COMP90025 Project 1D: MPI and Diameter of Graph

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

1 Introduction

In this project, I am going to experiment MPI parallel programming on a given serial program that determine the diameter of a given graph.

2 Related Work

As in Project 1B, it is the same problem where a random weighted directed graph is given and the program has to find the shortest path possible. Then after the diameter of the graph is the maximum eccentricity of any vertex in the graph. In the given serial program, it uses *All Paris Shortest Path* approach and the implementation is *Floyd-Warshall* algorithm. In Project 1B parallel implementation, I experimented with OpenMP for *Floyd-Warshall* algorithm. The parallel approach is using Row-wise Decomposition [1]. It is relatively easy to program the row decomposition approach with OpenMP. However, it is quite challenging to implement the similar row division approach in MPI environment.

3 Parallel Design

According to Foster article[1], there are two approaches to parallize the *Floyd* algorithm as follow:

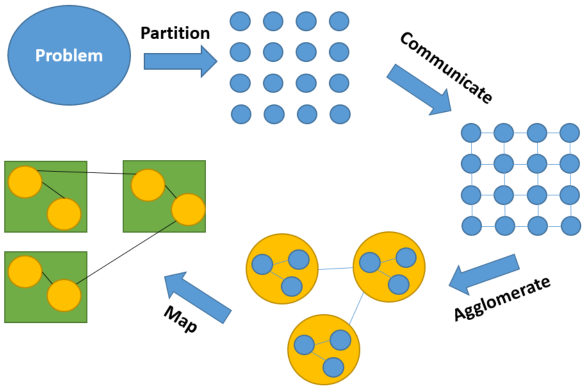
- One-dimensional row-wise decomposition
- Two-dimensional decomposition of the various matrices

In Foster's parallel design methodology, there are basically four steps (Figure 1).

- *Partitioning* – data operated on computation are decomposed into smaller tasks, also known as ***Primitive Tasks***.
- *Communication* – communication to coordinating these primitive tasks are determined, and appropriate communication pattern and algorithms are defined.
- *Agglomeration* – evaluate the performance requirements and implementations costs are accessed on first two stages, and if necessary, tasks are collected into larger task group to improve performance or reduce the development costs.
- *Mapping* – individual task is assigned to a processor in such that it attempts to full-fill the satisfactory goals of maximizing processor utilization and minimizing communication costs. Mapping is achieved by statically or dynamically at run-time by load-balancing algorithms.

The smimilar techniques has also further discussed details in Quinn[2] book with many real live problems including Floyd algorithm.

Figure 1: Foster: a design methodology for parallel programs



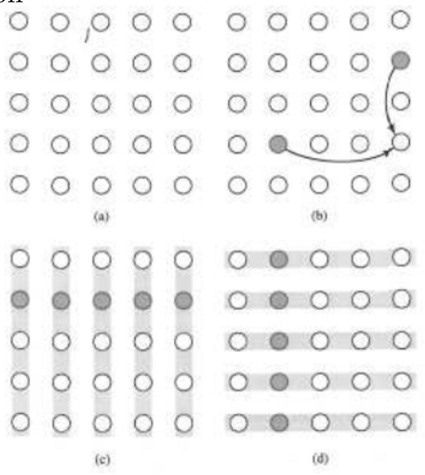
4 Challenges in MPI

Recall the *Floyd-Warshall algorithm*, it involves **3-loops**, thus having complexity of $0(N^3)$. During k'th iteration, elements in matrix '**a**' can be updated concurrently. It is because the k'th column and the k'th row have not changed during the k'th iteration.

```
for (k=0; k<n; k++)
  for (i=0; i<n; i++)
    for (j=0; j<n, j++)
      a[i][j]=MIN(a[i][j], a[i][k]+a[k][j]);
```

By using Foster's design approach, I can first partition each $a[i, j]$ as a primitive task. As for communication, updating $a[i,j]$ during k'th iteration requires values of $a[i,k]$ and $a[k,j]$, but in which they might be in other process memory after partition.

