COMS3008A Assignment 2 – Report

Kabelo M Rankoane (2603828)

June 02 2024

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.

Bitonic Sort

Pseudocode

This is the pseudocode for the bitonic sort algorithm in serial:

```
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
```

This is the basic idea of the bitonic sort algorithm. The bitonicMerge function is a recursive function that merges two 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.

Parallel Implementation

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}: