DA1
All Pairs Shortest Path Problem in OpenMP

-CED17I023
J.Veeren Chandrahas

Table of Contents

1. Introduction:

The shortest path problem is about finding a path between two nodes in a graph such that the path cost is minimized. One example of this problem could be finding the fastest route from one city to another by car, train or airplane.
The Floyd-Warshall algorithm is an algorithm that solves this problem. It works for weighted graphs with positive or negative weights but not for graphs with negative cycles.
It works by comparing all possible paths between all vertex pairs in the graph. A version of the algorithm implemented in the C language can be seen in the figure below.

```c
void ST_APSP(int *mat, const size_t N)
{
    for (int k = 0; k < N; k++)
        for (int i = 0; i < N; i++)
            for (int j = 0; j < N; j++)
            {
                int i0 = i * N + j;
                int i1 = i * N + k;
                int i2 = k * N + j;
                if (mat[i1] != -1 && mat[i2] != -1)
                {
                    int sum = (mat[i1] + mat[i2]);
                    if (mat[i0] == -1 || sum < mat[i0])
                        mat[i0] = sum;
                }
            }
}
```

The notations used in this figure; "k", "i", "j", "N" and "mat[...]" will be used throughout the report.
For each k all of the current values (mat[i*N,j]) of the matrix is compared to the sum of two other values in the matrix: mat[k*N,j] + mat[i*N,k]. Once the outer loop has run N times all paths between all vertex pairs have been compared.
The purpose of the lab was to parallelize this algorithm with the use of MPI. This report will show the results of the parallel

version that was implemented and explain the design of the parallel algorithm.

2. Results:

To measure the performance of our algorithm, we performed various tests by varying the number of processors and the size of the initial matrix. For simplicity, we have use values of N (N=1000) which are divisible by the numbers of processors (p=2,4,6,8,10,12,14,16,18,20,24).

2.1. Total time:

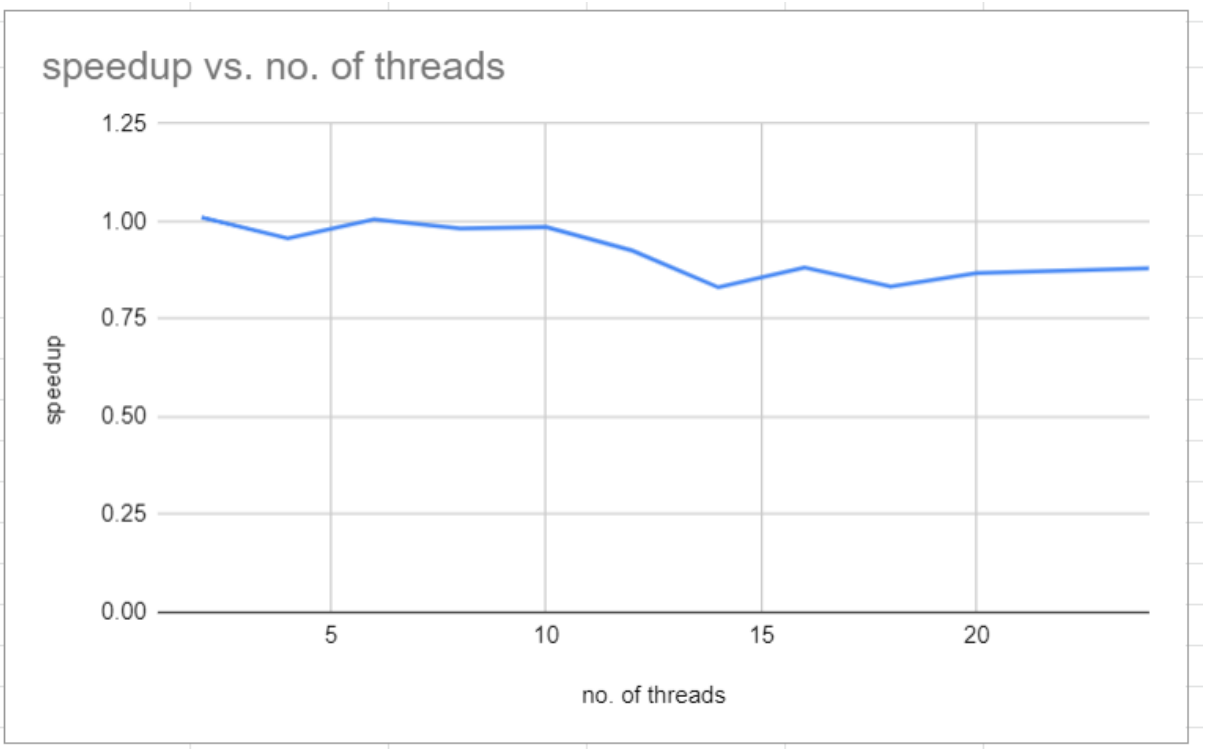
2.1.1. Table:

