Algorithm Analysis and Design
Team - Parallelizers

Team members

Tushar Choudhary - 2019111019
Samyak Jain - 2019101013

Proposal

Our team decided to work on parallelizing various algorithms and improve the runtime. We picked algorithms such as Quick sort, K-means clustering, Sieve of Eratosthenes, Decision Tree etc. and parallelized their computation tasks. This involved analysing how an algorithm can be broken down and the individual segments can be worked out parallely. We coded the algorithms with and without parallelization and compared the results on a sample dataset. The written codes can be used as libraries corresponding to each algorithm.

Concurrency vs Parallelism

During the initial research we did, the terms concurrency and parallelism were found oftenly in relation to multithreaded programs. We misinterpreted their meaning, thinking concurrency and parallelism refer to the same concept. However later we understood that concurrency refers to doing multiple tasks together, however not necessarily at the same time, whereas parallelism actually computes multiple tasks simultaneously. It is easy to conclude that if we want to reduce the runtime, we should be using parallelism.

Beginning with concurrency

We initially started the project using our knowledge of concurrency, using multiprocessing and multithreading as a means to achieve it. The first algorithm I did was merge sort, and once we analysed the results, we were quick to figure out that we had started out in the wrong direction.

Since concurrency doesn't run tasks simultaneously, it won't provide us with improved timings since essentially it is using a single core. The independent tasks take some amount of CPU's time in regular intervals. But since the tasks will proceed in a linear way, it will take the same time as the normal code (even more, since we have to handle threads/processes).

Concurrency is a good solution for computers having a single core to carry out various processes together and make the computer responsive. But if we have multiple cores, we can do better by distributing the tasks over different cores and running our algorithms parallely, hence finishing faster.

OpenMP

After studying more, we experimented with OpenMP. OpenMP stands for Open Multi-Processing and is a library in C/C++ which supports shared memory multiprocessing. Samyak wrote the K-means clustering algorithm using this header file, and the runtime was again found to be longer than the normal code. This was because openmp essentially uses multithreading only, and doesn't use multiple cores, making it a more superficial than implementing the threads manually.

Mpi

On doing further research, we found the MPI library. MPI stands for Message passing interface is a library in C/C++ which supports parallel applications on multiple cores. Here we started getting improved timings. The codes we covered have been discussed later.

Things to keep in mind

The most important issue that arises when parallelizing an algorithm is how to break down an algorithm, and how much should we break it. Tasks should be divided into the chunks which should be evenly distributed on cores so all the cores give roughly the same timing. If the chunks can not be perfectly divided on the cores, then the remainder portion must be handled well to make sure the computations are error free. Also if we have multiple cores with us, then how many should we use is an important question. Since managing multiple processors also consumes time and resources, we won't want to use all of them as a lot of time will be wasted on sending and receiving data from all of them.

Ideal case

An ideal case would be when we would have an infinite amount of cores and we could instantaneously communicate with them. On proceeding with an algorithm, if we ever reach a point where two tasks could be carried out parallely, we would use two processors to carry them out. And this could be done recursively as the number of processors isn't a constraint. Since the processors could communicate in zero time, collecting the data from child computations would happen instantaneously.

However at present day, the hardware is much more limited than an ideal case. So we should try to improve the software we design so that we are able to use the available hardware we have as close to an ideal way.

Expectations on runtime

Initially we thought that using two cores will give half the time as using a single core. However in none of the algorithms we parallelised was this found to be true. As we discussed earlier, hardware at the present day is limited and hence the cores take time while communicating (time depends on number of cores). When the runtime of original code is very small, this communication time begins to matter and it is found that for very short computations, parallelising may even lead to a larger time since managing the cores took more than than doing the actual computation. This being said, in large test cases, parallel algorithms were always found to give better timing than normally algorithms.

Timings depending on type of algorithm

A thought that came up in my mind was will parallelising always improve timings for any algorithm? It is easy to figure out that in case algorithms such as dynamic programming, divide and conquer, recursive programs an improvement in timing can surely be made as work can properly be divided.

But now let's take an example of a linear search where the element we are looking for is in the middle of the array. One core starts its search from the beginning and one core from the end. Here in this case the time taken by a single core and two cores will turn out to be the same.

So to calculate rough improvement in timing, we need to know the algorithm and the possible situations that might arise.

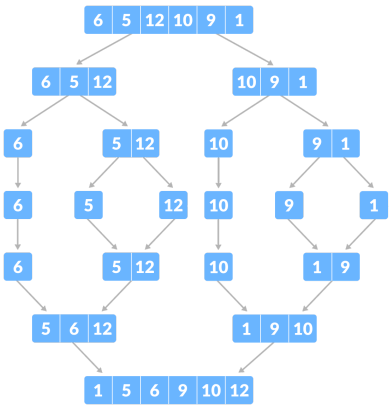
So should we parallelise algorithms ?

After implementing algorithms, we reached the question, so should we parallelise algorithms ? And the answer I concluded was, it depends on what the user wants. If the user has multiple cores and multiple tasks to finish before a deadline, then doing each task on a separate core would be a better deal as we would completely eradicate the core communication time in this case. But if we have multiple cores and a single task, for example, in case of a server, then distributing the tasks on multiple cores will without any doubt be good.

