home work 2

Shayan Shabanzadeh

1. Problem Summary

The task was to:

1. Use the provided serial prefix sum implementation as a baseline.
2. Implement two parallel versions using OpenMP:
   - An $O(N \log N)$ parallel scan (Blelloch-style upsweep/downsweep).
   - An $O(N)$ parallel scan (work-efficient two–phase prefix).
3. Experiment with different thread counts and evaluate performance.

2. Serial Prefix Sum (Baseline)

The serial implementation performs a cumulative sum in a single pass:

$$prefix[i] = prefix[i-1] + src[i]$$

This is $O(N)$ work and is used as the ground truth to verify correctness of parallel versions.

3. Parallel Version 1: $O(N \log N)$

The first parallel implementation follows the classical Blelloch parallel scan algorithm. It consists of two phases:

- **Upsweep (reduce phase)**: builds a binary tree structure of partial sums.
- **Downsweep**: walks back down the tree and propagates prefix values.

The code iterates using exponentially increasing offsets:

```
for (int offset = 1; offset < n; offset <<= 1)
    #pragma omp parallel for
    for (int i = (offset << 1) - 1; i < n; i +
        prefix[i] += prefix[i - offset];
    }
}
```

After the upsweep, we reset the last element to zero and perform the downsweep:

```
for (int offset = n >> 1; offset >= 1; offset
    #pragma omp parallel for
    for (int i = (offset << 1) - 1; i < n; i +
        int temp = prefix[i - offset];
        prefix[i - offset] = prefix[i];
        prefix[i] += temp;
    }
}
```

Each phase touches the entire array, and since the offset doubles each iteration, there are $\log N$ total passes. Each pass processes $O(N)$ work, resulting in:

$$O(N) \cdot O(\log N) = O(N \log N)$$

This version produces correct results but suffers from synchronization overhead, because each level of the tree requires a barrier before threads can continue.

4. Parallel Version 2: $O(N)$ Work-Efficient Prefix Sum

The second parallel implementation improves efficiency by reducing synchronization and avoiding tree-based updates.

The input array is divided into contiguous chunks, one per thread:

```
int chunk = (n + num_threads - 1) / num_thread
int start = tid * chunk;
int end   = min(start + chunk, n);
```

Each thread computes a serial prefix sum on its own segment:

```
for (int i = start + 1; i < end; i++)
    prefix[i] = prefix[i - 1] + src[i];
partial_sums[tid] = prefix[end - 1];
```

After a barrier, a single thread computes the prefix sum of the per-thread totals:

```
for (int t = 1; t < num_threads; t++)
    partial_sums[t] += partial_sums[t - 1];
```

Finally, each thread adds the correct offset to its local segment:

```
int offset = partial_sums[tid - 1];
for (int i = start; i < end; i++)
    prefix[i] += offset;
```

Each element is processed exactly once, so total work remains linear:

$$O(N)$$

Due to minimal synchronization (only 2 barriers) and no repeated passes over the data, this version scales significantly better than the $O(N \log N)$ tree-based version.

5. Performance Results

We ran the program using different thread counts:

$$\{1, 2, 4, 8, 16, 32, 64, 128\}$$

Execution time plots:

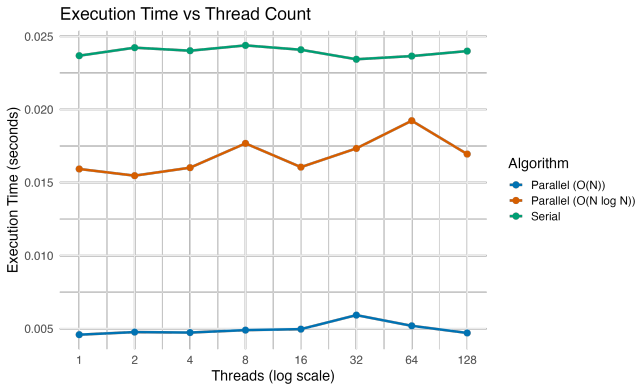

Figure 1: Execution time vs. thread count (log scale).

Speedup relative to serial:

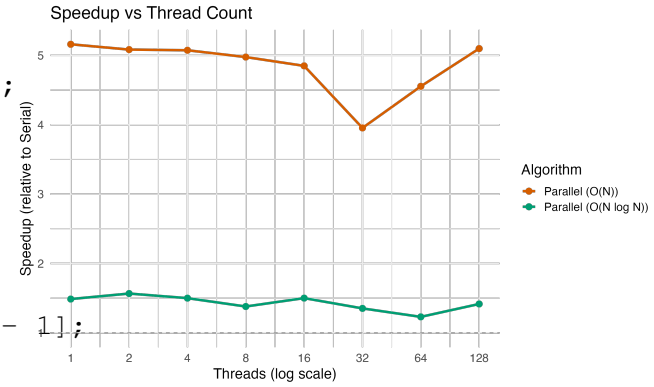

Figure 2: Speedup relative to serial prefix sum.

6. Observations and Conclusions

Based on the timing results and plots, the following observations can be made:

Performance of Parallel O(N) Algorithm

The work–efficient $O(N)$ version achieves the best performance (over 5× speedup). This is because:

- Each thread works on a contiguous block of data.
- Only two synchronization points are required (after local scans and after partial sum update).
- Very little communication occurs between threads.

This leads to excellent cache locality and low parallel overhead.

Performance of Parallel O(N log N) Algorithm

The $O(N \log N)$ version shows weaker scaling (only 1.4×–1.6× speedup). The primary reasons are:

- Each level of the tree requires a global synchronization.
- Threads repeatedly update shared memory, causing extra memory traffic.

In other words: it does more total work and spends more time waiting.

Effect of Increasing Threads

Both implementations improve initially, but beyond 16–32 threads:

- Speedup levels off or gets worse.
- Memory bandwidth and scheduling overhead become bottlenecks.

Although both algorithms are parallel, their performance is very different: