Comparing performance of Student marks sorting using Bubble Sort

Akash Hegde, Amith HB, Anup Vernekar, D Sanath Kumar

(USN: 1MS18CS013,1MS18CS017, 1MS18CS021 and 1MS18CS037)

**Abstract**

To use the CPU efficiently we need to use all its resources. Sorting is the technique to arrange the set of numbers in ascending or descending order. We make use of odd even transposition technique of bubble sort to enable parallel programming. In order to evaluate the performance of parallel programming we compare it with sequential bubble sort which does not utilize CPU efficiently. We compare the performance of both sequential and parallel sorting algorithm to see which algorithm is efficient. We use message passing interface in parallel sorting algorithm.

1. **Introduction**

In this day and age, new applications constantly require faster processors. Many of the commercial applications used nowadays are built using parallel systems. These systems are able to process large amounts of data in various ways, to achieve a high level of efficiency.

Parallel systems are all about breaking down discrete parts of instructions of a program so that they can execute them simultaneously on different CPUs.

Parallel systems are designed to decrease the execution time of programs by portioning them into various fragments and processing these fragments simultaneously, these systems can also be known as tightly coupled systems. A parallel system can deal with multiple processors, machines, computers, or CPUs etc. by forming a parallel processing bundle or a combination of both entities.

1. Application

Sort Student marks using Bubble sort

Bubble sort, sometimes referred to as sinking sort, is a simple [sorting algorithm](https://en.wikipedia.org/wiki/Sorting_algorithm) that repeatedly steps through the list, compares adjacent elements and [swaps](https://en.wikipedia.org/wiki/Swap_(computer_science)) them if they are in the wrong order. The pass through the list is repeated until the list is sorted. The algorithm, which is a [comparison sort](https://en.wikipedia.org/wiki/Comparison_sort), is named for the way smaller or larger elements "bubble" to the top of the list.

1. Parallelism techniques used in the work

Odd-Even Transposition Sort is a parallel sorting algorithm. It is based on the Bubble Sort technique, which compares every 2 consecutive numbers in the array and swap them if first is greater than the second to get an ascending order array. It consists of 2 phases – the odd phase and even phase:

Odd phase: Every odd indexed element is compared with the next even indexed element (considering 1-based indexing).

Even phase: Every even indexed element is compared with the next odd indexed element.

This article uses the concept of multi-threading, specifically pthread. In each iteration, every pair of 2 consecutive elements is compared using individual threads executing in parallel as illustrated below.

References

1. Modi, Jagdish, and Richard Prager. "Implementation of bubble sort and the odd-even transposition sort on a rack of transputers." Parallel computing 4.3 (1987): 345-348.

2. Ayoubi, Rafic, et al. "Hardware architecture for a shift-based parallel odd-even transposition sorting network." 2019 Fourth International Conference on Advances in Computational Tools for Engineering Applications (ACTEA). IEEE, 2019.

3. Kaur, Manpreet. "Comparative Study of Parallel Odd Even Transposition and Rank Sort Algorithm." International Journal of Soft Computing an Engineering 4.5 (2014).

1. **Method**

MPI methods used are:

1. MPI\_Init - Initialize the MPI execution environment

Syntax - int MPI\_Init(int \*argc, char \*\*\*argv)

1. MPI\_send – Sends message from one process to another process

Syntax - MPI\_Send(void \* data,

int count,

MPI\_Datatype datatype,

int destination,

int tag,

MPI\_Comm communicator)

1. MPI\_recv – Receives message from other process

Syntax- MPI\_Recv(void \* data,

int count,

MPI\_Datatype datatype,

int source,

int tag,

MPI\_Comm communicator, MPI\_Status\* status)

1. MPI\_Comm\_rank - Determines the rank of the calling process in the communicator

Syntax - int MPI\_Comm\_rank(MPI\_Comm comm, int \*rank)

