School of Computer Science and Applied Mathematics
UNIVERSITY OF THE WITWATERSRAND, JOHANNESBURG

COMS3008A Assignment – Report

Sahil Dinanath, Darshan Singh

5 June 2023

1 Problem 1: Parallel Scan

Given a set of elements, $[a_0, a_1, \cdots, a_{n-1}]$, the scan operation associated with addition operator for this input is the output set $[a_0, (a_0 + a_1), \cdots, (a_0 + a_1 + \cdots + a_{n-1})]$. 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

strategy:

The approach that was chosen to represent the serial implementation of the prefix scan algorithm was the serial workefficient implementation of blelloch's algorithm which is shown in Listing 1.

```
upsweep(inputArray,n):
  for d = 0 to log2(n) - 1 do
      for k = 0 to n - 1 by 2^(d+1) do
          inputArray[k + 2^(d+1) - 1] += inputArray[k + 2^d - 1]

downsweep(inputArray,n):
  input[n-1] = 0
  for d = log2(n) - 1 down to 0 do
    for k = 0 to n - 1 by 2^(d+1)do
      temp = inputArray[k + 2^d - 1]
      inputArray[k + 2^d - 1] = inputArray[k + 2^(d+1) - 1]
      inputArray[k + 2^(d+1) - 1] += temp
```
Listing 1: upsweep and downsweep pseudocode

The idea as stated in Parallel Prefix Sum with Cuda is to build a balanced binary tree with n leaves and log2(n) levels called depths.

1.2 OpenMP

Parallelisation method:

The serial implementation of blelloch's algorithm was used as the base for the OMP parallel implementation, with largely the core algorithm staying the same. However, the differences being with how the algorithm processed a single array. The first most obvious parallisation method is to parallise the inner for loops in both the upsweep and downsweep of the algorithm, see Listing 2

```
...
    for k = 0 to n - 1 by 2^(d+1) in parallel do
...
```

Listing 2: parallisation of the inner loop

this allows for the summations at each depth to be parallized. The issue with this method is that at each level the amount of elements that needs to be computed is halved, hence the algorithm becomes more serial as it progresses as there is less elements to split between the cores for both the up and down sweep.

The second Parallelisation strategy was to take the input array, break it up into chunks and have each processor perform it's own up and down sweep on it's section of the array. This would expose more concurrency as at each depth all the processors can be acting on their chunk of the array, in contrast to the original method where at the highest depth only 1 processor will be able to act.
This was achieved using a processor sum, which is the summation of each chunk given to each processor. Each processor then perform the up and down sweep on their respective chunk. Finally each processor adds their processor sum to each element of their array. eg: $[a_0, (a_0 + a_1), (a_0 + a_1 + a_3), (a_0 + a_1 + a_3 + a_4)]$

assume 2 processors hence 2 chunks:
$[a_0, (a_0 + a_1), (a_0 + a_1 + a_3)]$,
$[(a_0 + a_1 + a_3 + a_4), \ldots]$

let $processorSum_0 = a_0 + a_1 + a_3$ therefore
$[(processorSum_0 + a_4), (processorSum_0 + a_4 + a_5), \ldots]$

hence for each$[(a_4), (a_4 + a_5), \ldots]$ add $processorSum_0$
as you can see now each chunk is independant of eachother meaning we can now perform the up and down sweep independantly and concurrently on each processor. See Listing 3

```
getProcessSum(inputArray, size, processSum, chunkSize,rank){
  if (rank == 0):
    processSum = 0
    return
  start = (rank-1) * chunkSize
  end = rank * chunkSize
  for i = start to end-1 by i+1 in parallel do
    processSum += input[i];
```

Listing 3: Serial Sequenctial Scan Algorithm with + operator

1.3 MPI

Parallelisation Strategy:

All the parallisation stragegies that were used in the OMP algorithm was used for MPI with the differences being process wise, but the general concept being the same. Some of the differences include:

1. instead of a shared input array, each MPI process has their own copy.
2. Using ranks each process works on their chunk of the array independantly although they have the entire array.
3. Finally using *MPI_Gather* function, the chunks that they worked on are gathered into an array on the process with rank 0.

Although this method does require more memory it allows us to remove the Multiple broadcasts, as each process would need to communicate it's process sum to each node.