Figure 2: Floyd - communication during k'th iteration



Therefore, it requires to broadcast $a[k,j]$ to all $a[0,j], a[1,j], ..., a[n-1,j]$ as well as broadcast $a[i,k]$ to all of $a[i,0], a[i,1], ..., a[i,n-1]$ elements. For agglomeration and mapping, one MPI process can be responsible for a chunk of the '**a**' matrix. Memory storage of '**a**' can be evenly divided by number of vertices in respect to processors. I can choose row-wise data division or a block of data division. In this project, I approach using row-wise decomposition and I choose to assign each MPI process with a number of consecutive rows of matrix '**a**'.

4.1 Task Parallelism

One key challenge in this approach is the communication requirement of $a[k,j]$ during k'th iteration. Since entries of '**a**' are divided into row-wise blocks, the $a[k,j]$ is also in the local memory of the MPI process that owns $a[i,j]$. However, $a[k,j]$ is in another MPI process's memory.

```
bcast_row = (int *) malloc
 (nodesCount * sizeof(int));
int k;
for (k = 0; k < nodesCount; k++) {
 root = k / (nodesCount / nprocs);
  if (rank == root) {
   offset = k % (nodesCount / nprocs);
    int h;
    for (h = 0; h < nodesCount; h++) {
     bcast_row[h] =
     local_mat[offset * nodesCount + h];
     }
  }
MPI_Bcast(bcast_row, nodesCount,
  MPI_INT, root, MPI_COMM_WORLD);
```

This observes the key difference to parallellizing in Project 1C Mandelbrot Set MPI implementation. In Mandelbrot Set, it requires only data parallelism and once data is partitioned, each primitive task does not require to communicate each other any more.

4.2 Memory Consideration

Compare to OpenMP, the MPI programming requires more careful consideration of the memory data structure and how to it is used by each MPI processes. In sequential program, the distance matrix is back by *stack* memory. However, for sending and receiving between MPI processes, it requires contiguous memory structure such that it can be used in communication.

```
int* matrix = (int *) malloc
   (nodesCount*nodesCount * sizeof(int)
```

Therefore, a *heap* memory is used for underlying storage. Furthermore, I deconstruct 2D into 1D array and send each chuck of row vertices to each MPI processes so that it does not require to broadcast the whole adjacency matrix to every MPI processes. Then each MPI process maintain their own local matrix.

```
int* local_matrix = (int *) malloc
  (mychunk * sizeof(int))
```

Then, I use `MPI_Scatter` to distribute each chunk of task to processes.

```
MPI_Scatter(matrix, mychunk, MPI_INT,
 local_matrix, mychunk, MPI_INT,
 0, MPI_COMM_WORLD)
```

And, after the main shortest path computation task has performed by each process, I use `MPI_Gather` to collect back each process results.

```
MPI_Gather(local_mat, mychunk, MPI_INT,
 final_matrix, mychunk, MPI_INT,
 0, MPI_COMM_WORLD)
```

5 Conclusions

Run time breakdown table in Figure 3 and Figure 4 summarise the run time on different number of processor on number of node for different problem sizes, e.g. V(4096) is a graph with 4096 vertices. There is one limitation in my MPI implementation which is number of problem size has to be divisible by number of processors due to the row-wise division approach. This can be further improved using some heuristic approaches such as using of Block-wise decomposition or two-dimensional decomposition of the various matrices.

In summary the use of multiple cluster nodes have to consider for the communication cost when programming parallel implementation. As the amount of compute nodes grows, the communication overhead is observed in Figure 3 elapse time analysis. Compare to Mandelbrot Set computation, the All Pair Shortest Path graph search problem have to consider not only data partition but also task parallelism.

