COMS3008A Assignment 2 – Report

Kabelo M Rankoane (2603828)

June 02 2024

1 Bitonic Sort

Serial Pseudocode

This is the pseudocode for the bitonic sort algorithm in serial:
This is the basic idea of the bitonic sort algorithm. The bitonicMerge function is a recursive function that merges two

```
Function bitonicMerge(a[], low, cnt, dir)
    if cnt > 1 then
        k ← cnt / 2  for i ← low to low + k do
            if dir == (a[i] > a[i + k]) then
                swap a[i] and a[i + k]
            end
        end
        bitonicMerge(a, low, k, dir)  bitonicMerge(a, low + k, k, dir)
    end
end
Function bitonicSort(a[], low, cnt, dir)
    if cnt > 1 then
        k ← cnt / 2  bitonicSort(a, low, k, 1)  bitonicSort(a, low + k, k, 0)  bitonicMerge(a, low, cnt, dir)
    end
end
```

Figure 1: Bitonic Sort and Merge Algorithm in Serial

arrays of size $n/2$ each. The bitonicSort function is also a recursive function that first sorts the first half of the array in ascending order and the second half in descending order. The bitonicMerge function is then called to merge the two halves. This is the serial implementation and the base of what i will be using to implement the parallel version of the bitonic sort algorithm.

2 Baseline Algorithm

I used Quicksort as the baseline algorithm for this assignment. Quicksort is a comparison sort and is a divide-and-conquer algorithm. It works by selecting a 'pivot' element from the array and partitioning the other elements into two sub-arrays according to whether they are less than or greater than the pivot. I picked the last elements in the array as the pivot element. The pivot element is then placed in its correct position in the array and the array is partitioned into two sub-arrays. The quicksort function is then called recursively on the two sub-arrays. This is what i compare the time for my OPENMP and MPI implementations of the bitonic sort algorithm to.

3 Parallel Implementation

3.1 OPENMP Implementation

Using the Bitonic Sort Serial as a base, I used the task construct to create task that could be run in parallel. Two task are created in the bitonicSort and theres a taskwait so that the task are completed before the bitonicMerge is called. Then the bitonicMerge is called which then create 2 task that call bitonicMerge on the two halves of the array. This is done recursively until the array is sorted.But through testing i found that the parallel implementation had been slow so i added a cutoff point where the serial implementation would be used instead of the parallel implementation. This cutoff point was set to 1000. This allowed me to reduce the overhead of creating tasks for small arrays.

These are tehe Results of the OPENMP implementation on different number of threads on an array of size 2^{24}:

z

Threads	Speedup	Time Taken
1	0.2570223	14.018849
2	0.316548	11.382653
4	0.496615	7.219549
8	0.669296	5.419865
16	0.775642	4.649495
32	0.781515	4.599116

Table 1: Threads, Speedup and Time Taken

As we can see from the table, the speedup increases as the number of threads increase but it platuas at 32 threads. This is because the number of threads is greater than the number of cores on the machine i was testing on. This causes the overhead of creating threads to be greater than the speedup gained from the parallel implementation. This is why the speedup is the relativly the same for 16 threads and 32 threads. Therefore if i had to run the program on a machine with more cores would have achieved a higher speedup for 32 threads.

3.2 MPI Implementation

The MPI implementation of the bitonic sort algorithm is a bit more complex than the OpenMP implementation. Initially, the program runs the baseline algorithm on process 0 for benchmarking. We then run the Bitonic Sort MPI, distributing the array among the processes. Each process runs the bitonic sort on its piece of the array. Depending on the rank and number of processors, I assign the direction accordingly. After this, I will have n/2(n is the number of processors) bitonic

sequences that need to be sorted. I then let process 0 handle the merging of these sequences by calling my recursive bitonicMerge function.

Number of Processes	Speedup MPI	Time Taken MPI
1	0.191474	15.646154
2	0.302780	10.351526
4	0.417547	8.764265

Table 2: Number of Processes, Speedup MPI, and Time Taken MPI

4 Correctness and Performance Analysis

5 Conclusion