UNIVERSITY OF THE WITWATERSRAND, JOHANNESBURG

COMS3008A - Parallel Computing

BSc Computer Science

Parallel implementation of Scan and Bitonic sort

Author:

Luca von Mayer (2427051) 2427051@students.wits.ac.za

Note on plagiarism policy:

May/June 2023 – 1st Semester

Contents

1 Aim/Intent:

The aim of this experiment is to research and implement parallel versions of two fundamental algorithms: the scan (prefix sum) algorithm and the bitonic sort algorithm. The expectation is that these implementations take advantage the power of parallel computing to achieve improved performance when compared with the serial version of the aforementioned problems.

Following their development, their performance will be evaluated in terms of execution time, speedup, and comparisons with other implementations. Through this study, we will gain a deeper understanding of the principles and techniques involved in parallel algorithm design, as well as the challenges and benefits of parallel computing in the context of scan and bitonic sort algorithms.

2 Experiment details

In order to conduct this experiment and for it to be relevant, we need to test the algorithms not only against each other but over a variety of sample sizes to ensure the results are as conclusive as possible.

The general way this is done is the same for all the algorithms tested in this experiment. The code used to run each of the algorithms is surround by a loop that runs from 3 to a max value, either set by default or entered by the user. This loop index determines the sample size (array size) that will be computed with for each iteration of the loop, this is done by raising 2 to this value (2^k where k is the loop index). The reason for this is that the algorithms in this experiment, due to their use of tree structures, can only properly function with array sizes that are powers of 2.

Following this, the relevant code is timed for each loop execution using either ompgetwtime() or MPIWtime() respectively. These times are then stored along with their n size in a text file for each implementation. Finally, these text files are loaded by a python scrip so they can be graphed and interpreted using the matplotlib library.

These results are obtained on a local machine with an Intel i3 processor with 6 cores.

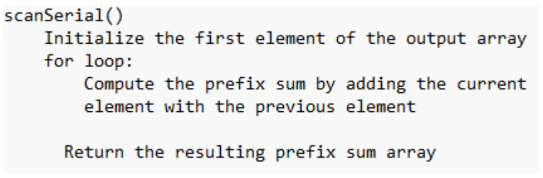
3 Parallel Scan

3.1 Describing the problem

The scan algorithm, also known as the prefix sum algorithm, is widely used in various computational tasks and as such is an important algorithm to understand and optimise. It is one of the simplest and most useful building blocks for parallel algorithms. Given a set of elements, [a0, a1, ..., an-1], the scan operation associated with the addition operator for this input is the output set [a0, (a0 + a1), ..., (a0 + a1 + ... + an-1)]. Our objective is to develop a parallel implementation of this algorithm that given an array of elements, the prefix sum can be computed using parallel techniques.

3.2 Serial Implementation

The serial implementation is developed as in the experiment brief [6].

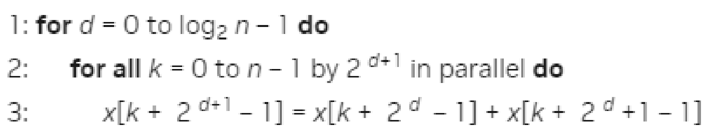


```
scanSerial()
    Initialize the first element of the output array
    for loop:
        Compute the prefix sum by adding the current
        element with the previous element

      Return the resulting prefix sum array
```

Figure 1: Pseudocode for serial scan

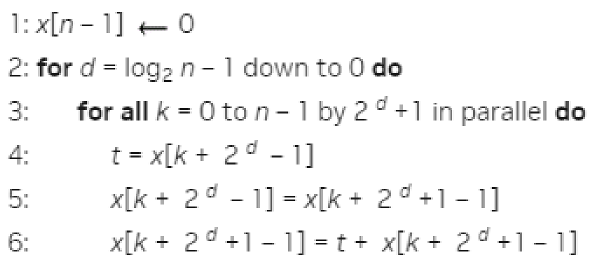
3.3 OMP parallelization method

The parallelization of the code is based on the technique described by Blelloch in 1990 [2]. The parallel implementation involves two phases: an upsweep phase and a downsweep phase, which are used to perform the prefix sum operation. In the upsweep phase, pairs of elements are summed together, allowing each processor to compute the sum of one pair. The downsweep phase utilizes the values computed during the upsweep phase to construct the pre-scan (this is an array close to the final scan but starts with 0 and is missing the final value) [2, 3]. Both of these phases are parallelized using the OMP for construct.



```
1: for $d = 0$ to $\log_2 n - 1$ do
2:     for all $k = 0$ to $n - 1$ by $2^{d+1}$ in parallel do
3:         $x[k + 2^{d+1} - 1] = x[k + 2^{d} - 1] + x[k + 2^{d}+1 - 1]$
```