1.4 Correctness

The correctness of all prefix sum algorithm above was determined by a simple sequential serial algorithm for computing scan operations shown in Listing 4

```
prefixSum(inputArray, size)
  for i = 1 to size-1 by i+1
    input[i] += input[i - 1]
```

Listing 4: Serial Sequenctial Scan Algorithm with + operator

The above algorithm is easily testable itself, simple enough implementation to ensure algorithmic correctness, and correct in all problem sizes with testing speeds being acceptable.
the steps that were taken to implement this were:

1. Create two arrays, let us call one the original array and the other the input array. The original array will contain the original set of elements and the input array will be modified using the above algorithms.
2. When reading/generating elements assign them both to each array. So that both arrays are identical.
3. perform the serial/parallel algorithm that are to be tested on the input array. This will modify the array and result in a final answer.
4. When this is done perform the serial sequential scan algorithm above on the original array.
5. Each element of each array is then compaired to see if they match.

Correctness of the testing algorithm was also ensured by reading in determinate data from files which solution is known, then comparing the output to what was expected. This ensures the correctness check is correct, which ensures that the other algorithms would be correct.

1.5 Testing

Each algorithm including the correctness were tested in two different ways. First method was determinate by reading unchanging data from files. The second was generating random numbers. The second approach allowed us to scale the testing input to significant sizes. A bash script was used allow the efficient testing of the algorithms which were designed to take variables such as threads and problem sizes as parameters. Variables could be used to change the arguments passed as parameters. The compiled algorithms would be run multiple times using a for loop in the bash script. The output for the time taken to complete a run were written to a text file where a python file performed calculations of the averages of the runs. The Bash script also allows us to run each algorithm with identical inputs to see the performance differences accurately.

1.6 performance evaluation

Table 1: Blelloch's prefix scan with different input sizes using 2 cores

No of elements	2^3	2^{18}	2^{26}	2^{28}
Serial	0.000002	0.004051	0.728403	2.896867
OMP	0.000032	0.003475	0.563001	2.242173
Speedup	0.063492	1.165588	1.293789	1.291991
MPI	0.000020	0.003168	0.559897	2.241902
Speedup	0.099999	1.278422	1.300958	1.292147

As seen above the average speed up given a problem size of decent size is 1.25x for OMP and 1.324x for MPI using 2 cores. As can be seen above the performance with OMP using more cores can be seen to gain a considerable speed up

Table 2: Blelloch's prefix scan with different input sizes using 4 cores

No of elements	2^3	2^{18}	2^{26}	2^{28}
Serial	0.000002	0.001549	0.719261	3.0204775
OMP	0.000064	1.001434	0.371427	1.4844075
Speedup	0.035294	1.079658	1.936479	2.0348034
MPI	0.000062	0.001851	0.707955	3.755351
Speedup	0.036144	0.836462	1.015970	0.804313

over it's other linear counter part by 2x. Due to the implementation of MPI we can see the performance decrease due to the lack of memory availiable to the system as MPI the implementation creates n copies of the input array (n being the amount of processors).

Table 3: Blelloch's prefix scan with different input sizes using 8 cores

No of elements	2^3	2^{18}	2^{26}	2^{28}
Serial	0.000001	0.001538	0.734418	2.9164853
OMP	0.000164	0.000825	0.367877	1.4725225
Speedup	0.009146	1.873364	1.996371	1.9806049
MPI	0.009259	0.000441	0.353135	4.1406010
Speedup	0.036144	3.492622	2.079710	0.7043629

Although OMP can be seen to not have a large benefit on this problem size on 8 cores, with it performing on par or slightly better then then it's 4 core counterpart. The MPI version can be seen to have a huge increase in performance on the 2^{26} and 2^{28} problem sizes. This improvement is due to the increase in cores since 8 cores allows the MPI to overcome the slow down due to the additional computation, and also the problem size fitting more comfortably in memory and cache.

2 Problem 2: Parallel Bitonic Sort

Approach

2.1 Serial

2.1.1 Algorithm

The main code that implements the Bitonic Sort algorithm is shown in Listing 5.

```
// Code snippet of the Bitonic Sort algorithm
// Function definitions for compAndSwap, bitonicMerge, and bitonicSort

int main(int argc, char *argv[]) {
  // Initialization and setup code
  // Read input file and convert characters to integers
  // Start timing
  // Call bitonicSort to sort the input array
  // Stop timing
  // Print sorted array
}
```

Listing 5: Bitonic Sort Algorithm

The sorting algorithm used in this test is the iterative bitonic sort. It consists of two main functions: `iterativeBitonicSort` and `compareArrays`.

The `iterativeBitonicSort` function implements the iterative bitonic sort algorithm, which performs a series of comparisons and swaps on the input array to sort it in ascending order.

The `compareArrays` function compares two arrays element by element and returns a result indicating whether the arrays are identical or not.

