Indian Institute of Technology Indore

Report for Lab Assignment 2

Parallel Computing Lab (CS 359)

Author
Aniket Sangwan (cse180001005@iiti.ac.in)

April 13, 2021

Contents

Chapter 1

Report

1.1 Problem Statement

Write a multi-core program for the "all-pairs shortest paths" problem. The input is a weighted graph with no negative cycles and the expected output are lengths of the shortest paths between all pairs of vertices (where the length of a path is a sum of weights along the edges that the path consists of). Write a sequential program (no OpenMP directives at all) as well. Compare the running time of the sequential program with the running time of the multi-core program and compute the speedup achieved:

- for different number of cores and different number of threads per core, and
- for different number of vertices and different number of edges.

1.2 Approach

```
void Floyd_Warshall(){
  for(int k=0;k<n;k++){

    #pragma omp parallel for num_threads(size) collapse(2)

      for(int i=0;i<n;i++){

        for(int j=0;j<n;j++){

          int v=matrix[i*n+k];
          if(v==inf) continue;
          if(matrix[k*n+j]==inf) continue;
          int val=v+matrix[k*n+j];

          if(val<matrix[i*n+j]) matrix[i*n+j]=val;

        }
      }
  }
}

```

- Floyd Warshall's Algorithm works by iterating over all the nodes that can act as the intermediate node between x and y.
- At k^{th} iteration of Floyd Warshall's algorithm, we have the shortest distance between all pairs of vertices with intermediate vertices present only in the set $0, 1, 2, ...k-1$. Once this iteration ends, k is added to this set. Hence, the order of outermost loop is important.
- The main idea behind Floyd Warshall's Algorithm is to find all those pairs with distance 0 between them, then all those pairs with distance 1 between them and keep iterating further. So we parallelize the loops in such a way that the value (k in the above code) representing this distance is not affected and no same memory address can be updated by parallel threads. Using OMP_FOR construct in second nested for loop helps us achieve this.

1.3 Input

- The first line of input contains n, the number of nodes in the graph.
- The next line contains n*n values, representing pairwise initial distance between two nodes. Here, value of 1e7 is taken to be infinity, i.e,the occurrence of 1e7 implies that there is no edge between the current pair. Existence of edge (i,j) with weight w implies that the $(n*i+j)^{th}$ and $(n*j+i)^{th}$ value is equal to w. Here is a sample input:

```
5
0 5 2 6 10000000 10000000 0 10000000 10000000 10000000 10000000
    10000000 0 1 5 10000000 10000000 10000000 0 10000000 10000000
    10000000 10000000 10000000 0
```

1.4 Output

- The first line outputs the time taken by program.
- The pairwise distance between nodes can be printed by following the instructions provided at the beginning of code.

1.5 Complexity

- The complexity of the serial program is $\mathbf{O(n^3)}$, where n is the number of nodes in the grammar.
- The complexity of the parallel program (OPENMP) is $\mathbf{O(n^3/no\_threads)}$, where n is the number of nodes in the grammar.
- The number of iterations significantly reduce with decrease in edges due to the continue statement in the second nested loop, which prevents the code from running idle.

1.6 Runtime Comparsion

To evaluate the performance for different cases, I used **three different values for number of nodes** (200, 500 and 1000) and generated **three different test files with number of edges as 500, 1000 and 10000** for each number of node (9 test files in total). I compared the runtime of each test case for serial and parallel programs and generated graphs to visualise the performance.

The images below shows sample input and output for a small test case.

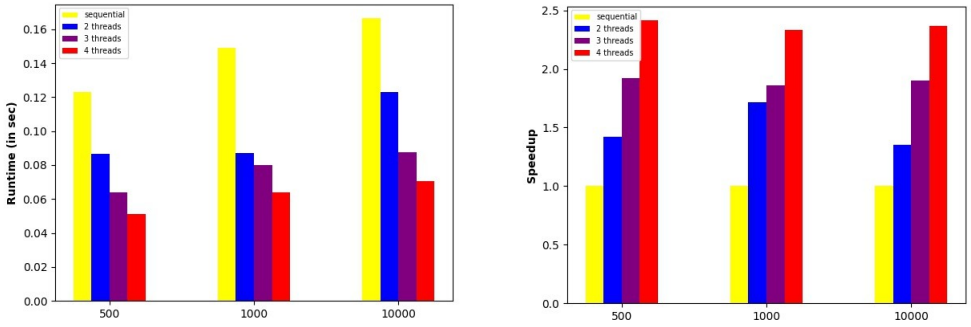
```
> g++ serial.cpp

 ~/Documents/5_sem/cs309-parallel/labs/Lab2
> ./a.out
5
0 5 2 6 1000 1000 0 1000 1000 1000 1000 1000 0 1 5 1000 1000 1000 0 1000 1000 1000 1000 1000 0
Time Taken by serial program: 0.00001 s
0       5       2       3       7
1000    0       1000    1000    1000
1000    1000    0       1       5
1000    1000    1000    0       1000
1000    1000    1000    1000    0
```

