COMP90025 Project 1B: OpenMP and Diameter of Graph

San Kho Lin (829463) — sanl1@student.unimelb.edu.au

1 Introduction

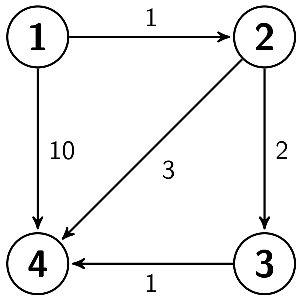
In this project, I am going to experiment OpenMP programming on a given serial program which is computing the diameter of a graph.

2 Shortest Path

To begin with, I first study on finding the Shortest path problem[1] for a given graph. Generally, there are two well-known approaches:

- Single Source Shortest Path (SSSP)
- All Pairs Shortest Path (APSP)

For this project, I experiment with *Floyd-Warshall* algorithm which is APSP and usually express in dynamic programming technique. Suppose given graph G as follow.



Then, the corresponding adjacency matrix is:

$$D_0 = \begin{bmatrix} 0 & 1 & \infty & 10 \\ \infty & 0 & 2 & 3 \\ \infty & \infty & 0 & 1 \\ \infty & \infty & \infty & 0 \end{bmatrix}$$

The characteristics of *Floyd-Warshall* algorithm are as follow.

- Finding shortest path in a weighted graph
- Allow positive or negative edge weight
- Do not allow negative cycle - no loop
- Adjacency matrix as initial input - D_0
- Contains 3-loops and has time complexity of $O(N^3)$
- Since adjacency matrix D_0 to given number of vertices N, each iteration go through intermediary vertex k for the path between i to j vertices.

Listing **Algorithm 1** pseudocode express the steps of sequential *Floyd-Warshall* algorithm for finding shortest path in a graph. In nutshell, the fundamental concept of the algorithm is to determine whether a path going from vertex V_i to vertex V_j via vertex V_k is *shorter than* the best-known path from vertex V_i to vertex V_j [3].

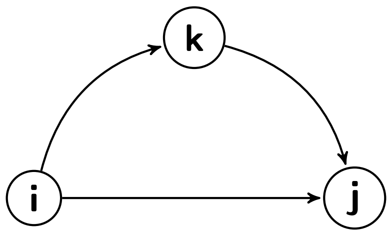


Illustration: The fundamental operation in Floyd-Warshall sequential shortest-path algorithm

The following is the the resultant matrix D_4 after N iterations using *Floyd-Warshall* algorithm.

$$D_4 = \begin{bmatrix} 0 & 1 & 3 & 4 \\ \infty & 0 & 2 & 3 \\ \infty & \infty & 0 & 1 \\ \infty & \infty & \infty & 0 \end{bmatrix}$$

The diameter of a graph is the maximum eccentricity of any vertex in the graph[2]. In this case, the diameter of graph G is 4.

[1]https://en.wikipedia.org/wiki/Shortest_path_problem

[2]https://en.wikipedia.org/wiki/Distance_(graph_theory)

3 Parallel Porting

Parallel algorithm is based on *Ian Foster* book, *Chapter 3.9 Case Study: Shortest-Path Algorithms* [3]. It is based on one-dimensional row-wise domain decomposition of the distance matrix D. In listing **Algorithm 2**, the parallel portion of code start at each k-th step, the i-th row of task will distribute into each processor. **Listing 1** shows the OpenMP code.

4 Observation

Compare to Project 1A's Mandelbrot Set, the Floyd-Warshall algorithm can do both data parallel and task computation parallel. In Mandelbrot Set, the computation is dependant of prior sequence condition. Though I only able to attempt the row-wise domain decomposition for at most N processors in this project, I also read that the *Flody-Warshall* algorithm can be further speed up by Block-wise domain decomposition which can utilise N^2 processors and Hypercube communication pattern on MPI parallel architecture [1] [3] [4]. In **Figure 4, 5 and 6**, the load balance distributed reasonably well by default OpenMP work sharing constructs. Even though this can be further enhanced by using heuristics approach like Block data partitioning on the matrix D for OpenMP architecture [2].

5 Conclusion

Figure 1, 2 and 3 show the outcome of wall-clock elapse times for number of processors by number of vertices. I run OpenMP jobs on VLSCI[3] Snowy cluster through Slurm[4] queue. The difference of speed up can be seen by using higher number of vertices. When a graph has less than 100 vertices, the speed up between serial and parallel is diminishing.