Correctness Assertion

To test the correctness of the sorting algorithm, we perform the following steps:

1. Read in random input and store it in an array.
2. Create a copy of the input array using the `arrayCopy` function.
3. Call the `imperativeBitonicSort` function to sort the copied array.
4. Compare the sorted array with the original input array using the `compareArrays` function.
5. If the result is `1`, the arrays are identical, and the sorting algorithm is correct. Otherwise, if the result is `0`, the arrays differ, indicating an error in the sorting process.

Example 1 with 2^8

Input array: [4 5 6 7 3 2 1 0]

Sorted array: [0 1 2 3 4 5 6 7]

Correctness assertion: [0 1 2 3 4 5 6 7]

Result: Correct

Testing

The code begins by reading the input file and converting the characters to integers.

- `getFileSize`: This function determines the size of the input file by seeking to the end of the file, retrieving the current position (which represents the file size), and then resetting the file position to the beginning.
- `readFile`: This function reads the contents of the input file into a character array `line` using the `fgets` function. It also closes the file after reading.
- `convertCharToIntArray`: This function converts the character array `fileCharacters` into an array of long integers `input` by subtracting the ASCII value of `'0'` from each character.

2.2 OpenMP

Parallelisation

My approach with parallelization was to strike a balance between workload distribution, task creation overhead, and task synchronization given that this was initially a recursive implementation.

1. Task Granularity: The granularity of tasks in the code may not be optimal. Creating tasks for each recursive call (‘bitonicSort‘ and ‘bitonicMerge‘) can result in a large number of small tasks, which may introduce overhead due to task creation and management. It is generally recommended to balance the workload of tasks and minimize task creation overhead.
2. Workload Imbalance: The workload distribution among tasks may not be even. In the ‘bitonicSort‘ function, the division of the array into two parts (‘k‘ and ‘count - k‘) may not always result in equal-sized subarrays. This can lead to workload imbalance among tasks and underutilization of available threads.
3. Task Synchronization: I had considered using `#pragma omp taskwait` to synchronize tasks. While this ensures that all tasks complete before proceeding, it introduces synchronization overhead.

My approach therfore had to change to an iterative bitonic sort due to the large overheads impose by a recursive sort.

- I firstly use `genBitonic` to determine within the parallel loop whether the range increasing or decreasing using `((i / j) % 2) == 0`

- `dir = 1` is increasing, `dir = 0` is decreasing
- it then calls `iterativeBitonicSort` with the respective direction
- `#pragma omp parallel for` is then used to parallelize the for loop

```
function genBitonic(startIndex, lastIndex, input)
    size <- lastIndex - startIndex + 1
    for j = 2 to size step j = j * 2
        parallel for i = 0 to size step i = i + j
            if (i / j) % 2 == 0 then
                iterativeBitonicSort(i, i + j - 1, 1, input)
            else
                iterativeBitonicSort(i, i + j - 1, 0, input)
            end if
        end parallel for
    end for
end function
```

2.3 MPI

Parallelisation

1. My approach with MPI was to scatter the `input` array with a size of `chunk_size` across each process.
2. run `genBitonic` then Gather each process back so the one half of processes that own the monotonic increasing is brought back with the other half of processes that own the monotonic decreasing sequence

Performance evaluation

The Bitonic Sort algorithm is then applied to the input array using the `bitonicSort` function. The execution time of the sorting process is measured using the `omp_get_wtime` function. Finally, the sorted array is printed using the `printArray` function, and the total execution time is displayed.

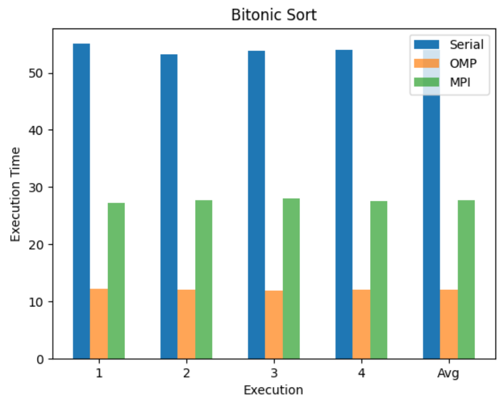
Figure 1: Bitonic sequence sort on 2 cores

Input size	2^3	2^{18}	2^{20}	2^{26}
Serial	0.000005	0.142559	0.538202	54.035102
OpenMP	0.000289	0.037189	0.134956	12.085019
Speedup	0.017256	3.833307	3.987979	4.471246
MPI	0.000073	0.068826	0.284624	27.667862
Speedup	0.068259	2.071306	1.890921	1.952991

Figure 2: Bitonic sequence sort on 4 cores

Input size	2^3	2^{18}	2^{20}	2^{26}
Serial	0.000001	0.139811	0.534981	54.402933
OpenMP	0.001198	0.035570	0.144845	12.836372
Speedup	0.000834	3.930511	3.693469	4.238186
MPI	0.218966	1.062465	1.372911	37.945344
Speedup	0.000004	0.131591	0.389669	1.433718

MPI here performs 30% worse on 4 cores compared to 2 largely due to increased overhead from having to split the data across more processes where it is much more beneficial to not split it up as much for this small input set. We can see from the jump of speedup from Figure 1, 2^3 to 2^{18} exhibits much better utilization of processes to parallelize the task. OpenMP is significantly better optimised on larger sizes due to it's iterative nature whereas the serial with pure recursion handles smaller sorts better.



