Prateek Bhatnagar
Matriculation ID : 4805056

CDS -LAB Report
Distributed System Engineering

1. Abstract

This report is about the parallelization of program Average Mean Distance. It explains what are the techniques considered while doing parallelization for this and how finally the speedup is achieved along with measurements.

2. Introduction

Given a directed weighted graph G = (V, E, w) with V the set of vertices, E the set of edges (ordered pairs of vertices), and w : E → N ∈ [1, |V||[the weight of the edges, we want to compute its average minimum distance (AMD) between all pairs of vertices with a path connecting them. Two algorithms can be used to compute All pair shortest path

1. Dijkstra's
2. Floyd Warshall

The task given in this problem is using the Floyd Warshall Algorithm. The algorithm works by first computing shortest Path (i,j,k) for all (i,k) (k,j) pairs for k=1, then k=2, etc. This process continues until k=N, and the shortest path for all (i,j) pairs have been found out using any intermediate vertices

Pseudo code for the algorithm is as follows:-

```
1 let dist be a |V| × |V| array of minimum distances initialized to
∞ (infinity)
2 for each edge (u,v)
3    dist[u][v] ← w(u,v)  // the weight of the edge (u,v)
4 for each vertex v
5    dist[v][v] ← 0
6 for k from 1 to |V|
7    for i from 1 to |V|
8       for j from 1 to |V|
9          if dist[i][j] > dist[i][k] + dist[k][j]
10             dist[i][j] ← dist[i][k] + dist[k][j]
11         end if
```

3. Basics

Before going further and introduce the concept of approaches to be used for parallelizing this algorithm, let's look into some basic concepts.

3.1 Sequential Vs Parallel Computation

In this type of computation, a process/CPU executes program step by step i.e. an instruction or command is executed completely before executing the next one. As opposed to sequential computing, in parallel computing (also known as concurrent computing) several instructions are executed in parallel or concurrently in an overlapping time period. In this one process need not wait for another process to complete its execution. In terms of threads, in sequential computing only one main thread executes the whole program and in parallel computing, multiple threads execute some parts of the program.

3.2 Level of Parallelism

3.2.1 Bit-level

In bit-level parallelism processor word size is increased, hence it reduces the number of instructions executed by a processor to perform computation on variables who have a size greater than the length of the word.

3.2.2 Instruction-level

Instruction-level parallelism means multiple processors units simultaneously execute instructions thus improving performance.

3.2.3 Data parallelism

Data parallelism focuses on distributing the data across different nodes, which operate on the data in parallel. It can be applied to regular data structures like arrays and matrices by working on each element in parallel

3.2.4 Task parallelism

Task parallelism focuses on distributing tasks—concurrently performed by processes or threads—across different processors. In contrast to data parallelism which involves running the same task on different components of data, task parallelism is distinguished by running many different tasks at the same time on the same data.

3.3 Amdahl's Law

According to Amdahl's law, if p is the part of the program that can be parallelized then 1-P is a part that can't be parallelized (which means it is serial), hence speed up on n number of processors/core is achieved as:-

$$A = \frac{1}{1 - p + p/n}$$

Hence even if p parallel parts speed up perfectly performance is limited by the sequential part.

3.4 Synchronization Barrier

In parallel computing, a barrier is a type of synchronization method. A barrier for a group of threads or processes in the source code means any thread/process must stop at this point and cannot proceed until all other threads/processes reach this barrier.

4. Approach used

4.1 Cache Cohrency

Cache coherency is a concept in a multi-processor environment. Each core or processor has its dedicated cache, and copy of variables(data)which is not always up-to-date. Let's consider an example if core 1 and core 2 are working on shared data and they both have brought the same block of memory in their respective caches. If one core writes a value or changes the value, then another core which attempts to read the value from its cache will not have the most recent version unless its cache is invalidated in this case cache miss occurs and core has to update its cached entry by fetching the latest data again. Hence this has a great performance impact as it requires to load data again in the cache using the cache line. The cache line is shared among multiple processors and it works on the principle "If you use an address now, you will probably use a nearby address soon in the same cache line". The data is used in a way such that all contiguous memory location in cache line are worked on before new data is brought back in the local cache. Hence to represent the graph an adjacency matrix is used. An adjacency matrix is v * v matrix where v is the number of nodes in the graph,

1. An A(ij) represents the distance between vertex i to vertex j
2. if no vertex is defined then A(ij) is set to INFINITY
3. In our example self-edges like A11, A22, A33, etc. are not considered and hence these are set equal to 0.

