

COMS3008A Assignment – Report

Claudio Da Mata - 2128358

23-10-2022

1 Problem 1: Parallel Scan

- Given a set of elements, $[a_0, a_1, \dots, a_{n-1}]$, the scan operation associated with addition operator for this input is the output set $[a_0, (a_0 + a_1), \dots, (a_0 + a_1 + \dots + a_{n-1})]$.
- For example, the input set is $[2, 1, 4, 0, 3, 7, 6, 3]$, then the scan with addition operator of this input is $[2, 3, 7, 7, 10, 17, 23, 26]$.

1.1 Serial Implementation:

Firstly I started off with the baseline implementation of serial scan operation given in Listing 1

```
1 void scan(int out[], int in[], int N){
2     out[0] = in[0];
3
4     for(int i=1; i<N; i++) {
5         out[i] = in[i] + out[i-1];
6     }
7 }
```

Listing 1: Sequential algorithm for computing scan operation with '+' operator¹.

This got me a baseline average time to measure and compare parallel implementations. I ran the operation on arrays ranging in sizes from 1000 to 50,000 with intervals of 1000 given in Figure 1

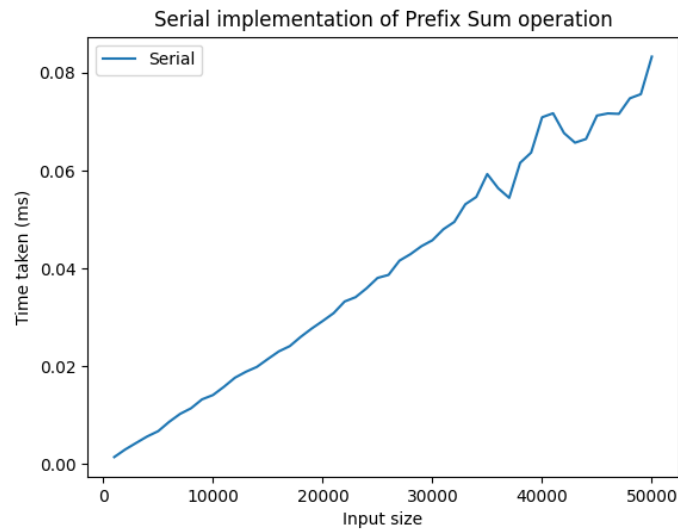


Figure 1: Serial scan operation average times

1.2 OpenMP Parallel Implementation:

Next I implented the scan operation within OpenMP's interface. The algorithm I used is based off the Blelloch² work-effecient algorithm given in Listing 2

```

1 void scan(int out[], int in[], int N){
2     int nthr, *z, *x = out;
3     #pragma omp parallel num_threads(4)
4     {
5         int i;
6         #pragma omp single
7         {
8             nthr = omp_get_num_threads();
9             z = malloc(sizeof(int)*nthr+1);
10            z[0] = 0;
11        }

12
13        int tid = omp_get_thread_num();
14        int sum = 0;
15        #pragma omp for schedule(static)
16        for(i=0; i<N; i++) {
17            sum += in[i];
18            x[i] = sum;
19        }

20
21        z[tid+1] = sum;
22        #pragma omp barrier

```

```

24     int offset = 0;
25     for(i=0; i<(tid+1); i++) {
26         offset += z[i];
27     }

29     #pragma omp for schedule(static)
30     for(i=0; i<N; i++){
31         x[i] += offset;
32     }
33 }
34 free(z);
35 }

```

Listing 2: OpenMP Parallel algorithm for computing scan operation³.

The algorithm works as follows:

- Start omp parallel
- Initiate a 'single' construct which allows only one thread to run the code
- Inside single should declare the size of the temp array and set $element[0] = 0$
- After the single construct, get current threadID and initialize $sum = 0$
- Begin an omp for construct with schedule(static)
- Inside the for loop (from 0 to $N - 1$), $sum +=$ the input array at i , and set the output array at i equal to the new sum
- After the for loop, set temp array at $[threadID + 1]$ equal to sum
- Declare a Barrier construct which forces all threads to wait until other threads finish computation
- Set $offset = 0$
- Begin a regular for loop that sums all elements of the temp array to $offset$, only adding elements that each specific thread has computed itself
- Begin an omp for construct with schedule(static) again, from $i = 0$ to $N - 1$
- sum output array element i with offset
- Finish off pragma omp and then free the temp array from memory

I experimented with different number of threads running and managed to narrow it down to thread counts of 2, 4, 6 given in Figure 2

With 2 threads running there is no noticeable speedup compared to serial implementation. With 4 and 6 threads running there is significant speedup compared to serial implementation. While input size n increases, 4 threads performs slightly better than 6 threads. Thus I choose 4 threads as my optimal implementation of OpenMP parallelization.