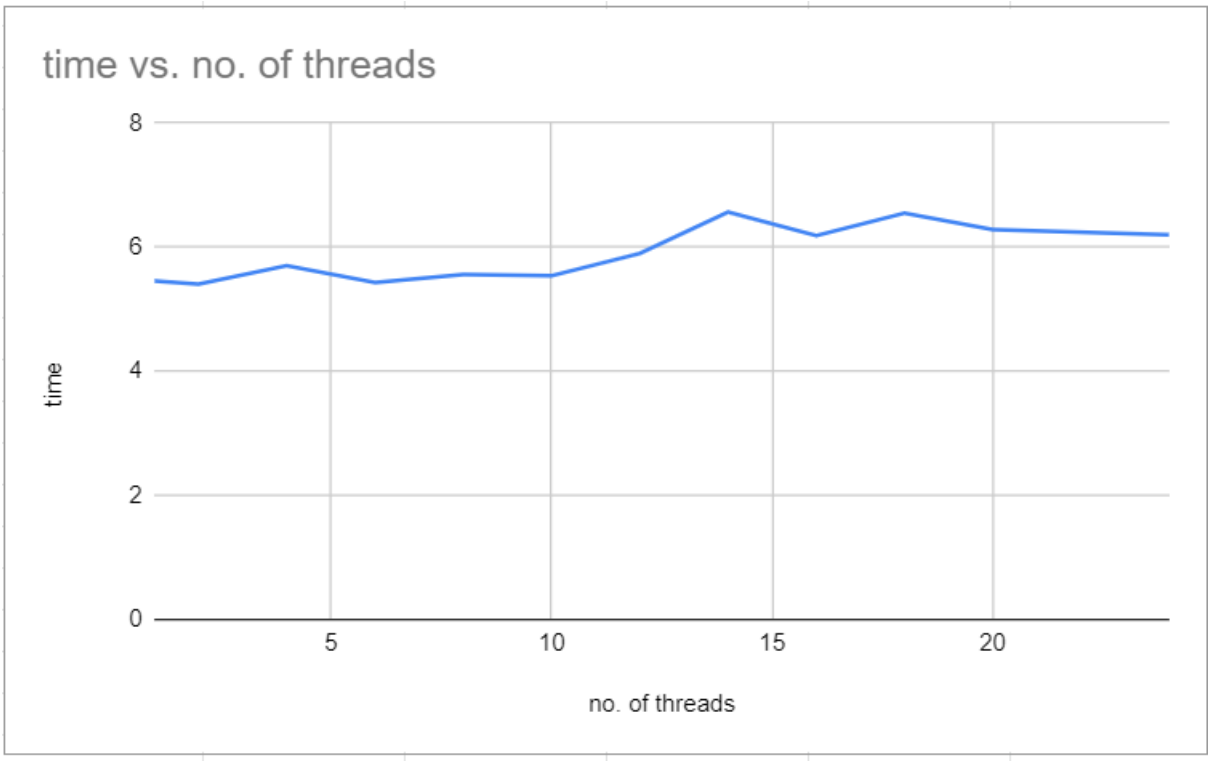
This table summarizes the total execution time of sequential and parallel programs based on the numbers of processors.