To conclude, the **Table 1** breakdown the program run of 4096 vertices on serial to parallel schedule *dynamic* with number of processors 2, 4, 8, 16 and 32. And the **Plot 1** visualize the speedup curve of the table.

$$S(p_{32}) = \frac{T_s}{T_p} = \frac{49.070}{3.618} \approx 13.5627$$

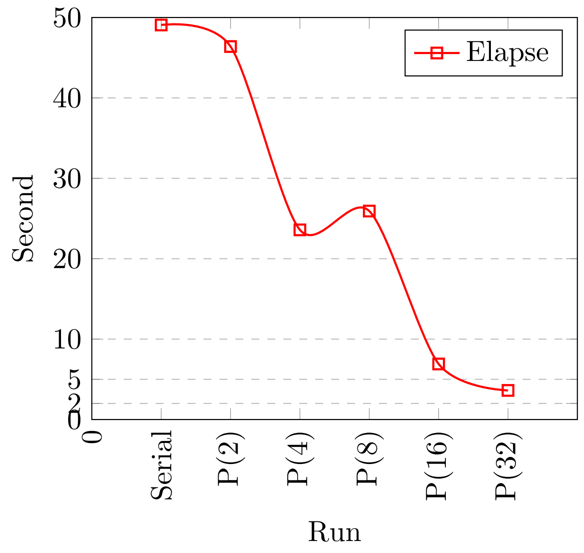
T_s = Execution time using one processor
T_p = Execution time using 32 processors
$LoadFactor$ = 4096 vertices

[3]Victorian Life Sciences Computation Initiative
[4]http://slurm.schedmd.com/

Table 1: Speedup Breakdown

Run	Elapse
Serial	49.070
P(2)	46.389
P(4)	23.588
P(8)	25.924
P(16)	6.908
P(32)	3.618

Plot 1: Speedup Plot



References

[1] Ananth Grama et al. *Introduction to Parallel Computing, Second Edition*. International series of monographs on physics. Addison Wesley, 2003. ISBN: 0-201-64865-2.

[2] Gayathri Venkataraman et al. "A blocked all-pairs shortest-paths algorithm". In: *Journal on Experimental Algorithmics* (2003).

[3] Ian Foster. *Designing and Building Parallel Programs*. 1995. URL: `http://www.mcs.anl.gov/~itf/dbpp/`.

[4] Quinn Michael J. *Parallel Programming in C with MPI and OpenMP*. International series of monographs on physics. McGraw Hill, 2003. ISBN: 007-282256-2.

6 Appendix

Algorithm 1 Floyd-Warshall Algorithm

```
procedure SEQUENTIAL_FW
begin
    if $i = j$ then
        $D_{ij}(0) = 0$
    if $i \neq j$ and edge exists then
        $D_{ij}(0) = length((V_i, V_j))$
    $D_{ij}(0) = \infty$ otherwise
    for $k = 1$ to $N$ do
        for $i = 1$ to $N$ do
            for $j = 1$ to $N$ do
                $D_{ij}^{(k)} = min(D_{ij}^{(k-1)}, D_{ik}^{(k-1)} + D_{kj}^{(k-1)})$
    $S = D(N)$
end
```

Algorithm 2 Floyd-Warshall Parallel Algorithm

```
procedure PARALLEL_FW
begin
    if $i = j$ then
        $D_{ij}(0) = 0$
    if $i \neq j$ and edge exists then
        $D_{ij}(0) = length((V_i, V_j))$
    $D_{ij}(0) = \infty$ otherwise
    for $k = 1$ to $N$ do
parallel for
        for $i$ = local_i_start to local_i_end do
            for $j = 1$ to $N$ do
                $D_{ij}^{(k)} = min(D_{ij}^{(k-1)}, D_{ik}^{(k-1)} + D_{kj}^{(k-1)})$
    $S = D(N)$
end
```

```
void floydwarshall() {
    int k, i, j;
    //Floyd-Warshall
    for (k=1;k<=nodesCount;++k){
        #pragma omp parallel
        {
            #ifdef STATIC
            #pragma omp for private(j) schedule(static)
            #elif defined GUIDED
            #pragma omp for private(j) schedule(guided)
            #else
            #pragma omp for private(j) schedule(dynamic) // default
            #endif
            for (i=1;i<=nodesCount;++i){
                if (distance[i][k]!=NOT_CONNECTED){
                    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
    ];
                        }
                    }
                }
            }

        }
    }
}
```