(a) Figure 2.1: Upsweep



```
1: $x[n - 1] \leftarrow 0$
2: for $d = \log_2 n - 1$ down to 0 do
3:     for all $k = 0$ to $n - 1$ by $2^{d}+1$ in parallel do
4:         $t = x[k + 2^{d} - 1]$
5:         $x[k + 2^{d} - 1] = x[k + 2^{d}+1 - 1]$
6:         $x[k + 2^{d}+1 - 1] = t + x[k + 2^{d}+1 - 1]$
```

(b) Figure 2.2: Downsweep

However, before performing the prefix sum operation, the array is divided amongst the processors. Each processor then calculates the sum of its assigned portion by applying the serial algorithm independently. The resulting sums from each processor are used as inputs for the upsweep and downsweep functions. Finally, the prefix sum array for each processor is obtained by adding the processor sums to the elements of the processor arrays. The parallelization of these steps is achieved using OMP for constructs as well.

3.4 MPI parallelization method

The MPI implementation of the scan algorithm is similar to the OMP algorithm, with upsweep and downsweep phases. However, the inherent sequential nature of the scan problem poses a challenge, as each calculation relies on the previous sum [2].

To overcome this, the processors are organized such that odd and even processors are paired together at each level of the tree. This pairing allows for communication between processors in a way that prevents the loop-carried dependency present in the serial implementation. In the upsweep phase, the odd processors initiate MPI sends, while the even processors receive. Additionally, in the downsweep phase, each processor is responsible for both sending and receiving, but the order of these operations varies based on whether the processor is odd or even. By exchanging values between processors, partial sums can be computed, leading to the desired scan operation. After the upsweep and downsweep phases have completed, MPIGather is used to combine the arrays of each processor to obtain the final completed array.

3.5 Correctness checks

For the parallel scan operations the correctness is tested simply by comparing the results of the parallel implementations with those achieved by the serial version. Simply looping through the results of both implementations and seeing if each element matches. If there is a discrepancy a relevant message will be displayed.

3.6 Evaluation of results and Discussion

In this study, we compared the performance of three scanning (prefix sum) algorithms: serial implementation, parallel implementation using OpenMP, and parallel implementation using MPI. The algorithms were tested on arrays of sizes ranging from 2^3 to 2^{27}.

The execution times for each algorithm were measured for each sample size, the results are shown in the graph below.