A one-dimensional array is used to solve the problem of consecutive rows to represent adjacency matrix and using arithmetic rows and columns are mapped with respective matrix element. Aij = A[i*v + j].This approach ensures that all rows of a 2D array are stored consecutively in memory. i.e. The first element of row 2 comes immediately after the last element of row 1 in memory. Using arithmetic rows and columns are mapped. Aij = A[i*v + j].This approach ensures that all rows of a 2D array are stored consecutively in memory. i.e. The first element of row 2 comes immediately after last element of row 1 in memory.

4.2 Synchronization - Barriers

It is known that all the elements of a matrix for each k iteration can be calculated in parallel. A technique is needed to parallelize the k loop. Once one iteration is completed for k loop a wait is needed until all the elements are calculated for this iteration before proceeding to the next iteration. The next iteration is depended upon the previous one. So the array is updated and wait until all the thread finishes the execution, hence barrier is used to synchronize all the threads for a single iteration of k loop. i.e. it works on the concept that everyone reads before everyone writes and everyone writes before everyone reads.

4.3 Partition

Partitioning is used to divide the array into small chunks or blocks so that each thread/core is assigned to a block of matrix. For this, block/chunk is computed dynamically. To achieve this, calculate block size = number of nodes/number of cores. As the number of cores increases block size decreases and each thread work gets easier and smaller and scales up with the number of cores/threads.

4.4 Divide and Conquer (Data Parallelism)

After dividing into blocks, each part of the array/matrix is assigned to a thread ,so that the array can be worked upon in parallel by defining the start and end for i loop as:-

1. for i = 0;i < v - block;i += block
2. start = i
3. end = i + block
4. start = i
5. end = v

This for loop defines the start and end of the block and the remaining block can be taken care of with start and end after them, wherein

1. start = i (index of last block)
2. end = v.

Hence this way work is divied among threads and conquer smaller blocks and later combine them. Concept of Data Parallelism is being used here as the same operation is performed again and again by each thread on respective block of data.

4.5 Localization/Duplication

Next, the data is duplicated for each thread so that each thread can work upon its respective block independently and there is no dependency of variables. For this, the required variables are passed which are to be duplicated in a function call and hence each thread gets their respective copies and can work on the data locally.

mdAllPairs(dists []uint32,v uint32,start uint32 , end uint32, k uint32, wg * sync.WaitGroup)

4.6 Precomputation

To check which computation or statement is taking longer time , CPU profiling tools are used to optimize the code.

4.6.1 Precomputation of array elements

dist[i*v+k] is calculated to further optimize algorithm by precomputing it in a variable before any operation is performed on these values, which helps to save some time.

```
for i = start;i < end;i++{
   A := dists[i * v + k]
   if A != INF {
      for j = 0; j < v; j++ {
         B := dists[x+j]
```

```
if B != INF {
   var intermediary= A + B;
   C := dists[i*v+j]
   if (intermediary < C) {
      dists[i*v+j] = intermediary
   }
```

4.6.2 Precomputation of array index

In j loop, index element k*v is always constant hence it can be precomputed and some time of computation can be saved

```
x := k*v
for i = start;i < end;i++{
   A := dists[i * v + k]
   if A != INF {
      for j = 0; j < v; j++ {
         B := dists[x+j]
```

4.6.3 Precomputation of array index

There are many elements which are set to INF (infinity) hence we can ignore those elements to optimize code further

```
for i = start;i < end;i++{
   A := dists[i * v + k]
   if A != INF {
      for j = 0; j < v; j++ {
         B := dists[x+j]
         if B != INF {
            var intermediary= A + B;
            C := dists[i*v+j]
            if (intermediary < C) {
               dists[i*v+j] = intermediary
```

5. Time Complexity

Let's assume n is the number of nodes. To find all n * n nodes , all k iteration is needed and two loops for i and j are needed. One outer loop k. Hence three loops are required from 1 to n, giving a time complexity of O(n*3)

6. Implementation

Go programming language is used here to implement parallelism and goroutines are used. Goroutines have the following advantages over threads:-

1. Goroutines have a faster start-up time than threads.
2. Goroutines come with built-in primitives to communicate safely between themselves called as channels.
3. Goroutines are extremely cheap when compared to threads. They are only a few kb in stack size and the stack can grow and shrink according to needs of the

application whereas in the case of threads the stack size has to be specified and is fixed.