Figure 1.1: Serial Program

```
 ~/Documents/5_sem/cs309-parallel/labs/Lab2
> g++ openmp.cpp -fopenmp

 ~/Documents/5_sem/cs309-parallel/labs/Lab2
> ./a.out 2
5
0 5 2 6 1000 1000 0 1000 1000 1000 1000 1000 0 1 5 1000 1000 1000 0 1000 1000 1000 1000 1000 0
Time Taken by openmp program: 0.00023 s
0       5       2       3       7
1000    0       1000    1000    1000
1000    1000    0       1       5
1000    1000    1000    0       1000
1000    1000    1000    1000    0

 ~/Documents/5_sem/cs309-parallel/labs/Lab2
>

 ~/Documents/5_sem/cs309-parallel/labs/Lab2
> ./a.out 3
5
0 5 2 6 1000 1000 0 1000 1000 1000 1000 1000 0 1 5 1000 1000 1000 0 1000 1000 1000 1000 1000 0
Time Taken by openmp program: 0.00037 s
0       5       2       3       7
1000    0       1000    1000    1000
1000    1000    0       1       5
1000    1000    1000    0       1000
1000    1000    1000    1000    0

 ~/Documents/5_sem/cs309-parallel/labs/Lab2
> ./a.out 4
5
0 5 2 6 1000 1000 0 1000 1000 1000 1000 1000 0 1 5 1000 1000 1000 0 1000 1000 1000 1000 1000 0
Time Taken by openmp program: 0.00046 s
0       5       2       3       7
1000    0       1000    1000    1000
1000    1000    0       1       5
1000    1000    1000    0       1000
1000    1000    1000    1000    0
```

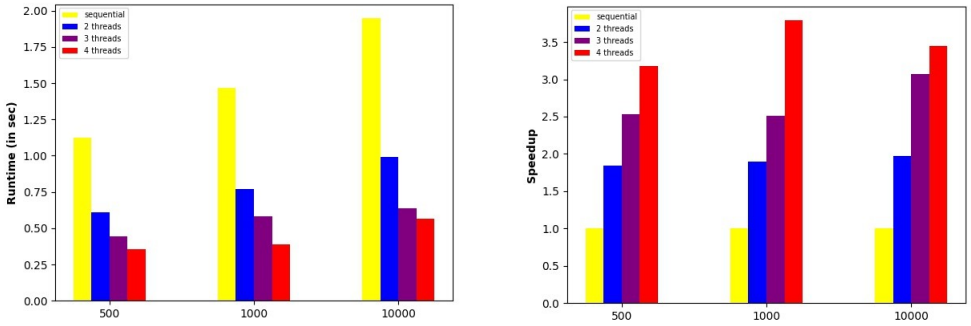
Figure 1.2: OPENMP Program

1.6.1 Keeping the number of Nodes constant



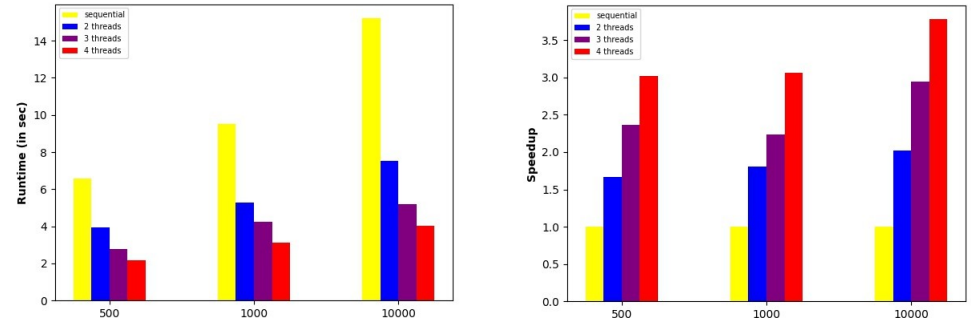
(a) Runtime vs No of edges (b) Speedup vs No of edges