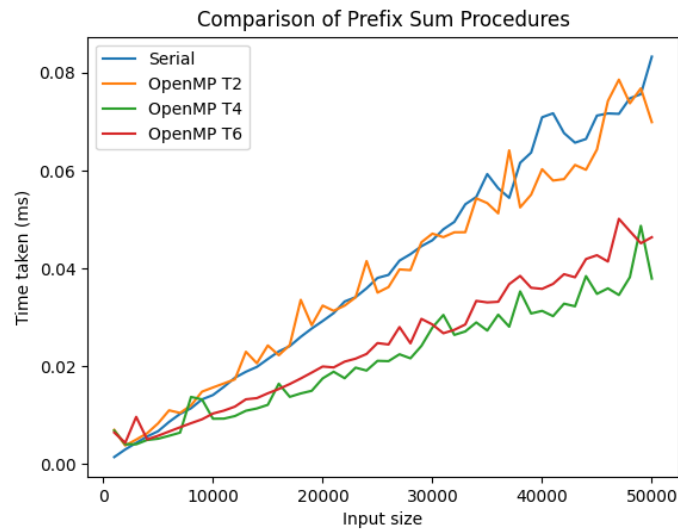


Figure 2: OpenMP Parallel scan operation average times for different thread counts

1.3 MPI Parallel Implementation:

Next I implemented the scan operation within MPI's interface. The algorithm I used is based off the Blelloch² prescan algorithm given in Listing 3

```

1  MPI_Init(&argc, &argv); //Initialize MPI
3  MPI_Comm_rank(MPI_COMM_WORLD, &my_rank);
4  MPI_Comm_size(MPI_COMM_WORLD, &comm_size);

6  MPI_Barrier(MPI_COMM_WORLD);

8  size_t num_per_proc = n / comm_size;
9  int sum;

11 start_find_sum(my_rank, comm_size, data, num_per_proc,
12 &sum);
13 start_find_psum(my_rank, comm_size, data, num_per_proc,
14 sum);

16 MPI_Barrier(MPI_COMM_WORLD);

18 MPI_Finalize(); //Close MPI

```

Listing 3: MPI Parallel algorithm for computing scan operation⁴.

The essence of the algorithm comes from the `start_find_sum()` and `start_find_psum()` functions.

The algorithm works as follows:

- Start MPI
- Get comm rank and size
- Barrier that forces all threads to wait
- Set num per proc equal to array size $n/commsize$. This dictates how many numbers each thread will compute the sums of
- Call start_find_sum() function
 - The algorithm works like a binary tree with levels
 - Gets MPI_Status
 - Sums input array elements from 0 to *numPerProc*
 - Begin for loop from $level = 0$ to $\log_2(commsize)$
 - Inside the loop set $position = rank/level^2$
 - If *position* is even, receives the *sendersSum* (with $senderRank = rank + level^2$) and adds to total *sum*
 - If *position* is odd, sends total *sum* to receiver with $senderRank = rank - level^2$, then kills current thread
 - Before for loop ends, calls Barrier so all threads wait
- Call start_find_psum() function
 - The algorithm works like a binary tree with levels
 - Gets MPI_Status
 - If $rank == 0$, sets $psum = sum$
 - Begin for loop from $level = \log_2(commsize) - 1$ down to 0
 - If process is on current level:
 - Set $position = rank/level^2$
 - If *position* is even this means this process was the parent of *sendingRank*, and so first sets $senderRank = rank + level^2$, then sends *psum*, then receives the *senderSum* and finally subtracts *psum* by *senderSum*
 - If *position* is odd, sets $receivingRank = rank - level^2$, then receives *psum* from parent, and then sends *sum* with *receivingRank*
 - Before for loop ends, calls Barrier so all threads wait
 - After the loop ends, put the *prefixSums* associated with this process in the input array
- Calls Barrier so all threads wait
- Ends MPI