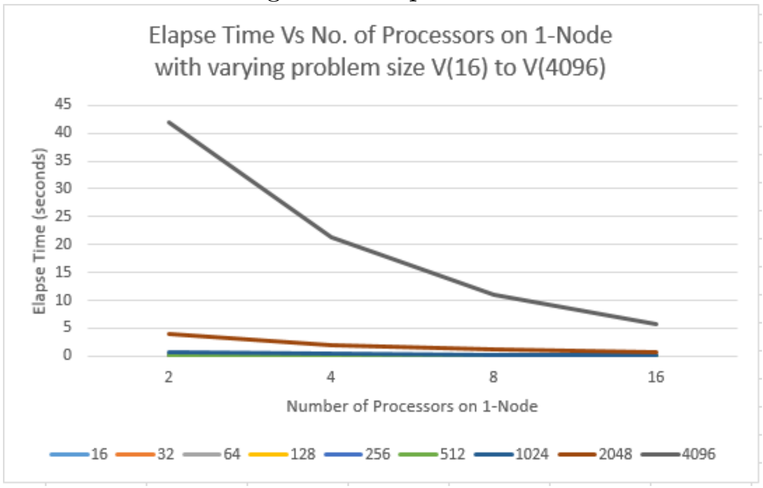
6 References

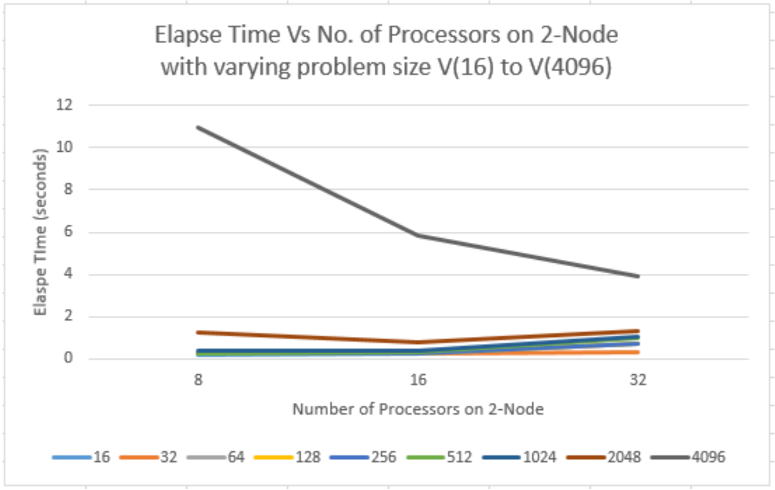
[1] Ian Foster. *Designing and Building Parallel Programs.* 1995. URL: `http://www.mcs.anl.gov/~itf/dbpp/text/node35.html` (visited on 10/14/2016).

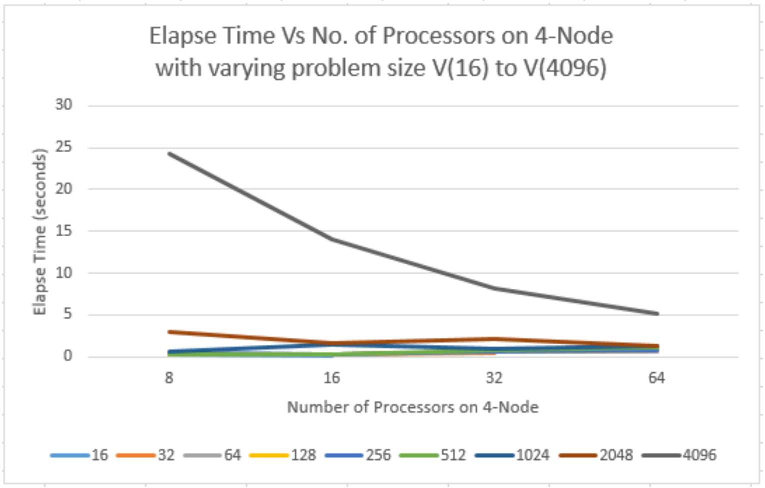
[2] Quinn Michael J. *Parallel Programming in C with MPI and OpenMP.* International series of monographs on physics. McGraw Hill, 2003. ISBN: 007-282256-2.