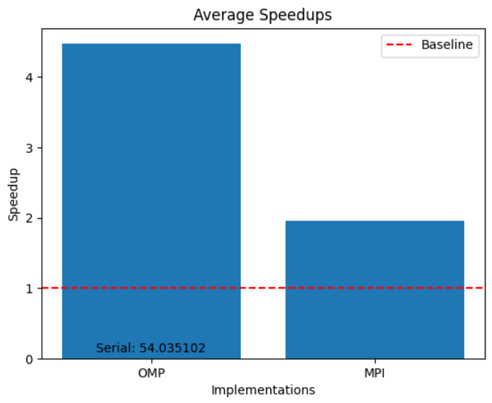


Figure 3: Bitonic sequence images

3 Problem 3: Parallel Graph Algorithm

3.1 Serial

Strategy

The algorithm that was chosen was Dijkstra's Single Source Shortest Path algorithm with a standard implementation.

3.2 OpenMP

Parallelisation Strategy

The strategy that was used to parallize Dijkstra's algorithm was by allowing each processor to visit adjacent nodes concurrently. This proved to be effective when graphs had many adjacencies however the amount of cores utilised were dependant on how many adjacencies the nodes had.

3.3 MPI

The MPI implementation follows the same parallelisation strategy as the OMP version. With the key difference being that the OMP implemention being that we take the 2d input graph and flatten it into a 1d array. This allows us to scatter the graph input graph to each process and then each perform their computation seperatly on their section of the graph. The results are then collected on the first node. An example figure is given in Figure **??**.

An example of table is given Table 2.

3.4 Correctness

To test the correctness of the algorithms we verfied that our sequential algorithm was correct by having it read in a known graph and comparing the end result with it's output.

After verifying our sequential algorithm to be correct we proceeded to use it to test the correctness of our OMP and MPI implementations. We did this by made two copies of the array, input and original. We then had each algorithm perform their method of computation on each array. Each element was then compared sequentially with eachother to ensure correctness of the final result.

3.5 Testing

Testing for this these algorithms were done using a bash script. The graph and the number of threads/processes were passed in as parameters into the executable. We ran each computation multiples times, wrote the results to a file and then computed the average speed up compared to the sequential algorithm and time. The data used to test the algorithms were given to us in text files each one larger then the last. The tests were done on laptops and on servers.

3.6 Performance Evaluation

Table 4: Results using 2 threads/processes on different problem sizes

No of elements	*graph*_2.*txt*	*graph*_8.*txt*	*graph*_10.*txt*
Serial	0.000005	0.003785	0.007159
OMP	0.000068	0.003383	0.005172
Speedup	0.076642	1.118921	1.384069
MPI	0.000039	0.005341	0.009295
Speedup	0.132911	0.708655	0.770172

As can be seen with the above results the overhead caused by parallelisation is minimised with larger problem sizes. When using 4 threads the OMP results can be seen to have a large boost in performance when computing the largest

Table 5: Results using 4 threads/processes on different problem sizes

No of elements	*graph*_2.*txt*	*graph*_8.*txt*	*graph*_10.*txt*
Serial	0.000006	0.003753	0.007254
OMP	0.000129	0.002836	0.003870
Speedup	0.042802	1.322992	1.874491
MPI	inconclusive	inconclusive	inconclusive
Speedup	inconclusive	inconclusive	inconclusive

problem size. Due to an error in our MPI implementation we could not retrieve retrieve consistent data from the MPI runs so it was decided to ommit the results from the report.

Table 6: Results using 8 threads/processes on different problem sizes

No of elements	*graph_*2*.txt*	*graph_*8*.txt*	*graph_*10*.txt*
Serial	0.000009	0.003823	0.007191
OMP	0.000312	0.003159	0.003885
Speedup	0.028823	1.210255	1.850910
MPI	inconclusive	inconclusive	inconclusive
Speedup	inconclusive	inconclusive	inconclusive

we see a small decrease in performance when moving up to 8 threads. The reason for this is due to the lack of concurrency that we could expose in our parallel implementation.