With start_find_sum() and start_find_psum() functions given respectively by Listing 4 and Listing 5

```

1 void start_find_sum(int rank, int mysize, int* in,
2                     size_t num_per_proc, int* overall_sum){
3     MPI_Status status;

4
5     int sum = find_sum(in, num_per_proc);

6
7     int still_alive = 1;
8     int level;

9
10    for (level = 0; level < (int)log2(mysize); level++) {
11        if (still_alive) {
12            int position = rank / (int)pow(2, level);

13
14            if (position % 2 == 0) {
15                // I am a receiver
16                int sender_sum;
17                int sending_rank = rank + (int)pow(2, level);

18
19                MPI_Recv(&sender_sum, 1, MPI_INT, sending_rank,
20                        0, MPI_COMM_WORLD, &status);

21
22                sum += sender_sum;
23            }else {
24                // I am a sender
25                int receiving_rank = rank - (int)pow(2, level);

26
27                MPI_Send(&sum, 1, MPI_INT, receiving_rank, 0,
28                        MPI_COMM_WORLD);
29                still_alive = 0;
30            }
31        }
32        MPI_Barrier(MPI_COMM_WORLD);
33    }
34    *overall_sum = sum;
35 }

```

Listing 4: start_find_sum()

```

1 void start_find_psum(int rank, int mysize, int* in,
2                     size_t num_per_proc, int sum){
3     int psum;
4     int level;
5     MPI_Status status;

6
7     if (rank == 0) {
8         psum = sum;
9     }
10    for (level = (int)log2(mysize) - 1; level >= 0; level--) {
11        // only trigger the processes on the current level

```

```

12     if (level == 0 || rank % (int)pow(2, level) == 0) {
13         int position = rank / (int)pow(2, level);

15         if (position % 2 == 0) {
16             int sender_sum;
17             int sending_rank = rank + (int)pow(2, level);

19             MPI_Send(&psum, 1, MPI_INT,
20                     sending_rank, // RIGHT CHILD
21                     0, MPI_COMM_WORLD);

23             MPI_Recv(&sender_sum, 1, MPI_INT,
24                      sending_rank, 0, MPI_COMM_WORLD, &status);

26             // psum <- (prefix sum of parent) - (sum of sibling)
27             psum -= sender_sum;
28         }else{
29             int receiving_rank = rank - (int)pow(2, level);

31             MPI_Recv(&psum, 1, MPI_INT,
32                     receiving_rank, // PARENT
33                     0, MPI_COMM_WORLD, &status);

35             // send sum to receiving_rank so it can fix its sum
36             MPI_Send(&sum, 1, MPI_INT,
37                     receiving_rank, 0, MPI_COMM_WORLD);
38         }
39     }
40     MPI_Barrier(MPI_COMM_WORLD);
41 }
42 // put the prefix sums associated with this node in input array
43 int next_sum = in[num_per_proc-1];
44 in[num_per_proc-1] = psum;

46 for (int j = num_per_proc - 2; j >= 0; j--) {
47     int next_sum_tmp = in[j];

49     in[j] = in[j+1] - next_sum;

51     next_sum = next_sum_tmp;
52 }
53 }

```

Listing 5: start_find_psum()

I experimented with different number of threads running and managed to narrow it down to thread counts of 2,4,8 given in Figure 3

With 2 threads running there is noticeably worse running time compared to serial implementation. With 4 and 8 threads running there is significant speedup compared to serial implementation. With smaller input sizes n , 4 threads performs

better than 8 threads, but as n increases, performance between 4 threads and 8 threads stays relatively the same. Thus I choose 4 threads as my optimal implementation of MPI parallelization.

