COMP90025 Parallel and Multicore Computing
Project 1A – Diameter of Graph
Lu Chen Yiru Pan

Introduction

This project aims to parallelize sequential Floyd Warshall algorithm to minimize running time of the program. In the experiment, SPARTAN HPC has been used, the project requested one node in cloud mode, and allocated CPU number from 1 to 8, problem sizes included 5000, 1000, 100, 10 vertices graph. There are three comparison parts in the program.

Part 1 Parallelize File Reading

At first, we tried to parallelize file-reading block. As shown in Figure-1, "in_file" is a global variable which points to the raw file stream. We hoped all threads can execute the while block in parallel. We then tested the running time of this algorithm using 8 processors against sequential reading.

```
fscanf(in_file,"%d", &nodesCount);

int a, b, c;
int faultFlag;
#pragma omp parallel firstprivate(a,b,c)
{
    while(fscanf(in_file,"%d %d %d", &a, &b, &c)!= EOF){
    {
        if ( a > nodesCount || b > nodesCount){
        faultFlag = -1;
        }
        distance[a][b]=c;
    }
    }

}

if(faultFlag == -1){
    printf("Vertex index out of boundary.");
        return -1;
}
```

Figure-1 Code for parallel reading of a file

We observed that parallel reading is even worse than the sequential reading algorithm. Possible reasons are as follows. All threads access the while loop almost simultaneously, but only one thread can read the file at one time. After that, this thread enters the while block. At this time, the file will be accessed by another thread to read the next line. Therefore, in general, the file is read sequentially. But the executions within while loop are performed in parallel. However, the time it saves cannot compensate for openMP's overhead. Therefore, the performance is not improved overall.

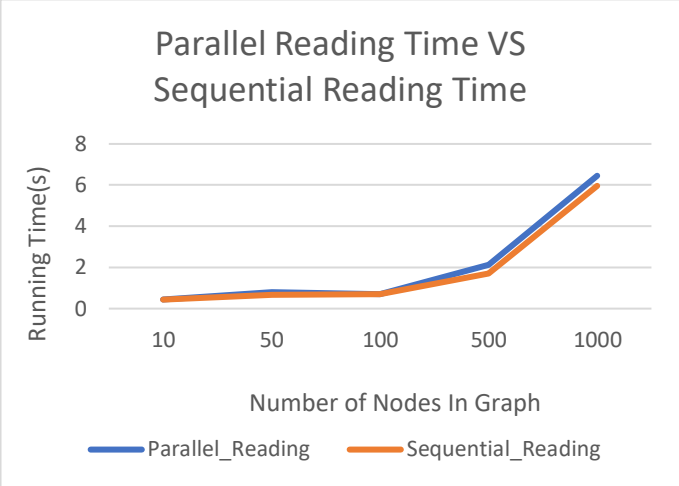

Figure-2

Part 2 Parallelize 1 loop in static and dynamic model

There are 3 for loops in Floyd-Warshall algorithm, the first for loop cannot be parallelized, because following matrix depends on previous matrix. When we design the parallel program, we need to make sure the iterations are independent. In our experiment, we implemented parallel on the second for (single loop), run it in different schedule models (static and dynamic) and different CPU numbers.

- Static

```
//Floyd-Warshall
int k;
for (k=1;k<=nodesCount;++k){
    int i;
    #pragma omp parallel for schedule(static)
    for (i=1;i<=nodesCount;++i){
        if (distance[i][k]!=NOT_CONNECTED){
            int j;
            for (j=1;j<=nodesCount;++j){
                if (distance[k][j]!=NOT_CONNECTED &&
                    (distance[i][j]==NOT_CONNECTED ||
                        distance[i][k]+distance[k][j]<distance[i][j])){
                    distance[i][j]=distance[i][k]+distance[k][j];
                }
            }
        }
    }
}
```