Go routine is a type of message passing implementation. Because of all the above reason go routines have been used here to implement Average Mean Distance

7. Measurements

In this section three types of comparison are shown

1. Baseline – This is the provided 'C' program
2. Optimization A – This is the rewritten version in 'Go'
3. Optimization B – This is the parallelized version in 'Go'
4.

For calculating speed up following formula is used :-

$$speedup = \frac{Old\ Execution\ Time}{New\ Execution\ Time}$$

All the measurement data have been used from the logs as mentioned in Appendix A

7.1 Baseline Vs Optimization A

Detailed measurement values used in graph can be found in table 1 at Appendix A

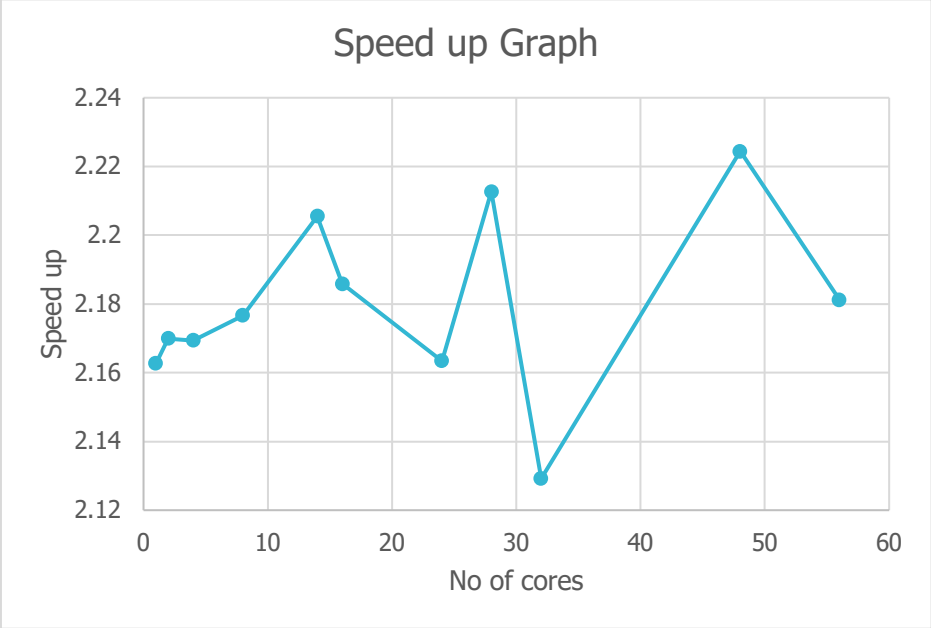

Figure 1

7.2 Baseline Vs Optimization B

On comparing with baseline algorithm vs scalability evaluation got speed up of almost 113 at 56 cores is attained as shown below.