no. of processor	time	speedup	parallel fraction
1	5.458024		
2	5.404284	1.009943963	0.01969210835
4	5.703328	0.9569893227	-0.05992498384
6	5.42943	1.005266483	0.00628667078
8	5.560977	0.9814865266	-0.02155735692
10	5.538498	0.9854700679	-0.01638240425
12	5.899237	0.9252084634	-0.08818636062
14	6.566482	0.8311945422	-0.2187099214
16	6.186035	0.8823137923	-0.1422758615
18	6.549778	0.8333143505	-0.2117936498
20	6.287106	0.8681297882	-0.1598963095
24	6.202211	0.8800126278	-0.1422754749

2.1.2. Graph:

This graph shows the total execution time of programs depending on the size of the initial matrix, N. Moreover, each curve is associated with a different number of processors.





2.1.3. Analyse:

Thanks to this information, we can see that the execution time fluctuates as the number of processors increases.

2.2. Speedup:

Speed Up is defined as the ratio of the serial run time of the best sequential algorithm for solving a problem to the time taken by the parallel algorithm to solve the same problem on p processors.

2.2.1. Table:

This table summarizes the speedup of parallel programs based on the numbers of processors and the size of the initial matrix. This allows us to compare the speed of execution
of a parallel program with a sequential program.

Time taken for 1 thread= 5.458024
Time taken for 6 thread= 5.404284
Speedup =T (1)/T (6) = 1.005266483

Parallelization Factor (f) = (1 - T (6)/T (1))/ (1 - (1/6)) = 0.009943963

3. Observations:

For each certain iteration (k) of the algorithm the calculations are independent. Calculating the value of mat [i*N, j] will not affect the calculation of mat [i'*N, j'] for a specific k. But the values calculated for the following iterations of k will depend upon the previous iterations. This means that the iterations of k have to be calculated in sequence but the values of the matrix within that iterations can be calculated in sequence.

4.Appendix:

```c
int main(int argc, char **argv)
{
    struct timeval tv1, tv2;

    if (argc != 2)
    {
        printf("Usage: test {N}\n");
        exit(-1);
    }
    //generate a random matrix.
    size_t N = atoi(argv[1]);
    int *mat = (int *)malloc(sizeof(int) * N * N);
    GenMatrix(mat, N);
    //compute the reference result.
    int *ref = (int *)malloc(sizeof(int) * N * N);
    memcpy(ref, mat, sizeof(int) * N * N);

    gettimeofday(&tv1, NULL);
    ST_APSP(ref, N);
    gettimeofday(&tv2, NULL);
    double tseq = (tv2.tv_sec - tv1.tv_sec) * 1000000 + tv2.tv_usec - tv1.tv_usec;
    printf("Elasped time = %f usecs\n", tseq);

    //compute your results
    int *result = (int *)malloc(sizeof(int) * N * N);
    memcpy(result, mat, sizeof(int) * N * N);
    gettimeofday(&tv1, NULL);
    //parallel algorithm
    MT_APSP(result, N);
    gettimeofday(&tv2, NULL);
    double tp = (tv2.tv_sec - tv1.tv_sec) * 1000000 + tv2.tv_usec - tv1.tv_usec;
    printf("Elasped time parallel = %f usecs\n", tp);

    //compare your result with reference result
    if (CmpArray(result, ref, N * N))
        printf("Your result is correct.\n");
    else
        printf("Your result is wrong.\n");
}
```

```
#pragma omp parallel for private(i, j) schedule(static)
        for (i = 0; i < N; i++)
        {
            // Calculation of i in the first loop, independent of j
            int i1 = i * N + k;
            // Compute rows of the matrix with the row k
            for (j = 0; j < N; j++)
            {
                int i0 = i * N + j;
                int i2 = k * N + j;
                if (mat[i1] != -1 && mat[i2] != -1)
                {
                    int sum = (mat[i1] + mat[i2]);
                    if (mat[i0] == -1 || sum < mat[i0])
                        mat[i0] = sum;
                }
            }
        }
    }
}
```