Parallel Computing Project Report

Project Description

The Single Source Shortest Path (SSSP) problem consists in finding the shortest paths from a vertex to all other vertexes in a graph. This project aims at solving SSSP in parallel. Input data is from "9th DIMACS Implementation Challenge - Shortest Paths". Code and running command is attached at appendix.

Background

Several sequential and parallel algorithms and implementations for SSSP have been proposed, including Dijkstra's algorithm, Bellman-Ford's algorithm, Chaotic Relaxation and Δ-stepping.

In this project, we mainly talk about Dijkstra's algorithm for its high efficiency and simplicity.

The pseudo code of Dijkstra algorithm is showed below:

```
1. set dis[] as infinite, dis[s] = 0, S = V(all vertexs)
2. while s is not empty:
        extract a vertex u with minimum dis from S
        for (u, v) in E:
            if dist[u] + weight(u, v) < dist[v]:
                dist[v] = dist[u] + weight(u, v)
```

In this algorithm, we call the for loop part a relaxation at vertex u. Dijkstra algorithm is similar to greedy search. So its correctness is promised provided that edge weights are non-negative. As out data is the road distance (or travel time) in American city, Dijkstra algorithm is suitable here.

Analysis on Dijkstra Algorithm

As shown above, main cost of Dijkstra algorithm consists of two parts:

- Choose the vertex with minimum distance
- Relaxation

Part II will take $O(m)$ time in any proper implementation. A naive implementation leads to a $O(n^2)$ time cost of Part I. However, using a priority queue, we can reduce this cost to $O(mlogn)$, while a more sophisticated implementation with Fibonacci heap only consumes $O(nlogn + m)$ time.

To accelerate Dijkstra algorithm with multi-core CPU, we can:

- Extract multiple vertexes from the set each time
- Parallelization the Relaxation part

We will discuss the first approach first.

Extract multiple vertexes

If we extract multiple vertexes at one time, the correctness of algorithm is obvious. However, if the stored distance of relaxed vertex is not minimized yet, the relaxation operation will be meaningless as this vertex will be relaxed again in the future.

So the point is the balance between parallelization and effectiveness.

One solution is to extract a vertex only when its distance is already optimal. For example, pop all vertexes with the same distance from the priority queue front is the simplest way. As there are not always vertexes with the same distance, this approach is not that effective. Besides, we can use some heuristic criteria to determine whether a vertex is already with optimal distance.

Following is an example:

define vertexes set as V, priority queue as Q, edges set as E

distance of vertex v is optimal when:

$$d[v] - min_{w \in V,(w,v) \in E} c(w,v) \leq min_{u \in Q} d[u]$$

This is called **IN-Criterion**.

Also, **OUT-Criterion** is:

$$d[v] \leq min_{u \in F, w \in V,(u,w) \in E} d[u] + c(u,w)$$

where F is relaxed vertexes set.

Using these criteria, we can introduce more parallelism in Dijkstra's SSSP algorithm. More details can be seen at [1].

The other solution is to balance the redundant work and parallelization. Δ-stepping algorithm[2] is a typical example.

Parallelize relaxation

Once we have chose a vertex to relax, the relaxation of edges connected to this vertex is independent so that we can do the relaxation in parallel. This part is obvious and the point is how to arrange the distribution of data on different CPU cores.

Implementation

Having discussed these, we implement some serial and parallel algorithms to solve SSSP.

Serial Algorithm

We implement some serial Dijkstra algorithms.

- Dijkstra with priority queue Complexity: $O(mlogn)$
- Dijkstra with Fibonacci heap Complexity: $O(nlogn + m)$

Parallel Algorithm

We implement parallelized version of Dijkstra algorithm using MPI.

The implemented algorithm is described below:

```
1. distribute n vertexes to p processes
    each process maintain the distance array of vertexes distributed to it
2. iterate at most n times:
        MPI_Reduce from all processes to get the closest unrelaxed vertex
        relax the vertex and update dist
3. MPI_Gather from all processes to get the global dist arary
```

We partition the vertexes uniformly to p processes. Each process stores n/p vertexes (except the last one as n may not be divided by p) and edges connected to these edges. So edges connecting vertexes from different processes is duplicated.

Details can be referred at source code.

Experiment

Serial Algorithm

We examine our algorithms with USA-road data. Results are showed in the table.