1. MPI\_Comm\_size – Determines the size of the group associated with a communicator

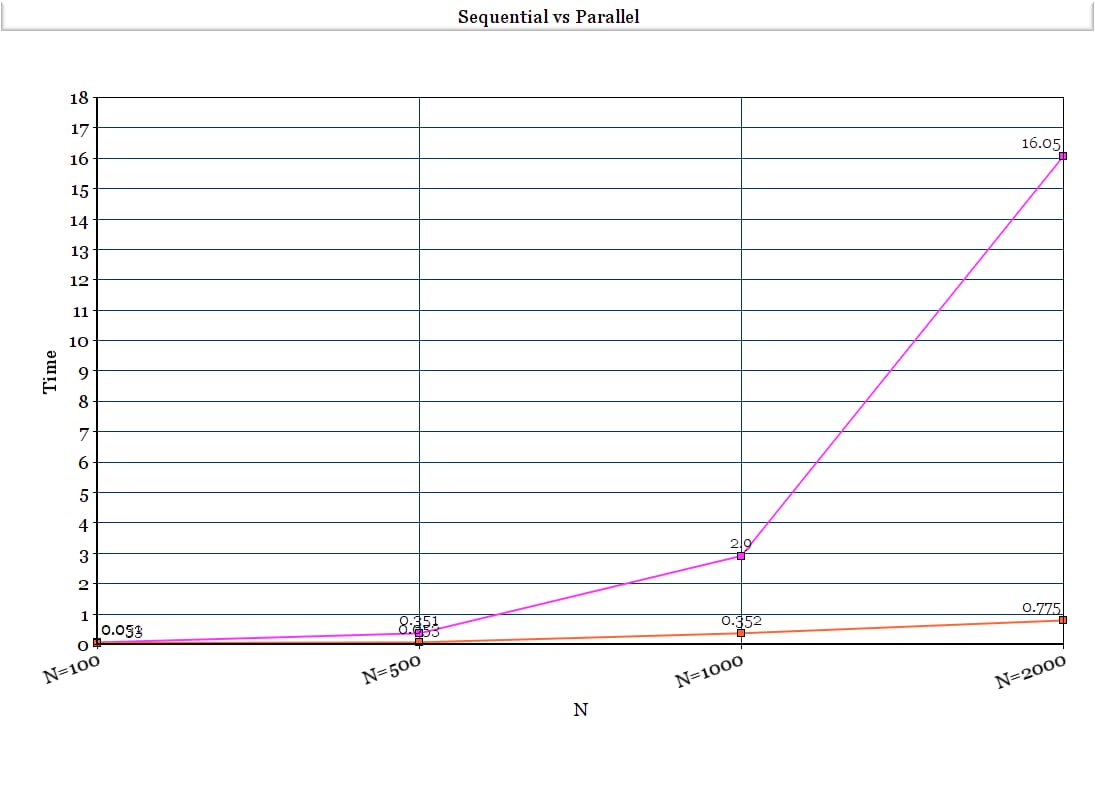
Syntax - int MPI\_Comm\_size( MPI\_Comm comm, int \*size )