Figure-3 code block

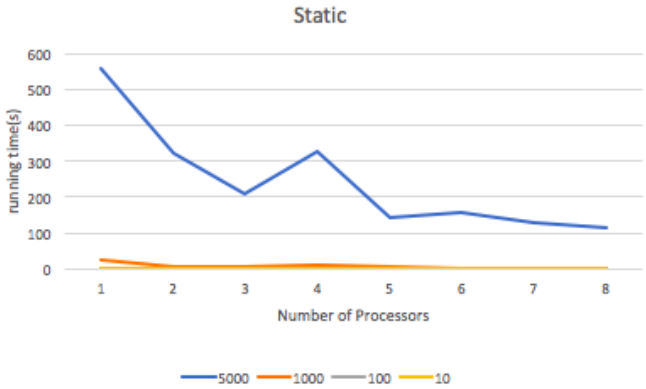

Figure-4 Static Model

From the chart, we can find with the problem size increasing, especially for 5000 vertices, there is a trend that more threads can expedite the program. This trend is not obvious for small problem size like 100 or 10. The max speed up for 5000 vertices' scenario is $\frac{T_s}{T_p} = \frac{560}{116} \approx 4.8$, so we can get the conclusion that in static model, for large problem size, with more threads, the program become faster.

- Dynamic

```
//Floyd-Warshall
int k;
for (k=1;k<=nodesCount;++k){
    int i;
    #pragma omp parallel for schedule(dynamic)
    for (i=1;i<=nodesCount;++i){
        if (distance[i][k]!=NOT_CONNECTED){
            int j;
            for (j=1;j<=nodesCount;++j){
                if (distance[k][j]!=NOT_CONNECTED &&
                    (distance[i][j]==NOT_CONNECTED ||
                        distance[i][k]+distance[k][j]<distance[i][j])){
                    distance[i][j]=distance[i][k]+distance[k][j];
                }
            }
        }
    }
}
```

Figure-5 code block

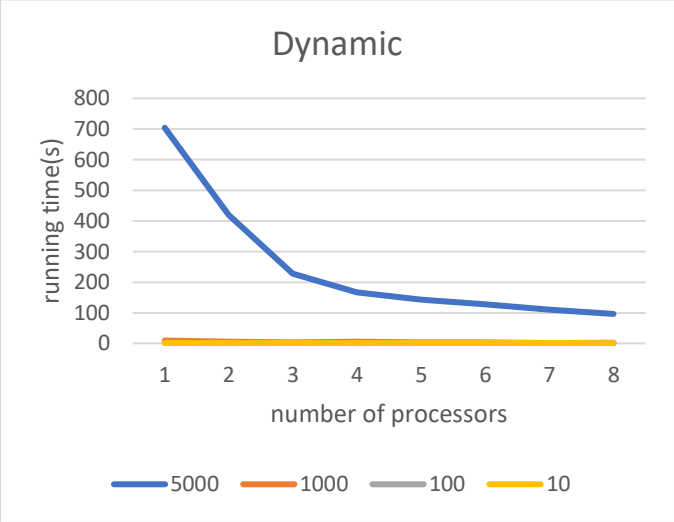

Figure-6 Dynamic model

From chart, dynamic has the obvious decline trend when problem size is 5000. For problem size like 1000, 100 and 10, the speed up is not obvious. The max speed up for 5000 vertices' scenario is $\frac{T_s}{T_p} = \frac{704}{96} \approx 7.3$, which is better than static model.

- Static and Dynamic (5000 vertices)

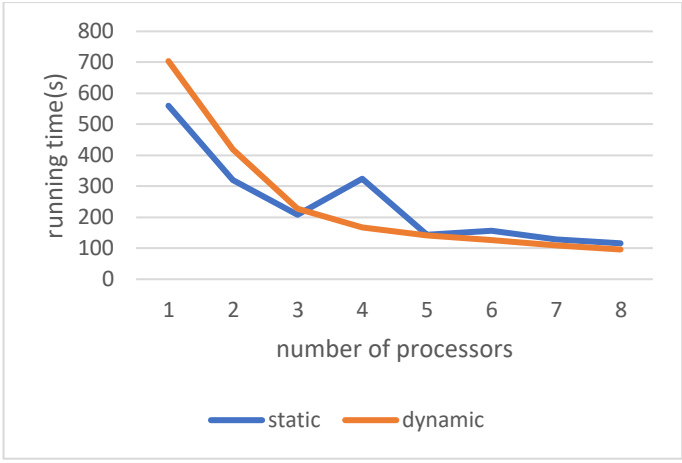

Figure-7 Static vs Dynamic

From the chart, we can find that when CPU number is greater than 4, dynamic performs better. Thus, we decide to divide our parallel program into two parts, if thread number is great than 4, we run program in dynamic model. Conversely, if thread number is smaller than 4, we run program in static model.