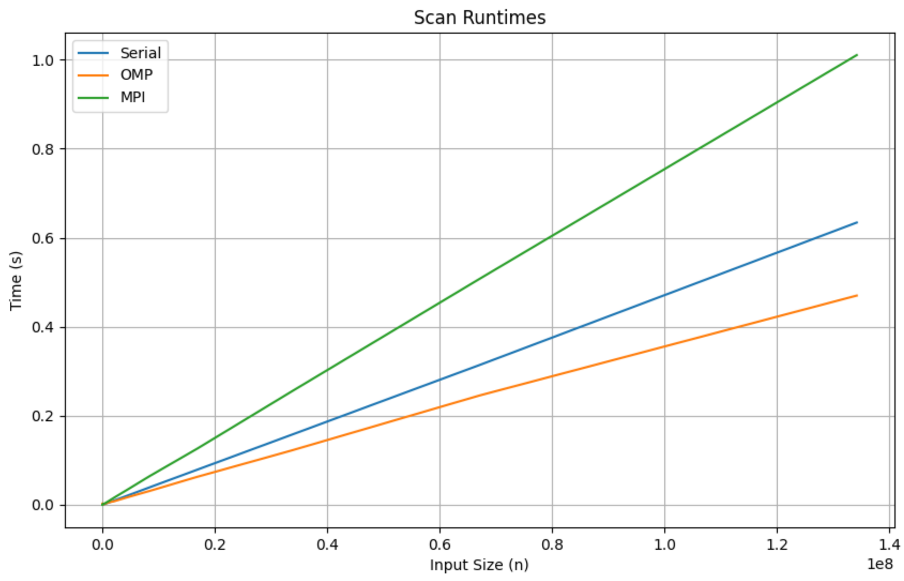


Figure 3: Scan runtimes

Evidently, by looking at the graph, the OMP implementation performed the best, it consistently performed better then both the serial and MPI versions. This algorithm used 8 threads to achieve its results but would likely have an even better performance with a higher thread count.

However due to hardware constraints the MPI version could only use two processors for its running, a possible explanation for its poor performance. Another explanation could be that the sample sizes tested were simply too low to achieve meaningful results for the MPI version.

Looking at the shape of all of the graphs they all seem to exhibit a linear time complexity, $O(n)$. Meaning that the performance gains achieved by the OMP version (and loss of the MPI version), are minimal and have not drastically changed the fundamental behaviour of the algorithm.

4 Parallel Bitonic Sort

4.1 Describing the problem

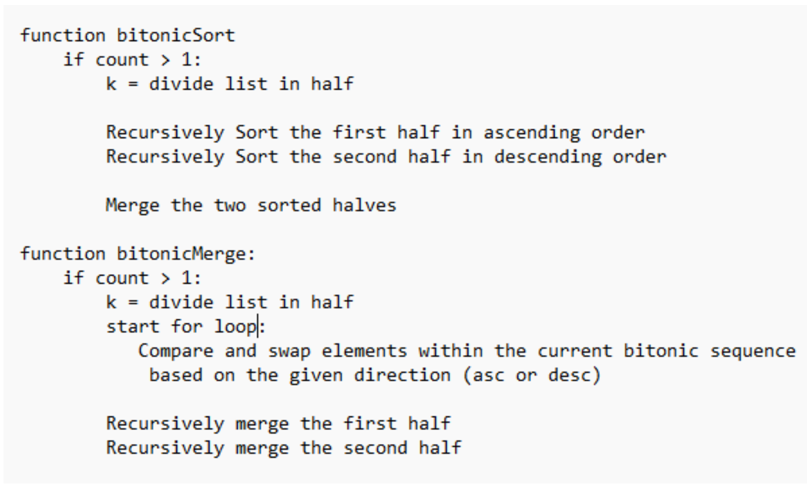
The following is an investigation of the bitonic sort algorithm. The bitonic sort algorithm is based on the concept of "bitonic sequences", this being a list of numbers which distinctly has increasing and decreasing sections. It has a nature that makes it well-suited for parallel computing architectures (primarily due to its recursive nature). Our objective is to develop a parallel implementation of this algorithm that can effectively sort a given array, allowing for efficient and scalable operations.

4.2 Serial Implementation

The serial implementation follows a straightforward approach, dividing the input list into two halves and recursively sorting each of these sublists. Following this, the two sorted halves are merged together after each pair of splits has completed. During the merging process, elements within the list are compared and potentially swapped to ensure their correct placement. The merge operation, leverages a similar recursive approach as the initial split recursion.

The recursive nature of the algorithm allows for the efficient sorting of increasingly smaller sublists, eventually resulting in a fully sorted array. This serial implementation serves as a foundation for the subsequent development of parallel versions of the algorithm. By thoroughly understanding and analyzing the serial implementation, we can identify potential opportunities for parallelization.

The basis for this pseudocode was taken from [1].



