Programming Assignment 1- Parallel Dijkstra

Ke Chang
Yesheng Ma

Oct 20th, 2016

Abstract

The known Dijkstra algorithm is used to find the single-source shortest path problem. And what we have done in this project is to parallelize it in order to complete the task with higher efficiency.

1 Introduction

First let's shed a light on how serial Dijkstra algorithm work. The main body is n-1 iterations. In each iteration, the algorithm selects the vertex which has the smallest distance from the origin vertex among those unknown vertices. Then it will be put in the known list and all its adjacent vertices will be updated for next iteration.
The general idea behind the partition is not to completely separate the whole algorithm, which will be too complex, but to decompose one single step- to find the min distance among the remaining vertex. During this process, we can divide the whole vertex set into n parts. And each progress finds the one with smallest distance respectively. And progress 0 does the reduce work- to find the desired one among those chosen one.

2 Implementation

2.1 Build and Test Automation

During building the whole project, we use the tool GUN Makefile, which is a software for auto-build of C programs. Also we write a shell script to test the program, which takes two arguments, i.e. data size and thread number. Example usage can be `sh test.sh 1000 10`.

2.2 Implementation of Parallel Dijkstra Algorithm

2.2.1 MPI API

We use several MPI APIs to implement this algorithm, type signatures are left over for simplicity:

- `MPI_Init` mark the entry of MPI code
- `MPI_Comm_size` get the size of MPI communication space
- `MPI_Comm_rank` get the rank of current thread
- `MPI_Allreduce` similar to the reduce function in functional programming languages
- `MPI_Gather` gather arrays in different threads to form a large array

2.2.2 Core implementation

The code for parallel Dijkstra is mainly shown in the `Dijkstra` function shown in source code. Before execution of this function, we have already partitioned the input data into separate matrices `loc_mat`. All the executions of `Dijkstra` should be make on these matrices.

Next, we will explain how this `Dijkstra` function actually works. In fact, the parallel Dijkstra is a quite naive algorithm. The major part parallelized is computation of the node which is currently unmarked and has the minimal distance to source. To operate on one submatrix, we first need to declare an array `loc_known` denoting the currently marked vertices. In next n-1 iterations, we find the vertices with locally minimal distance and reduce this to get the vertex with global minimal distance. Finally, we use this new minimal-distance vertex to update all unmarked vertices. Code for this algorithm is shown below:

```
1  void Dijkstra(int loc_mat[], int loc_dist[], int loc_pred[], int loc_n,
       int my_rank, int n)
2  {
3      int loc_v, *loc_known;
4      loc_known = malloc(loc_n * sizeof(int));
5      for (loc_v = 0; loc_v < loc_n; loc_v++) {
6          loc_dist[loc_v] = loc_mat[0*loc_n + loc_v];
7          loc_known[loc_v] = 0;
8          loc_pred[loc_v] = 0;
9      }
10     if (my_rank == 0)
11     loc_known[0] = 1;
12     for (int j = 1; j < n; j++) {
13         int my_min[2], glbl_min[2];
14         Find_min_loc_dist(loc_dist, loc_known, loc_n, my_min, my_rank);
15         MPI_Allreduce(my_min, glbl_min, 1, MPI_2INT, MPI_MINLOC,
               MPI_COMM_WORLD);
16         if (my_rank == glbl_min[1]/loc_n) {
17             loc_known[glbl_min[1]%loc_n] = 1;
18         }
19         int new_loc_dist;
20         for (loc_v = 0; loc_v < loc_n; loc_v++) {
21             if (!loc_known[loc_v]) {
```

```
            new_loc_dist = glbl_min[0] + loc_mat[glbl_min[1]*loc_n +
                loc_v];
            if (new_loc_dist < loc_dist[loc_v]) {
                loc_dist[loc_v] = new_loc_dist;
                loc_pred[loc_v] = glbl_min[1];
            }
        }
    }
    free(loc_known);
}
```

2.2.3 Potential Risks

During programming this algorithm, I found that if the graph is complex and edge weight is large, we may suffer from integer overflow. However, this may not be a big issue for data set in this course. To handle this issue, we may introduce a high-precision integer class, which can lead to some overhead.

2.3 Run It on Cluster

After the completion of the whole algortihm, we started to deploy it on the cluster. We utilize *scp* command to transfer files to the cluster. Then log in via *ssh*. After these pre-works, we write the slurm script to submmit the job. The script is shown below:

```
#!/bin/bash
#SBATCH --job-name=mpi_io
#SBATCH --partition=cpu
#SBATCH --mail-type=end
#SBATCH --mail-user=cktonychang@gmail.com
#SBATCH --ntask-per-node=16
source /usr/share/Modules/init/bash
unset MODULEPATH
module use /lustre/usr/modulefiles/pi
module purge
module load gcc openmpic

srun -n $1 --mpi=pmi2 --error=./result/$1_$2.err
    --output=./result/$1_$2.txt ./mpi_io ./data/$2.txt
```

Then we run all input files in the test set on 1, 2, 4, 8 progresses respectively and modify the code to record the serial and parallel time.

3 Result & Analysis

$\#proc$	10	100	300	400	500	800	1000	2000	3000	4000	5000
1	140	130	150	170	170	240	250	600	1290	2210	2950
2	150	170	180	210	210	270	320	730	1290	1870	2880
4	170	160	170	180	190	230	270	710	1280	2030	2910
8	210	170	260	220	250	270	310	710	1170	1990	2870

Table 1: Serial Time

$\#proc$	10	100	300	400	500	800	1000	2000	3000	4000	5000
1	0	0	10	0	0	10	10	50	90	150	260
2	0	0	0	0.	0	10	0	30	40	90	140
4	0	0	0	0	0	10	0	20	40	60	90
8	10	0	0	0	0	0	10	10	20	40	50

Table 2: parallel Time

From these two tables we can see some important facts: Most of the serial time is used on IO instead of the paralle algorithm. Although the number of progresses increased, the total time consumption stays almost the same. The reason besides IO time may be that most work is done by progress 0. So the speed-up is 1 and the efficiency is very low. All of these reflect the imperfection in the algorithm itself.