Figure 1.3: Runtime and Speedup Graphs for different number of edges and Different number of processes with **number of nodes=200**



(a) Runtime vs No of edges (b) Speedup vs No of edges

Figure 1.4: Runtime and Speedup Graphs for different number of edges and Different number of processes with **number of nodes=500**

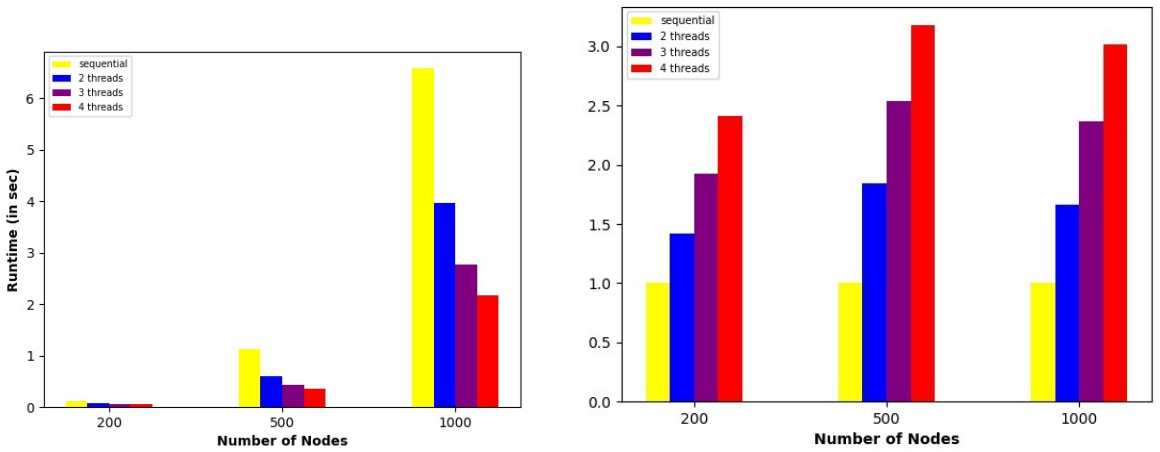


(a) Runtime vs No of edges (b) Speedup vs No of edges

Figure 1.5: Runtime and Speedup Graphs for different number of edges and Different number of processes with **number of nodes=1000**

The above figures represent the changes in runtime and speedup with the change in number of edges when number of nodes are equal to 200, 500 and 1000 respectively.

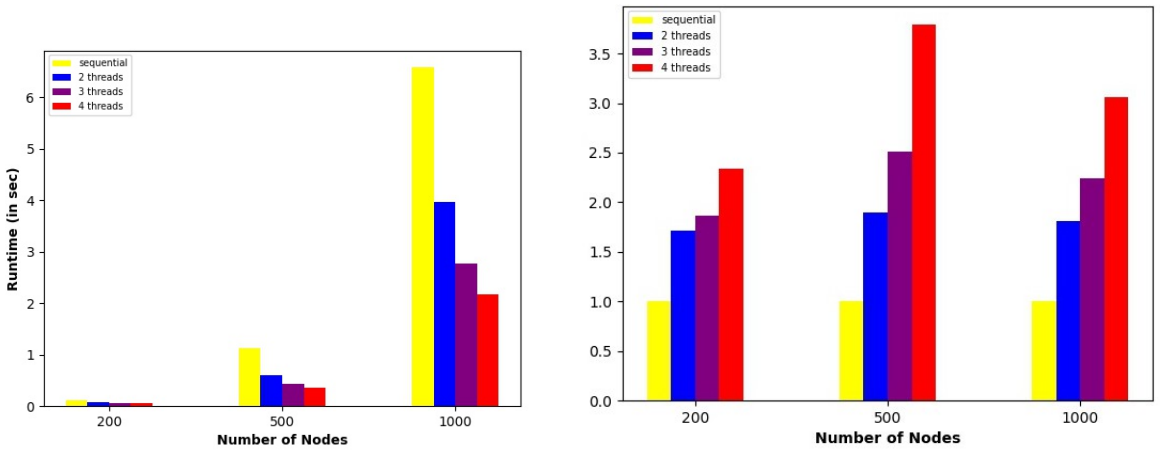
1.6.2 Keeping the number of Edges constant