Listing 1: OpenMP Parallel Floyd-Warshall

Figure 1: Elapse time - Dynamic Schedule

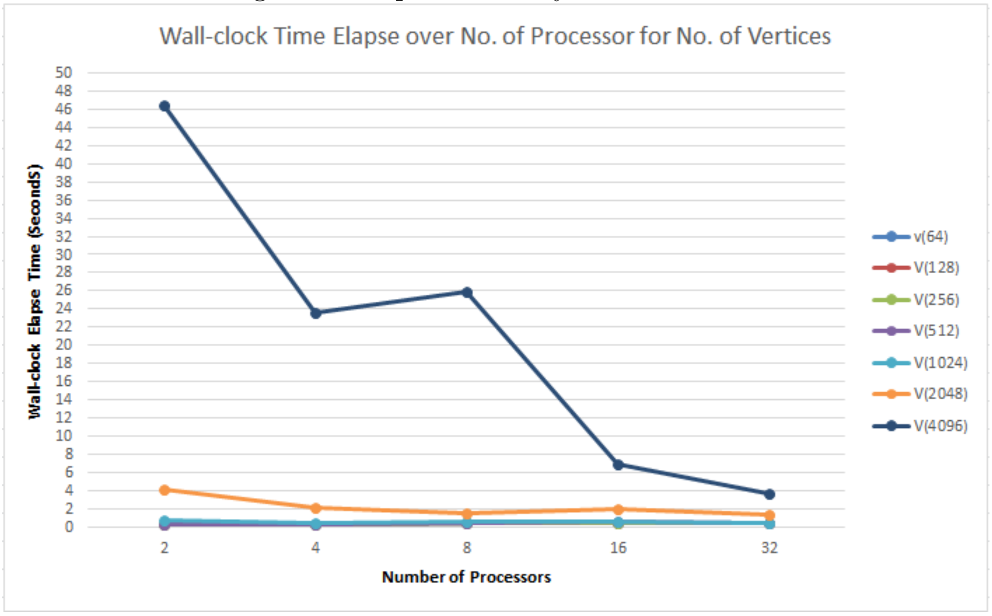


Figure 2: Elapse time - Guided Schedule

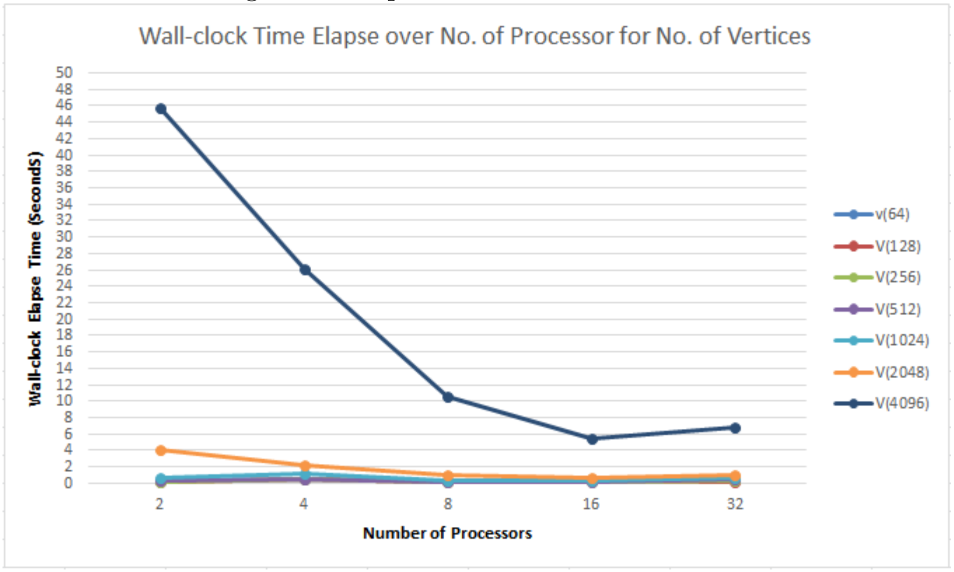


Figure 3: Elapse time - Static Schedule

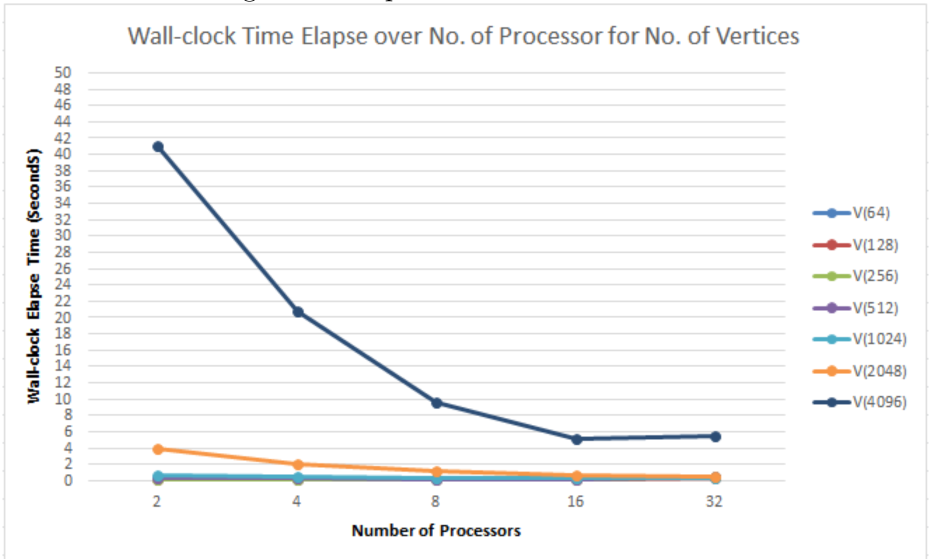


Figure 4: Intel VTune CPU Usage Timeline — schedule(dynamic) Default