Part 3 Parallelize 1 loop or 2 loops

```
//Floyd-Warshall
int k;
for (k=1;k<=nodesCount;++k){
    int i;
    #pragma omp parallel for schedule(static)
    for (i=1;i<=nodesCount;++i){
        if (distance[i][k]!=NOT_CONNECTED){
            int j;
            for (j=1;j<=nodesCount;++j){
                if (distance[k][j]!=NOT_CONNECTED &&
                    (distance[i][j]==NOT_CONNECTED || distance[i][k]+distance[k][j]<distance[i][j])){
                    distance[i][j]=distance[i][k]+distance[k][j];
                }
            }
        }
    }
}
```

Figure-8 parallelize the outer loop

```
//Floyd-Warshall
int k;
for (k=1;k<=nodesCount;++k){
    int i,j;
    #pragma omp parallel for private(j) collapse(2) schedule(static)
    for (i=1;i<=nodesCount;++i){
        {
            for (j=1;j<=nodesCount;++j){
                if ((distance[i][k]!=NOT_CONNECTED)&&
                    (distance[k][j]!=NOT_CONNECTED &&
                     (distance[i][j]==NOT_CONNECTED || distance[i][k]+distance[k][j]<distance[i][j]))){
                    distance[i][j]=distance[i][k]+distance[k][j];
                }
            }
        }
    }
}
```

Figure-9 parallelize two loops (OpenMP collapse)

Since the outermost loop of Floyd-Warshall algorithm has to be executed sequentially, we only consider parallelizing the inner two loops. Due to the openMP overheads, we were not sure whether parallelizing both loops (Figure-9) will be better than simply parallelizing the second loop (Figure-8). Therefore, we tested them respectively using different numbers of processors (threads) on different sizes of graphs and recorded the running times. Here are the results.

