Structured Data Standardization with one thread per feature

Contributors are: Asrar Farooq Bhat, Apurva Dewangan,Kaushik D,Fayyaz Ahmed Mohammad

Abstract

Parallel computing is the simultaneous use of multiple compute resources to solve a computational problem: A problem is broken into discrete parts that can be solved concurrently Each part is further broken down to a series of instructions.

Data standardization is the critical process of bringing data into a common format that allows for collaborative research, large-scale analytics, and sharing of sophisticated tools and methodologies. Standardizing data is a method in statistics for comparing two normal distributions if they have different arithmetic means and/or standard deviations.

So in order to make data available for research we need to standardised it but due to the presence of large amount of data we need faster mode of computation so just using sequential processing of data is sometimes time consuming so we need parallel programming so that many thread take up the work and we get results in less time.While making threads , it is to be taken care that none of the threads depend on one another that is there is no data dependency .So that there is no need for recomputation and waiting threads.

Introduction

Data standardisation is the process of converting data to a common format to enable users to process and analyze it.First of all, it helps you establish clear, consistently defined elements and attributes, providing a comprehensive catalog of your data. Whatever insights you're trying to get or problems you're attempting to solve, properly understanding your data is a crucial starting point. Like in artificial intelligence and machine learning the model training takes a large amount of time but if the data is found in better state then data can be made sense of easily. It also decreases training time and time taken to produce raw data.

- 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.

- Application

Sort Student marks using Bubble sort
Bubble sort, sometimes referred to as sinking sort, is a simple sorting algorithm that repeatedly steps through the list, compares adjacent elements and swaps 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, is named for the way smaller or larger elements "bubble" to the top of the list.

- 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).

- Method

MPI methods used are:

- MPI_Init - Initialize the MPI execution environment

  Syntax - int MPI_Init(int *argc, char ***argv)

- 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)

- 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)

- MPI_Comm_rank - Determines the rank of the calling process in the communicator

Syntax - int MPI_Comm_rank(MPI_Comm comm, int *rank)

- 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).