7 Appendix

Figure 3: Elapse Time







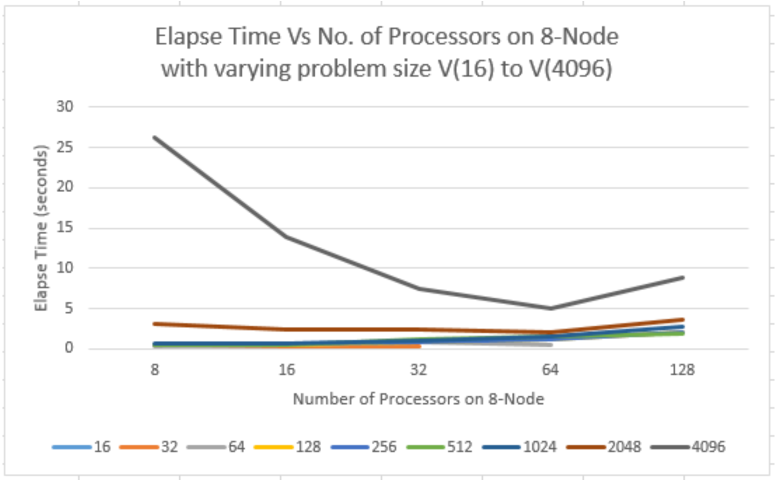
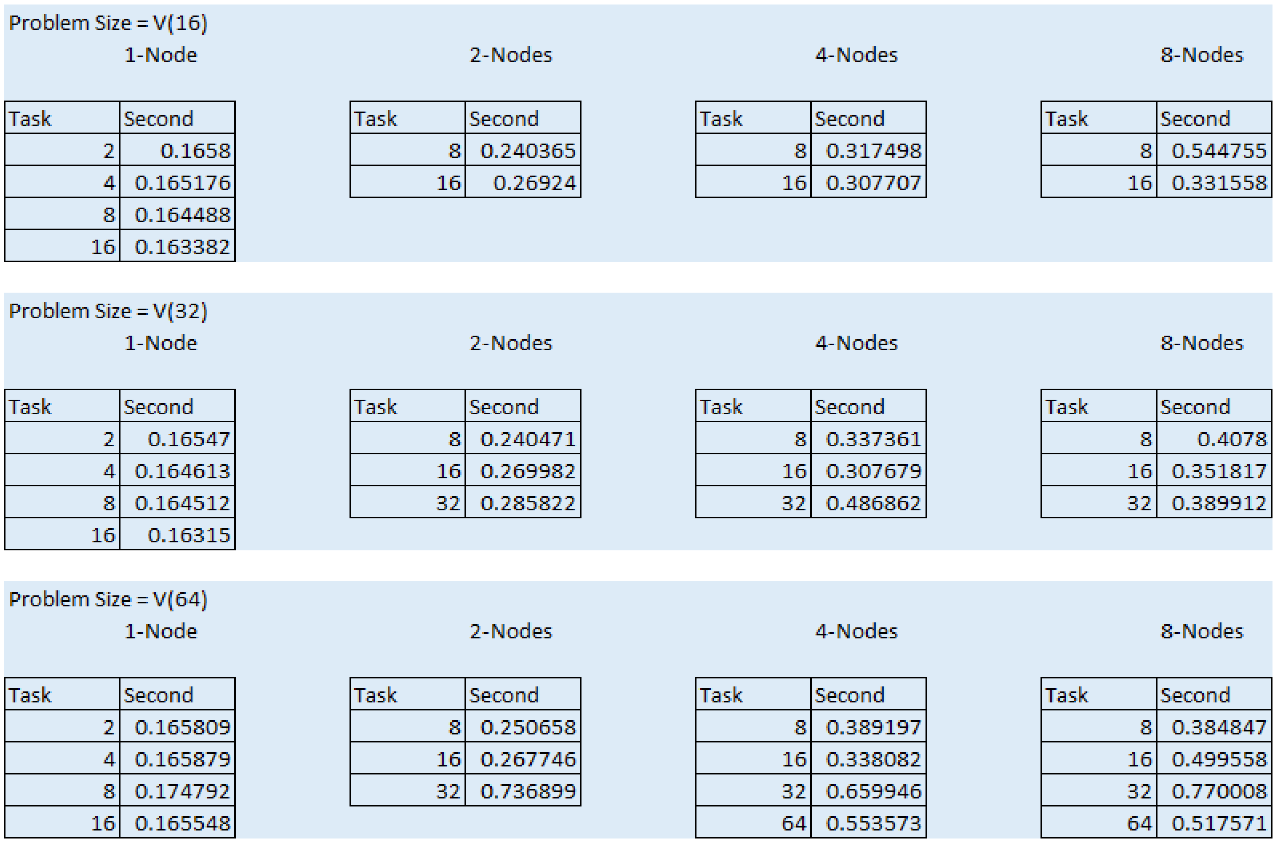