Summary of my algorithms

Merge sort

Merge sort is a sorting algorithm in which an array is split into two halves, the individual halves are sorted recursively in a similar fashion and then are merged so that the final array obtained is sorted.



To parallelise this algorithm I divided the given array into smaller segments (size of which depends on the number of cores) and sorted each segment using a core, and then finally merged the individual segments.

Matrix multiplication

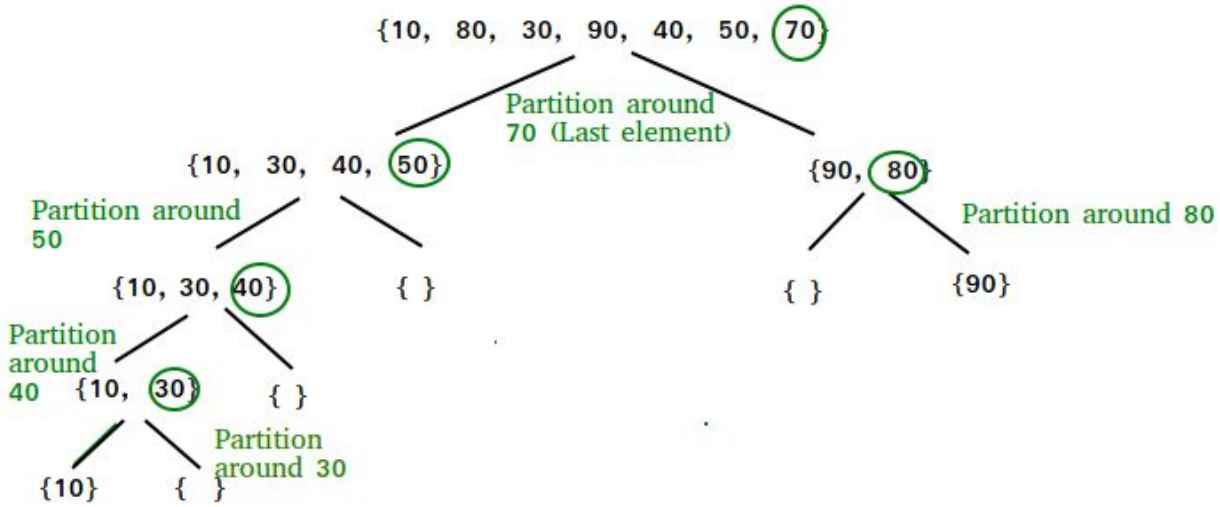
Given two matrices of size NxN, a basic matrix multiplication program would calculate the the product by running a loop of N^3 complexity and evaluating each entry as -

```
for (i = 0; i < N; i++) {
        for (j = 0; j < N; j++) {
            res[i][j] = 0;
            for (k = 0; k < N; k++)
                res[i][j] += mat1[i][k] * mat2[k][j];
```

Calculation of each res[i][j] is independent from res[i'][j']. Hence each entry can be calculated independently. In my parallelisation program, I have divided the rows of the first matrix among the cores available so that each core carries out all the computation related to the set of rows it has been assigned.

Quicksort

Quick sort is a divide and conquer algorithm to sort an array. It chooses one element of the array as a pivot and partitions the array to find its correct position. The two partitions follow this procedure repeatedly and the finally obtained array is sorted.



This algorithm has been parallelised by sending the initial recursive call to different processors. When all processors are finished sorting their partition then the final arrays partitions have been merged to obtain the sorted array.

Sieve of Eratosthenes

Given a number n, we use Sieve of Eratosthenes to find all the prime numbers smaller than n. In a basic algorithm, the steps followed to do so are -

1. Create a list of consecutive integers from 2 to n: (2, 3, 4, …, n).
2. Initially, let p equal 2, the first prime number.
3. Starting from 2p, count up in increments of p (here 2) and mark each of these multiples of 2 as non-prime. These numbers will be 2p, 3p, 4p, etc..
4. Find the first number greater than p in the list that is not marked. If there was no such number, stop. Otherwise, this p is a prime number. Now again go back to step 3.

While parallelising this algorithms, I divided the range of 2 to n into multiple segments and I have done the marking in each of the ranges simultaneously.

Another way to parallelise would be using multiple counters to traverse the list and mark the elements.

Data (average runtimes)

Mergesort

N = 1000
Single core = 0.000247 sec
Double core = 0.00130 sec

N = 1000000
Single core = 0.167 sec
Double core = 0.124 sec

Matrix Multiplication

N = 200
Single core = 0.053 sec
Double core = 0.041 sec

N = 400
Single core = 0.456 sec
Double core = 0.244 sec

Quicksort

The runtimes in this code can vary as the array has been randomly generated and there is no fixed complexity.
I have attached a screenshot for some examples in the quicksort folder.

Sieve

N = 10000
Single core = 0.00034 sec
Double core = 0.00021 sec

N = 1000000
Single core = 0.029 sec
Double core = 0.007 sec

Timeline

01/10 to 05/10 - finished mergesort using concurrency
06/10 to 15/10 - read up upon openmp and mpi
16/10 to 28/10 - completed mergesort
01/11 to 06/11 - completed quicksort
06/11 to 14/11 - completed sieve
14/11 to 18/11 - completed matrix multiplication
20/11 to 22/11 - compiled data and completed report