|  |  |
| --- | --- |
|  |  |
|  | #include<stdio.h> |
|  | #include <stdlib.h> |
|  | #include <time.h> |
|  | const int n = 5000; |
|  | //generate random marks from 50 to 100 |
|  | void give\_me\_random(int\* marks){ |
|  | int i; |
|  | for(i = 0;i < n;i++) |
|  | marks[i] =(int)rand()%(100-50+1) + 50; |
|  | } |
|  | //print marks |
|  | void print(int\* marks){ |
|  | int i; |
|  | for (i = 0;i < n;i++) |
|  | printf("%d \t",marks[i]); |
|  | printf("\n"); |
|  | } |
|  | int compare(const void\* a,const void\* b){ |
|  | return (\*(int\*)a - \*(int\*)b); |
|  | } |
|  | //index of max element in marks |
|  | int find\_max(int\* marks){ |
|  | int i, index = 0; |
|  | int max = marks[0]; |
|  | for (i = 1; i < n; i++) { |
|  | if (marks[i] > max) { |
|  | max = marks[i]; |
|  | index = i; |
|  | } |
|  | } |
|  | return index; |
|  | } |
|  | //index of min element in marks |
|  | int find\_min(int\* marks) { |
|  | int i, index = 0; |
|  | int min = marks[0]; |
|  | for (i = 1; i < n; i++) { |
|  | if (marks[i] < min) { |
|  | min = marks[i]; |
|  | index = i; |
|  | } |
|  | } |
|  | return index; |
|  | } |
|  | //sequential bubble sort function |
|  | void bubble\_sort\_sequential(int\* marks, int size){ |
|  | int i,j,t; |
|  | clock\_t start = clock(); |
|  | for (i = 0;i < (n-1); i++){ |
|  | for (j = 0 ; j < n-i-1; j++){ |
|  | if (marks[j] > marks[j+1]){ |
|  | t = marks[j]; |
|  | marks[j] = marks[j+1]; |
|  | marks[j+1] = t; |
|  | } |
|  | } |
|  | } |
|  | clock\_t end = clock(); |
|  | float elapsed = (float)(end - start) / CLOCKS\_PER\_SEC \* 1000; |
|  | printf("\nIt needed %f miliseconds \n", elapsed); |
|  | printf("Sequential sorted marks:\n"); |
|  | for (i = 0;i < n; i++ ) |
|  | printf("%d \t", marks[i]); |
|  | printf("\n"); |
|  | } |
|  | //parallel bubble\_sort\_parallel |
|  | void bubble\_sort\_parallel(int\* marks, int rnk, int size){ |
|  | clock\_t begining = clock(); |
|  | int i; |
|  | int second\_marks[n]; |
|  | for (i = 0;i < size;i++){ |
|  | qsort(marks, n, sizeof(int), &compare); |
|  | int partner; |
|  | if(i%2 != 0){ |
|  | if (rnk % 2 == 0) { |
|  | partner = rnk - 1; |
|  | } else { |
|  | partner = rnk + 1; |
|  | } |
|  | } |
|  | else { |
|  | if (rnk % 2 == 0) { |
|  | partner = rnk + 1; |
|  | } else { |
|  | partner = rnk - 1; |
|  | } |
|  | } |
|  | if (partner >= size || partner < 0 ) { |
|  | continue; |
|  | } |
|  | if (rnk % 2 == 0) { |
|  | MPI\_Send(marks, n, MPI\_INT, partner, 0, MPI\_COMM\_WORLD); |
|  | MPI\_Recv(second\_marks, n, MPI\_INT, partner, 0, MPI\_COMM\_WORLD, MPI\_STATUS\_IGNORE); |
|  | } else { |
|  | MPI\_Recv(second\_marks, n, MPI\_INT, partner, 0, MPI\_COMM\_WORLD, MPI\_STATUS\_IGNORE); |
|  | MPI\_Send(marks, n, MPI\_INT, partner, 0, MPI\_COMM\_WORLD); |
|  | } |
|  | if(rnk < partner){ |
|  | while(1){ |
|  | int min\_index = find\_min(second\_marks); |
|  | int max\_index = find\_max(marks); |
|  | int t; |
|  | if(second\_marks[min\_index] < marks[max\_index]){ |
|  | t = second\_marks[min\_index]; |
|  | second\_marks[min\_index] = marks[max\_index]; |
|  | marks[max\_index] = t; |
|  | } |
|  | else break; |
|  | } |
|  | } else { |
|  | while(1){ |
|  | int min\_index = find\_min(marks); |
|  | int max\_index = find\_max(second\_marks); |
|  | int t; |
|  | if (second\_marks[max\_index] > marks[min\_index]) { |
|  | t = second\_marks[max\_index]; |
|  | second\_marks[max\_index] = marks[min\_index]; |
|  | marks[min\_index] = t; |
|  | } else break; |
|  | } |
|  | } |
|  | } |
|  | clock\_t end = clock(); |
|  | float elapsed = (float)(end - begining) / CLOCKS\_PER\_SEC \* 1000; |
|  | printf("It needed %f miliseconds \n", elapsed); |
|  | } |
|  | int main(int argc, char\* argv[]){ |
|  | int rnk, size = 5000; |
|  | int marks[n]; |
|  | int marks1[n]; |
|  | MPI\_Init(&argc, &argv); |
|  | MPI\_Comm\_rank(MPI\_COMM\_WORLD, &rnk); |
|  | MPI\_Comm\_size(MPI\_COMM\_WORLD, &size); |
|  | give\_me\_random(marks); |
|  | int i = 0; |
|  | for (i; i < n; i++){ |
|  | marks1[n] = marks[n]; |
|  | } |
|  | printf("marks before sorting: \n"); |
|  | print(marks); |
|  | printf("\n\n"); |
|  | bubble\_sort\_parallel(marks,rnk,size); |
|  | printf("marks after parallel sorting: \n"); |
|  | print(marks); |
|  | printf("\n\n"); |
|  | bubble\_sort\_sequential(marks,size); |
|  | MPI\_Finalize( ); |
|  | return 0; |
|  | } |

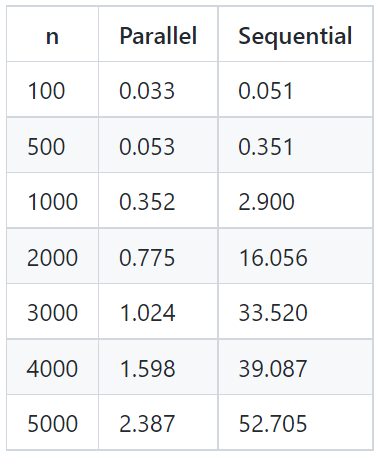
The sequential bubble sort is the traditional way of sorting. It works by repeatedly swapping the adjacent elements which are not in the order.