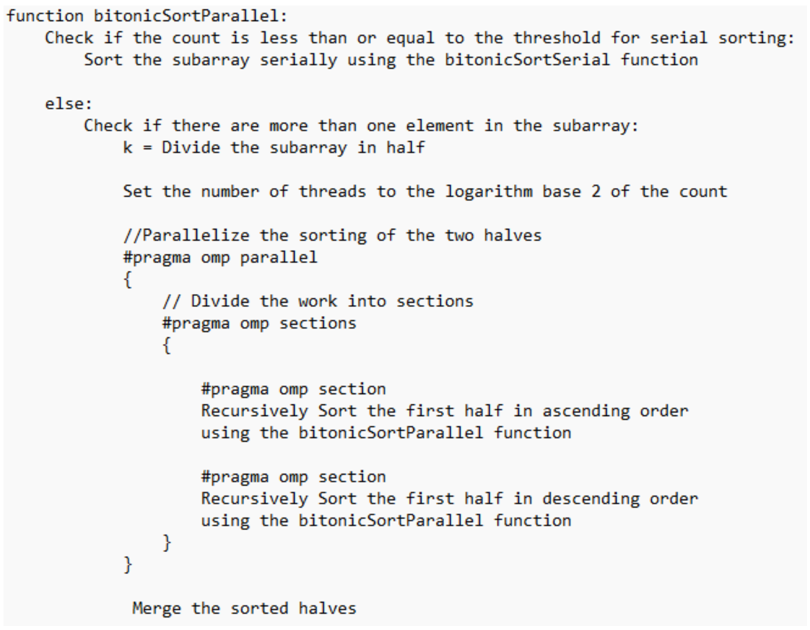
```
function bitonicSort
    if count > 1:
        k = divide list in half

        Recursively Sort the first half in ascending order
        Recursively Sort the second half in descending order

        Merge the two sorted halves

function bitonicMerge:
    if count > 1:
        k = divide list in half
        start for loop:
           Compare and swap elements within the current bitonic sequence
            based on the given direction (asc or desc)

        Recursively merge the first half
        Recursively merge the second half
```

Figure 4: Pseudocode for serial bitonic sort

4.3 OMP parallelization method

The inspiration for the parallelization technique was based on the method used to parallelize quicksort [7]. This is due to the fact that the implementations of bitonic sort and quicksort are similar in that they both recursively split and sort their lists in halves.

Hence the parallel code is near identical to the serial code with the only change being that the two recursive calls are placed inside section constructs. An attempt was made to parallelize the merging function in a similar way but it was inconclusive as to whether or not this provided any performance gains and is thus omitted in the final version. Lastly, a cut off point is implemented at list sizes less than 2048, at which it switches to the serial version.

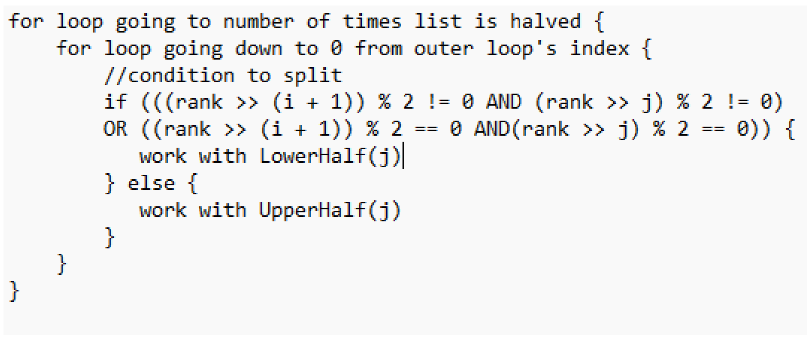


```
function bitonicSortParallel:
    Check if the count is less than or equal to the threshold for serial sorting:
        Sort the subarray serially using the bitonicSortSerial function

    else:
        Check if there are more than one element in the subarray:
            k = Divide the subarray in half

            Set the number of threads to the logarithm base 2 of the count

            //Parallelize the sorting of the two halves
            #pragma omp parallel
            {
                // Divide the work into sections
                #pragma omp sections
                {

                    #pragma omp section
                    Recursively Sort the first half in ascending order
                    using the bitonicSortParallel function

                    #pragma omp section
                    Recursively Sort the first half in descending order
                    using the bitonicSortParallel function
                }
            }

             Merge the sorted halves
```