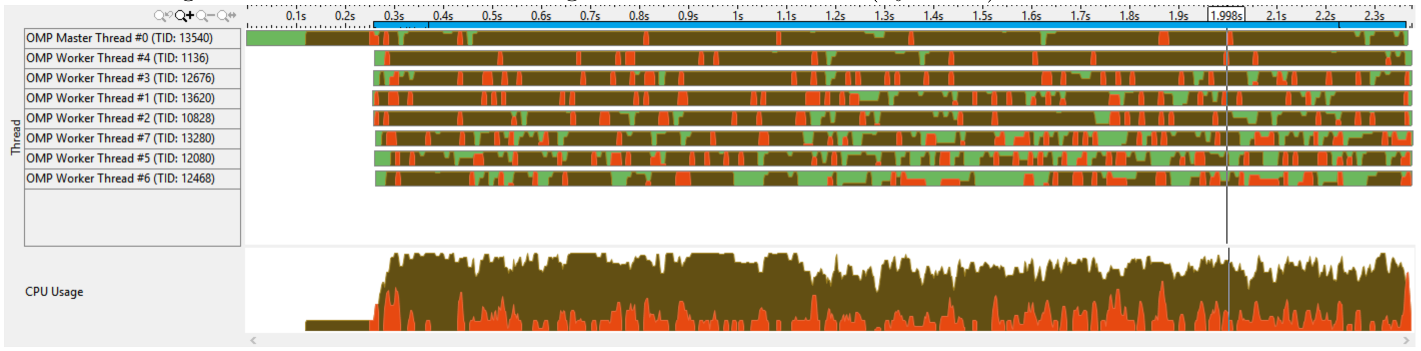

OMP Master Thread #0 (TID: 13540)
OMP Worker Thread #4 (TID: 1136)
OMP Worker Thread #3 (TID: 12676)
OMP Worker Thread #1 (TID: 13620)
OMP Worker Thread #2 (TID: 10828)
OMP Worker Thread #7 (TID: 13280)
OMP Worker Thread #5 (TID: 12080)
OMP Worker Thread #6 (TID: 12468)
Thread
CPU Usage
1.998s

Figure 5: Intel VTune CPU Usage Timeline — schedule(static)