Detailed measurement values used in graph can be found in table 2 at Appendix A

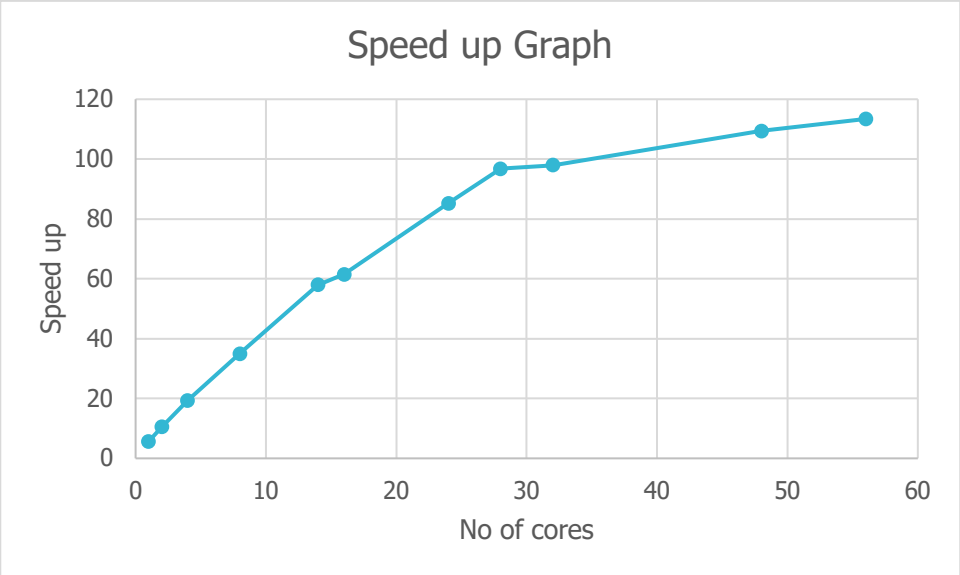

Figure 2

7.3 Optimization A vs Optimization B

On comparing with Optimization A vs scalability evaluation got speed up of almost 52 at 56 cores is attained as shown below.

Detailed measurement values used in graph can be found in table 3 at Appendix A

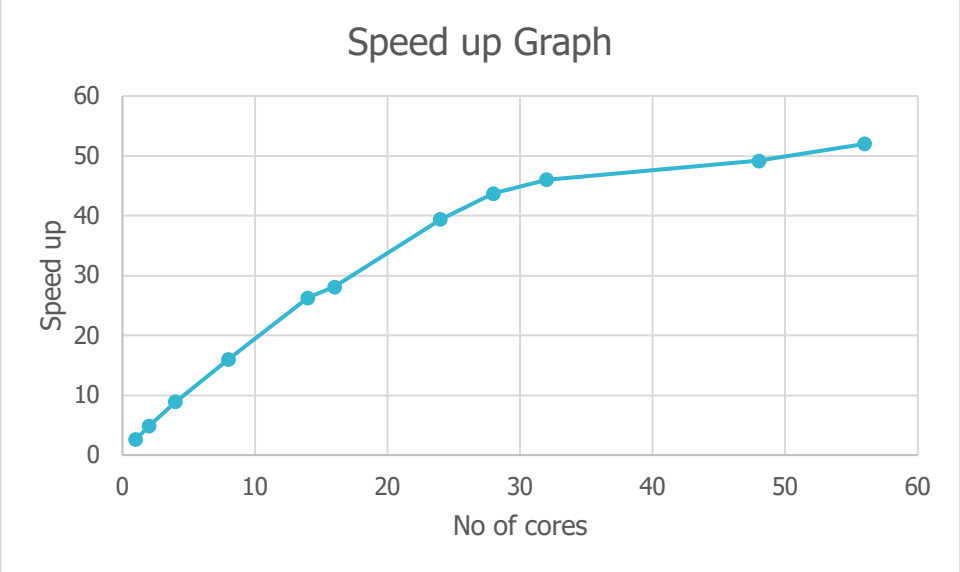

Figure 3

7.4 Scalability

On comparing scalability evaluation speed up of almost 20 at 56 cores is attained.

Detailed measurement values used in graph can be found in table 4 at Appendix A

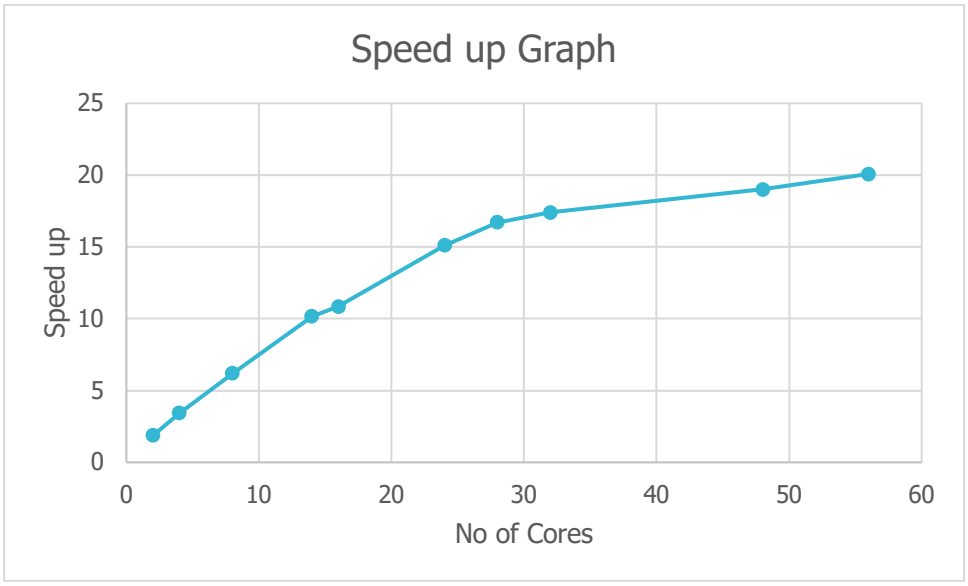

Figure 4

8. Parallelization

To calculate how many percentage of program is executed in parallel Amdahl's law is used :-

