Comparing Performance of Bitonic Sort in Sequential and Parallel Execution

[1]Keerthana Ningaraju, [2]Kuchi Ravi Teja, [3]M K Nikhil
(USN: [1]1MS18CS059, [2]1MS18CS060, [3]1MS18CS060)

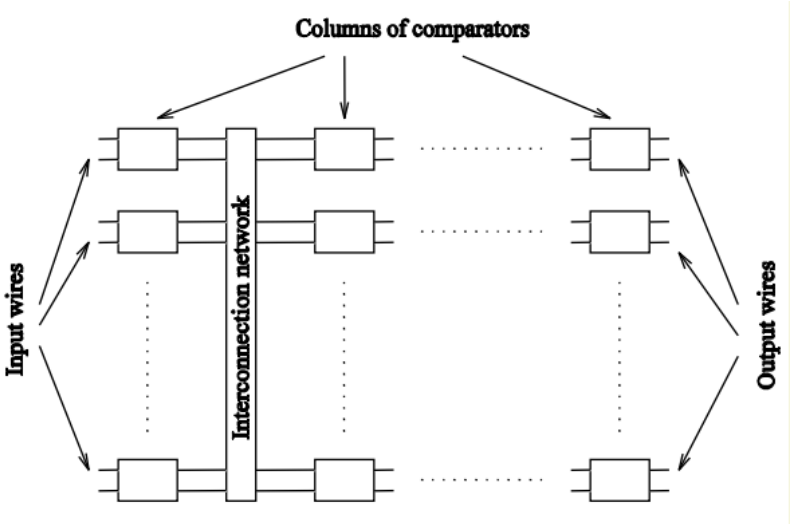
Abstract: We used Bitonic sort to construct and compare sequential and parallel sorting algorithms. Bitonic sort is a parallel-capable comparison-based sorting algorithm. It focuses on transforming a random number series into a bitonic sequence, which increases and lowers monotonically. A bitonic sequence's rotations are likewise bitonic. C++ was used to implement sequential algorithms on a central processing unit, whereas OpenMP was used to develop parallel algorithms on a central processing unit. We tested the algorithms on fifteen different input distributions containing numbers to the second power. The results show that the increase in speed up from serial to parallel makes the latter more advantageous.

1. Introduction

Bitonic sort is a parallel sorting algorithm that conducts comparisons in the order of O(n2log n). Despite the fact that it has more comparisons than any other popular sorting algorithm, it performs better in parallel because elements are compared in a specified order that is independent of the data being sorted. The Bitonic sequence is the predefined sequence.

SERIAL BITONIC SORT

Bitonic sort was created by Ken E. Batcher is one of the most researched algorithms due to its simplicity. It belongs to the sorting network family, which means that the order and direction of comparisons are predetermined and independent of the input sequence. In a Bitonic sequence, elements are placed in increasing order first, then decreasing order after a certain index.



An example of a bitonic sorting network. Diagram from wiki.rice.edu.

PARALLEL BITONIC SORT

Because shared memory size is restricted and long bionic sequences cannot be saved into it, parallel implementation of bitonic sort is very efficient when sorting short sequences, but it gets slower when sorting very long sequences. Instead of using shared memory, global memory must be employed to integrate large bionic sequences. Furthermore, every step of the bionic merging requires access to global memory.

2. Method

Serial code execution

```
#include<bits/stdc++.h>
#include <chrono>
#include <ctime>

using namespace std;

void compAndSwap(int a[], int i, int j, int order)
{
  if (order == (a[i]>a[j]))
    swap(a[i],a[j]);
}

void bitonicMerge(int a[], int low, int length, int order)
{
    if (length>1)
  {
    int k = length/2;
    for (int i=low; i<low+k; i++)
      compAndSwap(a, i, i+k, order);
    bitonicMerge(a, low, k, order);
    bitonicMerge(a, low+k, k, order);
  }
}

void bitonicSort(int a[],int low, int length, int order)
{
  if (length>1)
  {
    int k = length/2;

    // sort in ascending order because order is 1
    bitonicSort(a, low, k, 1);

    // sort in descending order because order is 0
    bitonicSort(a, low+k, k, 0);

    // merge whole sequence in ascending order
    bitonicMerge(a,low, length, order);
  }
}

int main()
{

    int n;
    cout << "Enter the number of elements to be sorted (number should be in the order of 2^n)";
    cin >> n;
    int *a = new int[n];
    srand(time(NULL));
    for (int i = 0; i < n; i++)
    {
        a[i] = i + rand() % 1000;
    }
    auto start = std::chrono::system_clock::now();
    int incr = 1; // 1 represents ascending order
    bitonicSort(a, 0, n, incr);

    printf("Sorted array: ");
    for (int i=0; i<n; i++)
        printf("%d ", a[i]);
    auto end = std::chrono::system_clock::now();

    std::chrono::duration<double> elapsed_seconds = end-start;
    std::time_t end_time = std::chrono::system_clock::to_time_t(end);

    std::cout << "Time taken for serial execution : " << elapsed_seconds.count() << "s\n";

    return 0;
}
```