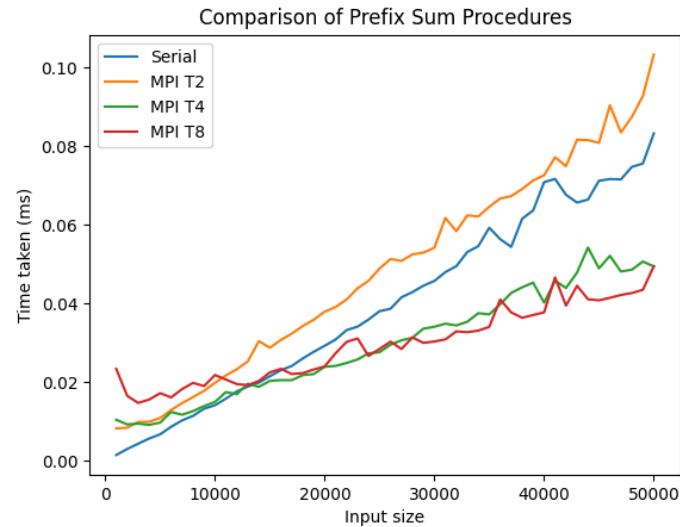


Figure 3: MPI Parallel scan operation average times for different thread counts

1.4 Validation:

Checking if elements are correctly summed, through running *run.sh* given in Figure 4

```
-----Serial scan is starting-----
Validate passed.
-----Serial scan is done-----

-----OpenMP scan is starting-----
Validate passed.
-----OpenMP scan is done-----

-----MPI scan is starting-----
Validate passed.
-----MPI scan is done-----
```

Figure 4: Validation of scan operations

1.5 Conclusions:

After comparing the results from serial, optimal OpenMP (with 4 threads) and optimal MPI (with 4 threads), I can make the conclusion that serial performs the best at very small array size n . While at large n , both OpenMP and MPI have significant speedup from serial, OpenMP performs consistently better than MPI. Comparisons given in Figure 5

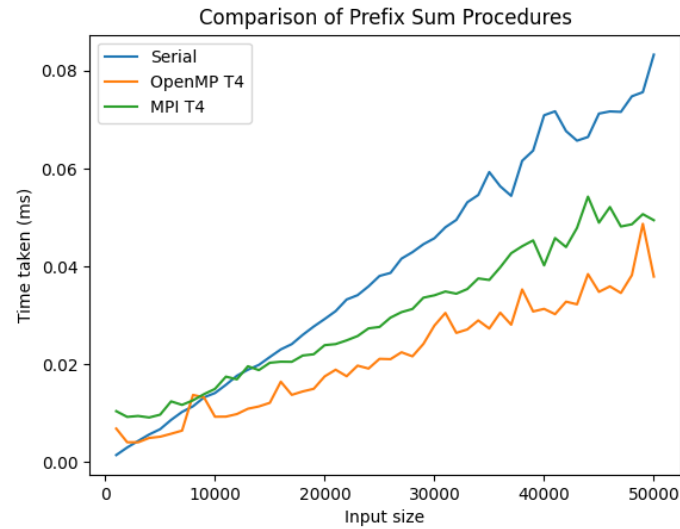


Figure 5: Comparison of all scan operations

2 Problem 2: Parallel Bitonic Sort

Not applicable, working alone.

3 Problem 3: Parallel Graph Algorithm

An example figure is given in Figure 6.

An example of table is given Table 1.

Table 1: An example of a table					
No of vertices	64	128	256	384	512
Serial	0.1	0.2	0.3	0.4	0.5
Parallel					
Speedup	2	3	4	5	6

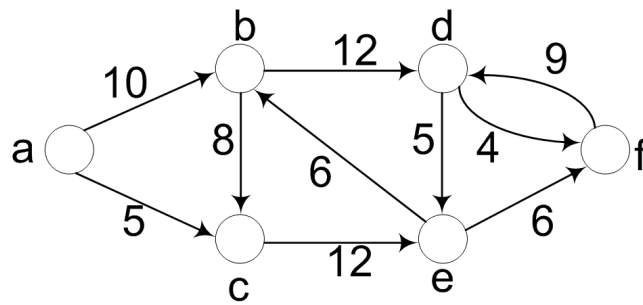


Figure 6: A directed graph

References

- [1] Prefix Sum Array - implementation and applications in competitive programming, <https://www.geeksforgeeks.org/prefix-sum-array-implementation-applications-competitive-programming/>, accessed: 23-10-2022.
- [2] Prefix Sum - algorithm 2: Work-efficient, https://en.wikipedia.org/wiki/Prefix_sum, accessed: 23-10-2022.
- [3] CSE 231 Introduction to Parallel and Concurrent Programming at Washington University - scan, <https://classes.engineering.wustl.edu/cse231/core/index.php/Scan>, accessed: 23-10-2022.
- [4] MPI Parallel Prefix Sum (1A), <https://upload.wikimedia.org/wikiversity/en/2/2f/ParaPrefix.MPI.1.A.20140730.pdf>, accessed: 23-10-2022.