Figure 5: Pseudocode for serial bitonic sort

4.4 MPI parallelization method

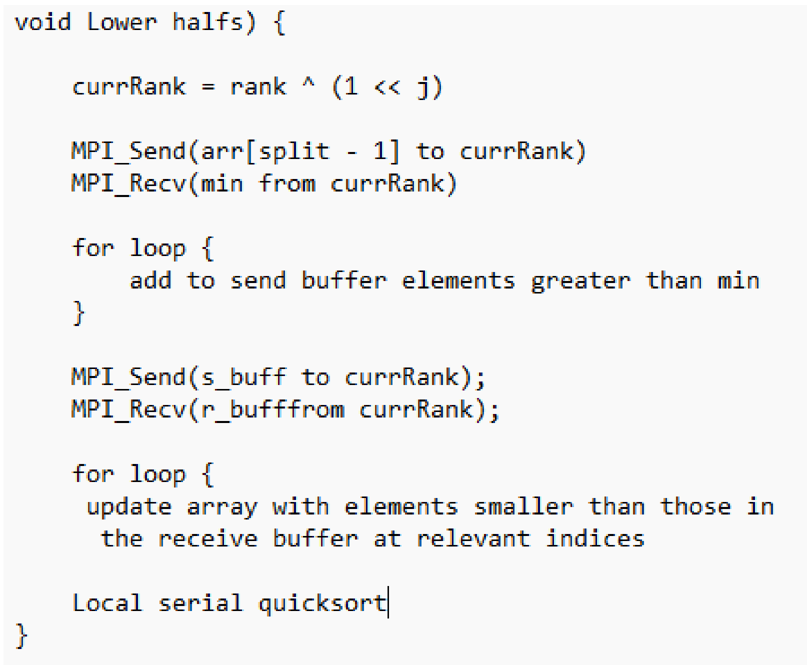
The MPI implementation follows a similar approach to the previous OMP and serial versions, with the main difference being the absence of recursion for sorting the halves. Instead, a loop is used to split the list into upper and lower halves, this is done in the below figure, the source of this method from [5].



```
for loop going to number of times list is halved {
    for loop going down to 0 from outer loop's index {
        //condition to split
        if (((rank >> (i + 1)) % 2 != 0 AND (rank >> j) % 2 != 0)
        OR ((rank >> (i + 1)) % 2 == 0 AND(rank >> j) % 2 == 0)) {
           work with LowerHalf(j)
        } else {
           work with UpperHalf(j)
        }
    }
}
```

Figure 6: Pseudocode nested loop that finds the right pair

Following this the functions for sorting the lower and upper half work by sending and receiving the minimum and maximum numbers in each sublist respectively [4], this allows them to determine the basis for further splits. Local sorting within each split is achieved using serial quicksort. The psuedocode for the lower comparison is shown below, the upper comparison is similar.



```
void Lower halfs) {

    currRank = rank ^ (1 << j)

    MPI_Send(arr[split - 1] to currRank)
    MPI_Recv(min from currRank)

    for loop {
        add to send buffer elements greater than min
    }

    MPI_Send(s_buff to currRank);
    MPI_Recv(r_bufffrom currRank);

    for loop {
     update array with elements smaller than those in
      the receive buffer at relevant indices

    Local serial quicksort
}
```

Figure 7: Pseudocode nested for lower comparison

4.5 Correctness checks

The correctness checks for the bitonic sorts are very simple. The final list is iterated through and each element is compared with the one at the following index, checking that it is less than or equal to that element. If this fails a message will be displayed.

4.6 Evaluation of results and Discussion

In this study, we compared the performance of three bitonic sorting algorithms: serial implementation, parallel implementation using OpenMP, and parallel implementation using MPI. The algorithms were tested on arrays of sizes ranging from 2^3 to 2^{25}.

The execution times for each algorithm were measured for each sample size, the results are shown in the graph below.