Figure 4: Run Table



Problem Size = V(16)

1-Node

Task	Second
2	0.1658
4	0.165176
8	0.164488
16	0.163382

2-Nodes

Task	Second
8	0.240365
16	0.26924

4-Nodes

Task	Second
8	0.317498
16	0.307707

8-Nodes

Task	Second
8	0.544755
16	0.331558

Problem Size = V(32)

1-Node

Task	Second
2	0.16547
4	0.164613
8	0.164512
16	0.16315

2-Nodes

Task	Second
8	0.240471
16	0.269982
32	0.285822

4-Nodes

Task	Second
8	0.337361
16	0.307679
32	0.486862

8-Nodes

Task	Second
8	0.4078
16	0.351817
32	0.389912

Problem Size = V(64)

1-Node

Task	Second
2	0.165809
4	0.165879
8	0.174792
16	0.165548

2-Nodes

Task	Second
8	0.250658
16	0.267746
32	0.736899

4-Nodes

Task	Second
8	0.389197
16	0.338082
32	0.659946
64	0.553573

8-Nodes

Task	Second
8	0.384847
16	0.499558
32	0.770008
64	0.517571

Problem Size = V(128)

1-Node

Task	Second
2	0.167632
4	0.166724
8	0.166358
16	0.165866

2-Nodes

Task	Second
8	0.24073
16	0.268549
32	0.737151

4-Nodes

Task	Second
8	0.337626
16	0.338047
32	0.711636
64	0.835303

8-Nodes

Task	Second
8	0.474591
16	0.510095
32	0.771399
64	1.110322
128	1.981473

Problem Size = V(256)

1-Node

Task	Second
2	0.175
4	0.171491
8	0.169811
16	0.168017

2-Nodes

Task	Second
8	0.251587
16	0.282596
32	0.737676

4-Nodes

Task	Second
8	0.418542
16	0.349163
32	0.695405
64	0.870887

8-Nodes

Task	Second
8	0.406569
16	0.60847
32	0.761192
64	1.108386
128	2.039784

Problem Size = V(512)

1-Node

Task	Second
2	0.230862
4	0.203306
8	0.205352
16	0.180499

2-Nodes

Task	Second
8	0.259407
16	0.297346
32	1.010841

4-Nodes

Task	Second
8	0.376087
16	0.374232
32	0.84108
64	1.056008

8-Nodes

Task	Second
8	0.475802
16	0.571581
32	1.117283
64	1.475391
128	1.863416

Problem Size = V(1024)

1-Node

Task	Second
2	0.646522
4	0.419225
8	0.304788
16	0.245886

2-Nodes

Task	Second
8	0.378103
16	0.361261
32	1.04515

4-Nodes

Task	Second
8	0.621656
16	1.464363
32	0.92203
64	1.288938

8-Nodes

Task	Second
8	0.655523
16	0.649683
32	1.041977
64	1.574144
128	2.78052

Problem Size = V(2048)

1-Node

Task	Second
2	3.935024
4	2.081632
8	1.156527
16	0.708292

2-Nodes

Task	Second
8	1.235428
16	0.802614
32	1.290229

4-Nodes

Task	Second
8	2.905379
16	1.676032
32	2.121082
64	1.371864

8-Nodes

Task	Second
8	3.139268
16	2.315067
32	2.365576
64	2.063466
128	3.614111

Problem Size = V(4096)

1-Node

Task	Second
2	41.8382
4	21.22626
8	10.91949
16	5.742229

2-Nodes

Task	Second
8	10.94741
16	5.838925
32	3.926428

4-Nodes

Task	Second
8	24.22412
16	14.1228
32	8.14385
64	5.153336

8-Nodes

Task	Second
8	26.18674
16	13.88357
32	7.504528
64	5.076737
128	8.749066