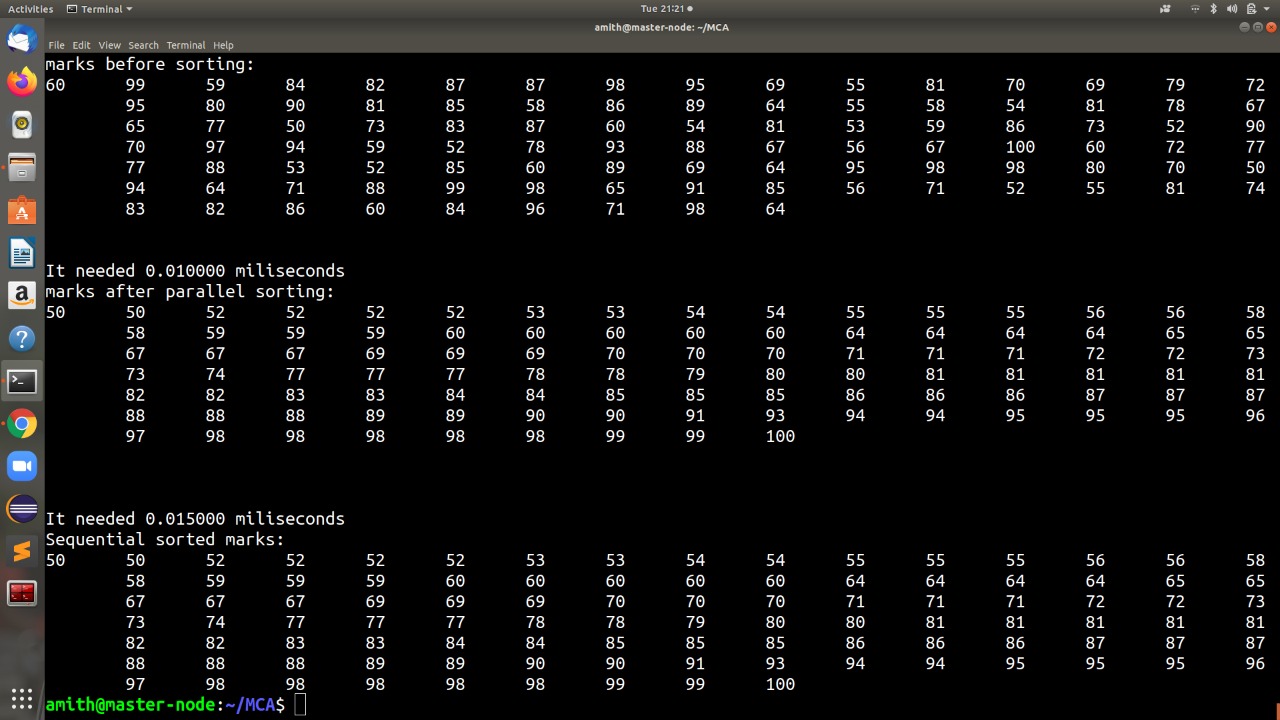
The parallel bubble sorting algorithm is done by using the odd even transposition technique which is discussed above. The function takes marks, rank and the size of the array as input and sorts the marks array. It calls find max and find min functions repeatedly to find the index of the max and min elements and sort the array accordingly.

3. Results

Purple line in graph represents sequential and orange line indicates parallel performance.

Limitations : difficult to visualize or plot the results for higher values of N since there is a huge difference between the time taken in sequential and parallel.





**References**

1. Modi, Jagdish, and Richard Prager. "Implementation of bubble sort and the odd-even transposition sort on a rack of transputers." Parallel computing 4.3 (1987): 345-348.

2. Ayoubi, Rafic, et al. "Hardware architecture for a shift-based parallel odd-even transposition sorting network." 2019 Fourth International Conference on Advances in Computational Tools for Engineering Applications (ACTEA). IEEE, 2019.

3. Kaur, Manpreet. "Comparative Study of Parallel Odd Even Transposition and Rank Sort Algorithm." International Journal of Soft Computing an Engineering 4.5 (2014).