 + 3 = 10.
- The fifth entry is the sum of all five numbers: 2 + 4 + 1 + 3 + 5 = 15.

So, the prefix sum array is: [2,6,7,10,15]

For example, to find the sum of numbers from the 2nd to the 4th position in the original list:

- Look at the prefix sum array: 10 (which is the sum of the first 4 numbers) and 6 (which is the sum of the first 2 numbers).
- Subtract the two values: 10 - 6 = 4. This tells you the sum from the 2nd to the 4th number is 4.

```
void computePrefixSum(int inputArr[], int lengthOfArray, int prefixArr[])
{
            prefixArr[0] = inputArr[0];
            for(int i = 0;i<lengthOfArray;i++) {
             prefixArr[i] = prefixArr[ i-1] + inputArr[i] ;
            }
}
```

1) Finding Dependencies:

   In the above sequential code the value of prefixArr[i] depends on previous value i.e. prefixArr[i-1], creating a chain of dependencies.

2) Work

   Work refers to the total amount of computational work done.

   For each element in inputArr[], you perform one addition. Therefore, if the array has n elements, the total work done is n-1 additions (since the first element is copied without computation).

   Work : **O(n-1)**

3) Width

   The width of this algorithm is 1, as each iteration of the loop depends on the previous iteration's result. This sequential dependency limits parallelism in the current form.

   Width : 1

4) Critical Path

   In this case, the critical path is equal to the length of the array because each step prefixArr[i] depends on the previous step prefixArr[i-1].

   If there are n elements in the array, the critical path length is n, meaning it takes n sequential steps to complete the computation.

   Critical path Length : n

5) Parallelized code

- Divide the array into chunks, one per thread.
- Compute partial prefix sums for each chunk in parallel (first loop).
- Compute cumulative sums by adding the partial sums of previous chunks to each chunk (second loop).

Detailed Steps

Step 1: Divide the Array into Blocks

1. Divide the input array into `numBlocks` equal-sized blocks.
2. Assign each block to a separate thread or processor.

Step 2: Compute Prefix Sums Within Each Block

1. Each thread computes the prefix sum of its assigned block independently.
2. Store the prefix sums in the corresponding positions of the output array.

Step 3: Compute the Sum of Each Block

1. Obtain the total sum of each block (the last element of each block's prefix sum).
2. Store these block sums in a separate array (`blockSums`).

Step 4: Compute Prefix Sums of Block Totals

1. Compute the prefix sums of the `blockSums` array.
2. This step can be done sequentially.

Step 5: Adjust the Prefix Sums in Each Block

1. For each block, starting from the second block:
   a. Add the total sum of all previous blocks to each element in the current block.
   b. Update the prefix sums in the output array.

```
void compute() {
            if (end - start <= THRESHOLD) {
                // Computing sequentially
                if (start < end) {
                    prefixArr[start] = inputArr[start];
                    for (int i = start + 1; i < end; i++) {
                        prefixArr[i] = prefixArr[i - 1] + inputArr[i];
                    }
                    sum = prefixArr[end - 1];
                }
            } else {
                // Split task
                int mid = (start + end) / 2;
                PrefixSumTask leftTask = new PrefixSumTask(inputArr,
prefixArr, start, mid);
                PrefixSumTask rightTask = new PrefixSumTask(inputArr,
prefixArr, mid, end);

                invokeAll(leftTask, rightTask);

                // Add back to right half
                for (int i = mid; i < end; i++) {
                    prefixArr[i] += leftTask.sum;
                }
                sum = leftTask.sum + rightTask.sum;
            }
```

```
        }
    }
```

Above is the Pseudo code for splitting tasks and submitting to fork join pool in java .