(a) Runtime vs No of nodes

(b) Speedup vs No of nodes

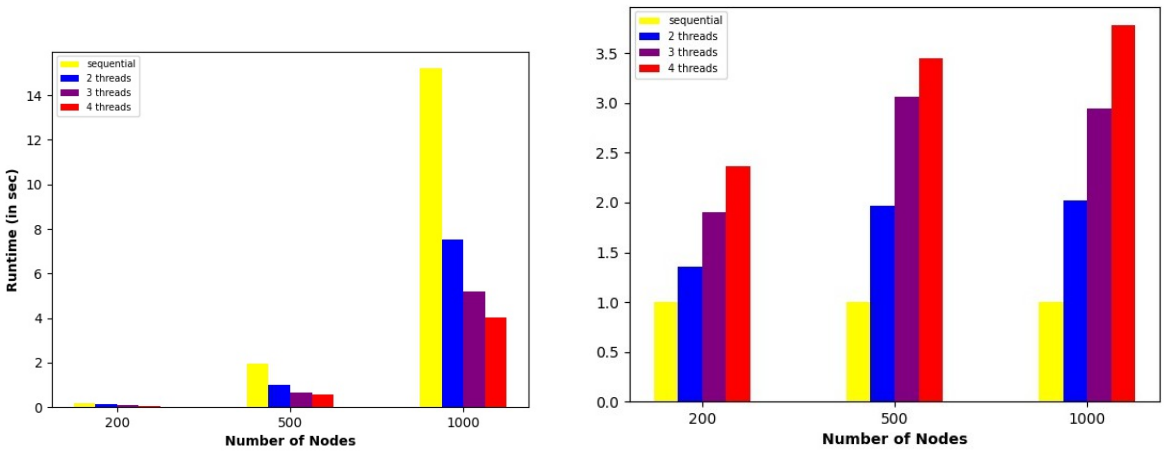
Figure 1.6: Runtime and Speedup Graphs for different number of nodes and Different number of processes with **number of edges=500**



(a) Runtime vs No of nodes

(b) Speedup vs No of nodes

Figure 1.7: Runtime and Speedup Graphs for different number of nodes and Different number of processes with **number of edges=1000**



(a) Runtime vs No of nodes

(b) Speedup vs No of nodes

Figure 1.8: Runtime and Speedup Graphs for different number of nodes and Different number of processes with **number of edges=10000**

The above figures represent the changes in runtime and speedup with the change in number of nodes when number of edges are equal to 500, 1000 and 10000 respectively.

1.7 Conclusion

We conclude that the runtime reduces with a increase in the number of threads. We see a significant increase in the runtime with increase in the number of nodes. Although the algorithm's complexity does not depend on the number of edges, but the 'if condition' checking whether $matrix[i*n+k] = inf$ or $matrix[k*n+j] = inf$ reduces the number of update operations and helps in the speedup of upto 4 times. As the number of edges increase, both runtime and speedup also increase.

1.8 Implementation

```
// Uncomment lines 54-59 to output the pairwise distance

#include<bits/stdc++.h>
#include<omp.h>

const int inf=1e7;
#define endl "\n"
int size;

using namespace std;

int n;
vector<int>matrix;

void Floyd_Warshall(){
  for(int k=0;k<n;k++){

    #pragma omp parallel for num_threads(size) collapse(2)

      for(int i=0;i<n;i++){

        for(int j=0;j<n;j++){

          int v=matrix[i*n+k];
          if(v==inf) continue;
          if(matrix[k*n+j]==inf) continue;
          int val=v+matrix[k*n+j];

          if(val<matrix[i*n+j]) matrix[i*n+j]=val;

        }
      }
  }
}

int main(int argc, char **argv){

  size = atoi(argv[1]);

  cin>>n;

  matrix.resize(n*n); //Storing edges in 1D array of size n*n

  for(int i=0;i<n*n;i++) cin>>matrix[i];
```

```
45
46    double start_time=omp_get_wtime();
47
48    Floyd_Warshall();
49
50    double final_tot=omp_get_wtime()-start_time;
51
52    cout<<fixed<<setprecision(5) <<"Time Taken by openmp program: "<<
        final_tot <<" s "<<endl;
53
54    // for(int i=0;i<n*n;i++){
55
56    //   cout<<matrix[i]<<"\t";
57    //   if((i+1)%n==0)cout<<endl;
58
59    // }
60
61    return 0;
62 }
63
64
```