(Baseline is code from teacher and the contest website.)

Updates	Map-1	Map-2	Map-3
Baseline	281775	1584641	14542507
Dijkstra (priority queue)	298869	1662387	15088677
Dijkstra (Fibonacci heap)	298869(264345)		

Using Fibonacci heap, we can reduce the update number. But the extent is limited.

Time(ms)	Map-1	Map-2	Map-3
Baseline	22.36	160	2432
Dijkstra (priority queue)	40.98	304	3920
Dijkstra (Fibonacci heap)	94.60		

In theoretical analysis, Fibonacci heap is with $O(nlogn + m)$ time while priority queue is with $O(mlogn)$ time, which means Dijkstra algorithm implemented by Fibonacci heap is faster. However, result is just the opposite. Reason is that C++ Standard Library (STL) has implemented priority queue for our use. The Fibonacci heap is implemented by ourselves. So our code is slower than STL within a constant times is reasonable.

And the input graph is sparse graph, which means m is only 3 times of n. The two reasons lead to the fact that Fibonacci version is more slower.

Parallel Algorithm

Time(ms)	Map-1	Map-2	Map-3
Baseline	22.36	160	2432
Parallel Dijkstra -np 1	177.62	1111.66	13484.38
Parallel Dijkstra -np 2	1052.01	6625.22	52954.50

From the result we can see, parallel algorithm is much slower than serial ones. One reason is that input graph is a sparse graph -- average degree of vertex is about 6, which means Parallelization on the Relaxation Part is meaningless. In other word, input graph is not appropriate for parallelization. Another reason is the high communication overhead. Future work is to implement more adequate parallel algorithm.

Reference

[1] Kainer M, Träff J L. More Parallelism in Dijkstra's Single-Source Shortest Path Algorithm[J]. arXiv preprint arXiv:1903.12085, 2019.

[2] Meyer U, Sanders P. Δ-stepping: a parallelizable shortest path algorithm[J]. Journal of Algorithms, 2003, 49(1): 114-152.

[3] Maleki S, Nguyen D, Lenharth A, et al. DSMR: A parallel algorithm for single-source shortest path problem[C]//Proceedings of the 2016 International Conference on Supercomputing. ACM, 2016: 32.

Appendix

File Tree Stucture

bin/* executable binary files

bin/parallel/* parallel binary files

bin/serial/* serial binary files

bin/results/* execute results

bin/*.pl perl scripts

code/parallel source code

code/serial source code

report/* report

```
.
├── bin
│   ├── parallel
│   │   ├── mpi.out
│   │   └── mpiC.out
│   ├── results
│   │   ├── parallel.chk
│   │   ├── parallel.res
│   │   ├── serial.chk
│   │   └── serial.res
│   ├── run-parallel.pl
│   ├── run-parallelC.pl
│   ├── run-serial.pl
│   ├── run-serialC.pl
│   └── serial
│       ├── dij.out
│       └── dijC.out
├── code
│   ├── parallel
│   │   ├── Makefile
│   │   ├── longlong.h
│   │   ├── mpi_Dijkstra.c
│   │   ├── mpi_solve.backup.cpp
│   │   ├── mpi_solver.cpp
│   │   ├── nodearc.h
│   │   ├── parser_gr.cc
│   │   ├── parser_ss.cc
│   │   ├── timer.cc
│   │   └── values.h
│   └── serial
│       ├── Makefile
│       ├── dij.cpp
│       ├── dij.h
```

```
|       ├── dij_solver.cpp
|       ├── fib.cpp
|       ├── fib.h
|       ├── longlong.h
|       ├── nodearc.h
|       ├── parser_gr.cc
|       ├── parser_ss.cc
|       ├── timer.cc
|       └── values.h
└── report
    ├── report.md
    └── report.pdf
```

Execute Command

For binary executable files (dij.out, dijC.out, mpi.out, mpiC.out), the command is:

```
./solver graphname(.gr) auxname(.ss) resfile(.res/.chk)
```

For perl scripts, the command is:

```
perl perl_script.pl
```

dij.out and mpi.out will print running time.

dijC.out and mpiC.out will generate check files.

Notice: For files in directory bin/results, all are generated with only the first three graphs except serial.res is with all 6 graphs.