$$A = \frac{1}{1 - p + p/n}$$

Hence if A and n are knows then 'p' formula would be

$$p = \frac{(1 - A) * n}{A * (1 - n)}$$

A=Execution Time
n =No of cores
p= part which is parallelized

If this equation is solved then p = .97 , which means parallel factor of almost 97% is achieved.
Detailed measurement values used in calculation can be found in table 5 at Appendix A.

9. Conclusion

To conclude various technique used to parallelize average mean distance and attain speedup with those techniques are discussed and results and measurements are also shown.

10. Appendix A

Measurement	Git Hash	Evaluation Url
Baseline	8b7bf47	https://cds-lab-2019.netlify.com/logs/a53bb77da9138d603bfa1ccd449ddfb33f7e72ac5ff5f5811eccef587c7b4548/2019-06-14T22:37:41+02:00.log
Optimization A	fec2e3c	https://cds-lab-2019.netlify.com/logs/a53bb77da9138d603bfa1ccd449ddfb33f7e72ac5ff5f5811eccef587c7b4548/2019-06-13T04:45:51+02:00.log
Optimization B	6715771	https://cds-lab-2019.netlify.com/logs/a53bb77da9138d603bfa1ccd449ddfb33f7e72ac5ff5f5811eccef587c7b4548/2019-06-16T16:22:25+02:00.log

10.1 Baseline Vs Optimization A

No of Cores	Baseline (µs)	Optimization A (µs)	Speed up Graph
1	128331387	59338252	2.16270926
2	128278357	59116426	2.16992747
4	128199461	59093943	2.16941796
8	128401325	58989639	2.17667589
14	129906271	58901328	2.20548968
16	128907000	58973629	2.1858414
24	128295119	59302880	2.16338766
28	131717388	59528266	2.21268646
32	128197738	60209359	2.12919952
48	130980940	58882104	2.22446093
56	128519079	58920218	2.18123903

Table 1

10.2 Baseline Vs Optimization B

No of Cores	Baseline (µs)	Optimization B (µs)	Speed up Graph
1	128331387	22749838	5.64098026
2	128278357	12210293	10.5057558
4	128199461	6629257	19.3384358
8	128401325	3677260	34.9176629

14	129906271	2240211	57.9884087
16	128907000	2096952	61.4735101
24	128295119	1506277	85.173656
28	131717388	1360945	96.7837701
32	128197738	1308159	97.9985904
48	130980940	1197527	109.376189
56	128519079	1133173	113.415232

Table 2

10.3 Optimization A vs Optimization B

On comparing with Optimization A vs scalability evaluation got speed up of almost 52 at 56 cores is attained as shown below

No of cores	Optimization A (µs)	Optimization B (µs)	Speed up Graph
1	59338252	22749838	2.60829339
2	59116426	12210293	4.84152395
4	59093943	6629257	8.91411255
8	58989639	3677260	16.0417373
14	58901328	2240211	26.292759
16	58973629	2096952	28.1234997
24	59302880	1506277	39.3705009
28	59528266	1360945	43.7403907
32	60209359	1308159	46.0260251
48	58882104	1197527	49.1697507
56	58920218	1133173	51.9957835

Table 3

10.4 Scalability

On comparing scalability evaluation speed up of almost 20 at 56 cores is attained

No of cores	Optimization B (µs)	Speed up Graph
1	22749838	1
2	12210293	1.863168885
4	6629257	3.431732696
8	3677260	6.186627543

14	2240211	10.15522109
16	2096952	10.84900274
24	1506277	15.10335616
28	1360945	16.71620675
32	1308159	17.3907285
48	1197527	18.9973487
56	1133173	20.07622667

Table 4

10.5 Parallelization

No of Cores	Parallel Factor
2	0.926560005
4	0.944802684
8	0.958126911
14	0.970876831
16	0.968347338
24	0.974389096
28	0.974999216
32	0.97290126
48	0.967517694
56	0.967466022

Table 5