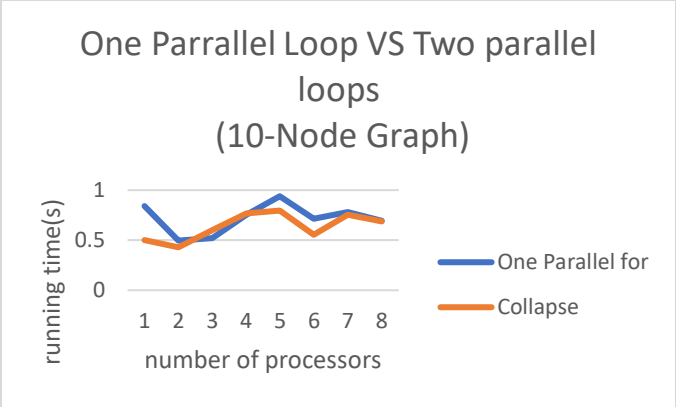

Figure-10

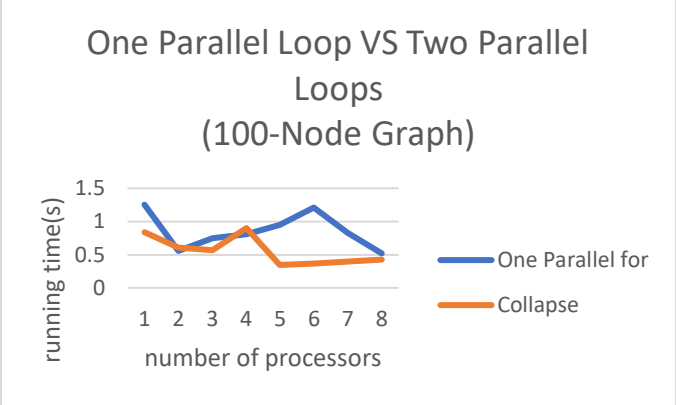

Figure-11

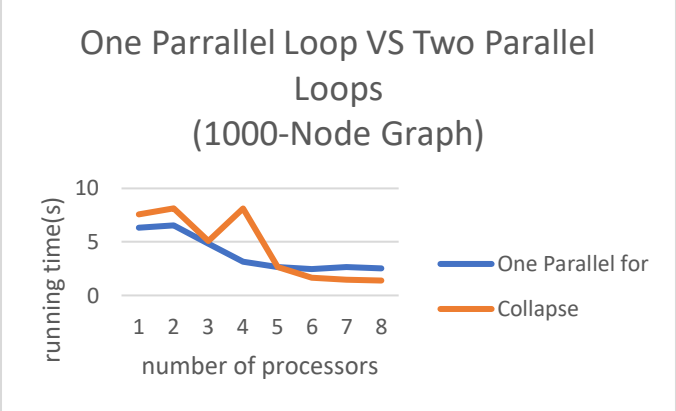

Figure-12

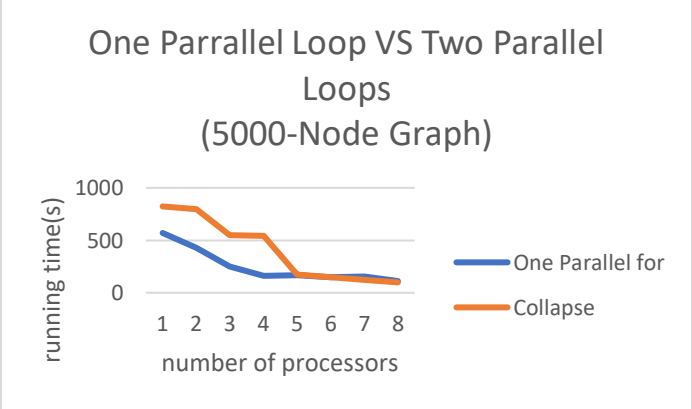

Figure-13

It's shown clearly from the above figures that when the number of processors is very small like 1 or 2, one parallel loop wins. However, when the number of processors approximately exceeds 4, two parallel loops (omp-collapse) wins. Also, when the size of graph becomes bigger, especially when there are 1000 or 5000 nodes, the efficiency of two parallel loops becomes more obvious.

Analysis on the results is as follows. When we collapse the loops, openMP turns the nested loops into a big loop and each thread deals with a certain chunk of iterations. However, the single parallel-for only divides the outer iterations. Therefore, omp-collapse can divide the workload more evenly for each thread. However, when there are too few processors, the overheads of omp-collapse will weigh its advantages because it takes some time to calculate and manage more iterations. Hence, when the number of processors is smaller, single parallel-for turns out to be better.

Conclusion

Based on the experimental results from Part 2 and Part 3, we decided to apply the following strategy (Figure-14) to ensure the best performance. When there are less than or equal to 4 processors, we choose to use one parallel-for in static model. Otherwise, we parallelize two loops via omp-collapse in dynamic model.

```
//Floyd-Warshall
int k;
for (k=1;k<=nodesCount;++k){
    int i,j;
    // if else
    if(threadNum > 4){
        #pragma omp parallel for private(j) collapse(2) schedule(static)
        for (i=1;i<=nodesCount;++i){
            {
                for (j=1;j<=nodesCount;++j){
                    if ((distance[i][k]!=NOT_CONNECTED)&&
                        (distance[k][j]!=NOT_CONNECTED &&
                         (distance[i][j]==NOT_CONNECTED || distance[i][k]+distance[k][j]<distance[i][j]))){
                        distance[i][j]=distance[i][k]+distance[k][j];
                    }
                }
            }
        }
    }
    else{
        #pragma omp parallel for private(j)schedule(static)
        for (i=1;i<=nodesCount;++i){
            {
                for (j=1;j<=nodesCount;++j){
                    if ((distance[i][k]!=NOT_CONNECTED)&&
                        (distance[k][j]!=NOT_CONNECTED &&
                         (distance[i][j]==NOT_CONNECTED || distance[i][k]+distance[k][j]<distance[i][j]))){
                        distance[i][j]=distance[i][k]+distance[k][j];
                    }
                }
            }
        }
    }
}
```

Figure-14 Final parallel strategy