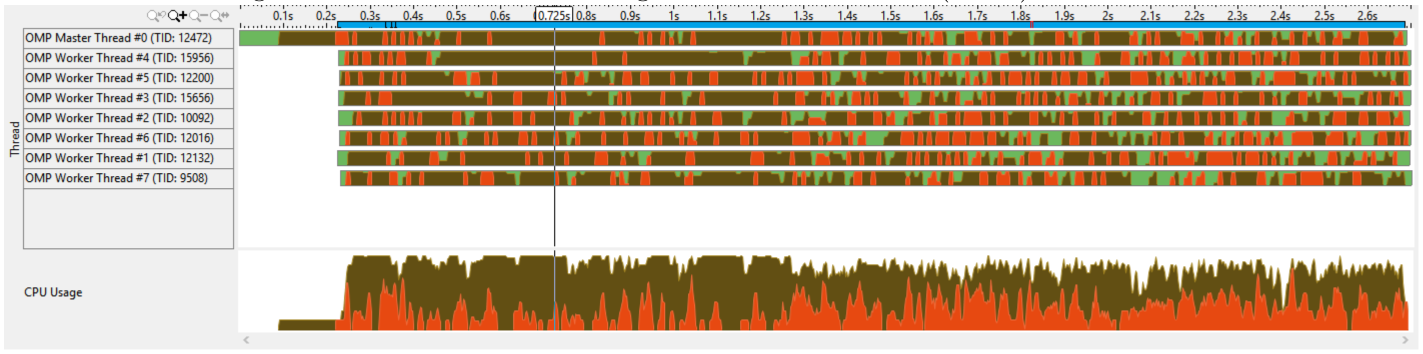
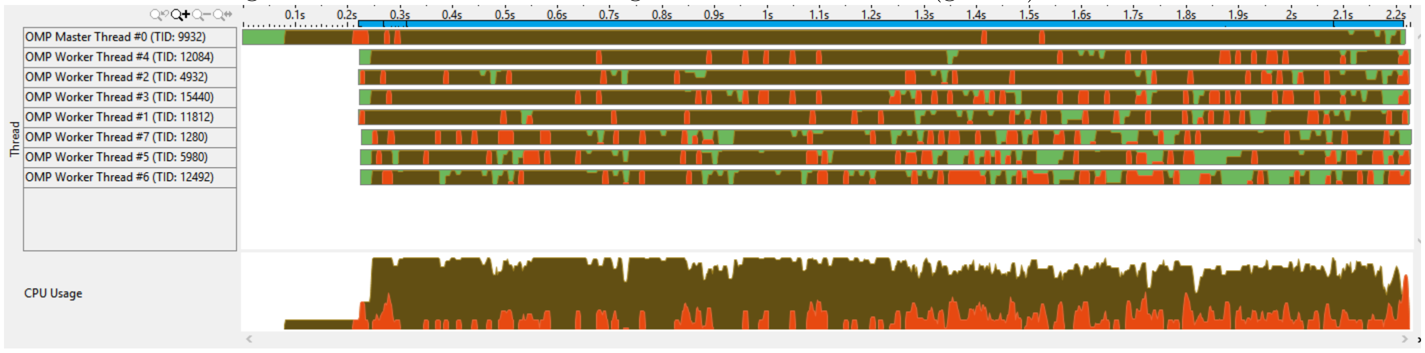

OMP Master Thread #0 (TID: 12472)
OMP Worker Thread #4 (TID: 15956)
OMP Worker Thread #5 (TID: 12200)
OMP Worker Thread #3 (TID: 15656)
OMP Worker Thread #2 (TID: 10092)
OMP Worker Thread #6 (TID: 12016)
OMP Worker Thread #1 (TID: 12132)
OMP Worker Thread #7 (TID: 9508)
Thread
CPU Usage
0.725s

Figure 6: Intel VTune CPU Usage Timeline — schedule(guided)


OMP Master Thread #0 (TID: 9932)
OMP Worker Thread #4 (TID: 12084)
OMP Worker Thread #2 (TID: 4932)
OMP Worker Thread #3 (TID: 15440)
OMP Worker Thread #1 (TID: 11812)
OMP Worker Thread #7 (TID: 1280)
OMP Worker Thread #5 (TID: 5980)
OMP Worker Thread #6 (TID: 12492)
Thread
CPU Usage