Parallel code execution

```
#include <iostream> //for std::cout ,std::cin
#include <cstdlib>
#include <time.h>c
#include <omp.h>
using namespace std;

void ascendingSwap(int index1, int index2, int *ar) // Swap two values such that they appear in ascending order in the array
{
    if (ar[index2] < ar[index1])
    {
        int temp = ar[index2];
        ar[index2] = ar[index1];
        ar[index1] = temp;
    }
}
void decendingSwap(int index1, int index2, int *ar) // Swap two values such that they appear in decending order in the array
{
    if (ar[index1] < ar[index2])
    {
        int temp = ar[index2];
        ar[index2] = ar[index1];
        ar[index1] = temp;
    }
}
void bitonicSortFromBitonicSequence(int startIndex, int lastIndex, int dir, int *ar) // Form a increaseing or decreasing array when a bitonic input is given to
{
    if (dir == 1)
    {
        int counter = 0; // Counter to keep track of already swapped elements ,, parallelising this area results in poor performance due to overhead ,,need to
        int noOfElements = lastIndex - startIndex + 1;
        for (int j = noOfElements / 2; j > 0; j = j / 2)
        {
            counter = 0;
            for (int i = startIndex; i + j <= lastIndex; i++)
            {
                if (counter < j)
                {
                    ascendingSwap(i, i + j, ar);
```

```
                    counter++;
                }
                else
                {
                    counter = 0;
                    i = i + j - 1;
                }
            }
        }
    }
    else // Descending sort
    {
        int counter = 0;
        int noOfElements = lastIndex - startIndex + 1;
        for (int j = noOfElements / 2; j > 0; j = j / 2)
        {
            counter = 0;
            for (int i = startIndex; i <= (lastIndex - j); i++)
            {
                if (counter < j)
                {
                    decendingSwap(i, i + j, ar);
                    counter++;
                }
                else
                {
                    counter = 0;
                    i = i + j - 1;
                }
            }
        }
    }
}
void bitonicSequenceGenerator(int startIndex, int lastIndex, int *ar) // Generate a bitonic sequence from a random order
{
    int noOfElements = lastIndex - startIndex + 1;
    for (int j = 2; j <= noOfElements; j = j * 2)
    {
        #pragma omp parallel for //parallel implementation results in most performance gains here
```

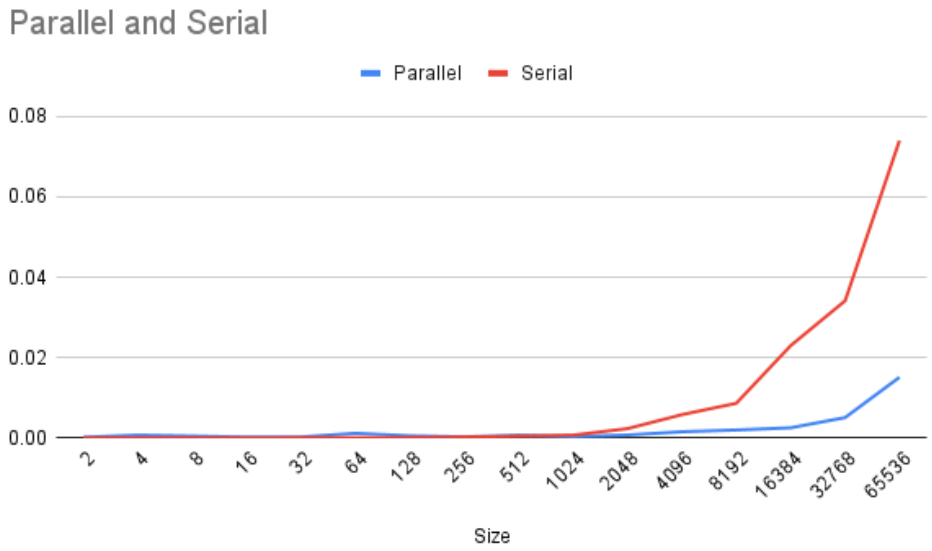
```
        for (int i = 0; i < noOfElements; i = i + j)
        {
            if (((i / j) % 2) == 0)
            {
                bitonicSortFromBitonicSequence(i, i + j - 1, 1, ar);
            }
            else
            {
                bitonicSortFromBitonicSequence(i, i + j - 1, 0, ar);
            }
        }
    }
}

int main() //main driver function
{
    omp_set_dynamic(0);                         // Disabled so that the os doesnt override the thread settings
    int maxNumberOfThreads = omp_get_num_procs(); // Gives number of logical cores
    omp_set_num_threads(maxNumberOfThreads);      // Set the no of threads
    int n;
    cout << "Enter the number of elements to be sorted (number should be in the order of 2^n)";
    cin >> n;
    int *ar = new int[n];
    srand(time(NULL));
    for (int i = 0; i < n; i++)
    {
        ar[i] = i + rand() % 1000;
    }
    double start, end;
    start = omp_get_wtime();
    bitonicSequenceGenerator(0, n - 1, ar);
    end = omp_get_wtime();
    cout << "Time taken for Parallel Execution  : " << end-start << endl;
}
```

The serial execution has three important functions, the first one is compAndSwap is a basic function to compare two numbers and swap them if the latter is greater. Secondly, bitonicMerge and bitonicSort functions for merging and sorting the array respectively.

The parallel execution has four important functions, two of them are ascending and descending sort which swap two values such that they appear in ascending and descending order in the array respectively. The next function is bitonicSortFromBitonicSequence which forms an increasing or decreasing array when a bitonic input is given to the function. Lastly, bitonicSequenceGenerator function generates a bitonic sequence from a random order. In parallel execution is implemented inside this function and results in most performance gains here.

3. Results

Size	Serial	Parallel
2	0.000025793	0.000171474
4	0.000010151	0.000680845
8	0.000019871	0.000454984
16	0.000018185	0.000199540
32	0.000030874	0.000219794
64	0.000040141	0.001142600
128	0.000069786	0.000509213
256	0.000205139	0.000262721
512	0.000418569	0.000649883
1024	0.000682430	0.000299951
2048	0.002300770	0.000707444
4096	0.005779860	0.001521800
8192	0.008608490	0.001958770
16384	0.022959700	0.002511480
32768	0.034109300	0.005050080
65536	0.073991300	0.015081000



Tabulate your results and do the analysis with the related graphs, point out the limitations of the code.

References