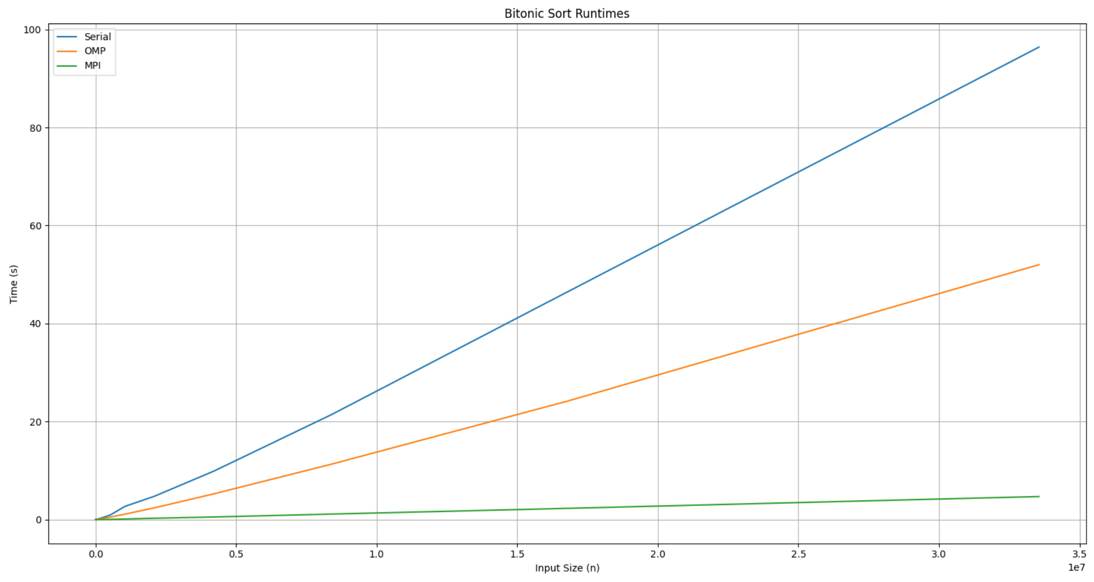


Figure 8: Bitonic sort runtimes

The graphs clearly demonstrate that both parallel implementations outperform the serial implementation. As the array size increases, the serial implementation's execution time grows faster, while the parallel implementations exhibit better scaling behaviors. Among the parallel implementations, the MPI algorithm consistently performed the best.

In summary, the results tested on varying array sizes highlight the significant performance improvements offered by parallel sorting algorithms compared to the serial implementation. These findings emphasize the importance of parallel computing techniques for efficient sorting, especially with large datasets.

5 Conclusion

In conclusion, the aim of our experiment was to explore and optimize parallel algorithms for the bitonic sort and scan operations. This was successful for parallel versions of the bitonic sort algorithm (including OpenMP and MPI implementations), as well as the OpenMP scan operation, which demonstrated improved performance compared to the serial implementation across a multitude of sample sizes. These parallel algorithms effectively utilized the available computational resources and demonstrated the potential for efficient parallelization.

However, the experiment yielded less favorable results for the MPI scan operation. Despite our efforts to parallelize this algorithm using MPI, the inherent sequential nature of the problem posed challenges in achieving significant performance improvements, as well as possible hardware constraints (limited to 2 processors for that experiment).

Nonetheless, this experiment provided valuable insights into the capabilities and limits of parallel algorithms, particularly when dealing with inherently sequential problems like the scan operation, and conversely with highly parallelizable problems such as the bitonic sort. It highlights the importance of carefully analyzing a problem's characteristics and dependencies to determine the feasibility and potential benefits of parallelization.

References

[1] Rahul Agarwal. Bitonic sort. https://www.geeksforgeeks.org/bitonic-sort/, 2022.

[2] Guy E. Blelloch. Prefix sums and their applications. https://www.cs.cmu.edu/afs/cs/academic/class/15750-s11/www/handouts/PrefixSumBlelloch.pdf, 1990.

[3] Mark Harris. Chapter 39. parallel prefix sum (scan) with cuda. https://developer.nvidia.com/gpugems/gpugems3/part-vi-gpu-computing/chapter-39-parallel-prefix-sum-scan-cuda.

[4] Mihai Florin Ionescuand and Klaus E. Schauser. Optimizing parallel bitonic sort. https://liuyehcf.github.io/resources/paper/Optimizing-parallel-bitonic-sort.pdf.

[5] Sajid Khan. Parallel bitonicsort implementation. https://cse.buffalo.edu/faculty/miller/Courses/CSE702/Sajid.Khan-Fall-2018.pdf, 2018.

[6] Hairong Wang. Coms3008a course assignment, 2023.

[7] Hairong Wang. Coms3008a: Parallel computing - introduction to openmp: Part iii, 2023.