Lab1 Report

Dayou Du(dd2645@nyu.edu)

1. General Description and Compilation Guide

    a. Code Structure
    There are three parallel versions (used to illustrate the effect of deeper-loop-unrolling) in src/ folder – src/version{0,1,2} , where version2 has the best performance. In each folder MultithreadSolution.cpp contains the multithread-tsm code, ptsm.cpp contains the entrance code, and Common.h contains the global definitions and switches.

    For the difference between the three versions, please refer to the READMEs in each folder and the further descriptions below (section 2.c).

    b. Compilation Guide
    Simply, module load gcc-6.2.0 and then type make should be fine on crunchy machines.

2. Algorithm Description

    a. General Description

    The solutions are based on the same DFS-and-pruning idea (which also given on the lab instruction): Depth-First-Search through all the possible routes, recording the shortest path that it could get so far, pruning when the visited distance already exceed this value.

    b. Parallel Techniques

    Deeper Loop unrolling:
    If we simply parallelize the first loop, then there will lead to two problems.

    I. The workload for each thread could be unbalanced. E.g. If we run 10 cities (9 loop counts) on 8 threads and we only parallelize the first loop, then 7 threads will be assigned paths that rooted at one city, and 1 thread will be assigned paths that rooted at two cities. Meanwhile, the overall performance depends on the slowest thread.
    We can notice this effect clearly by comparing the result between version1 and version2 on 15 Cities Result (section 3.d), with number of threads 7-13: version1 (without a deeper loop unrolling) runs in approximately same time while version2(with a deeper loop unrolling) is gaining performance with numThreads increasing.

II. The parallelism will be limited by the number of cities. Since the loop count for the first loop is number of cities - 1, thus we cannot exploit more scalability than it.
We can notice this effect clearly by comparing the result between version1 and version2 on 15 Cities Result (section 3.d), with number of threads more than 15: version1 (without a deeper loop unrolling) runs in approximately same time while version2(with a deeper loop unrolling) is still gaining performance with numThreads increasing.

To target at these two problems, we can unroll the first two loops. i.e. Instead of distributing the first visited cities among threads, we distribute the combination of the first two visited cities among all the threads. By doing this we have a larger loop with loop count = (numCities – 1) * (numCities – 2). Then the load balancing could be much better, also we can exploit more scalability than the number of cities. We can see this gain by comparing the result between version1 and version2.

Minimize critical section:
To minimize the length of the critical section (updating global minimal path when a thread finds one), I set only one shared variable (literally, an int) for version1 and version2, which stores the length of the shortest path found so far. The path sequence to reach this "shortest path" is still stored privately for each thread. After the searching finish, the program gathers up the sequences and print out the correct one.

c. Version Difference
   i. version0
   This version simply divides the work as even as possible, use the same algorithm as serial version, and finally gather up the result.
   Since different threads do NOT share the shortest path they got, thus the "user time" would be larger than serial version.

   Shared Var | NONE
   Loop Unrolling | YES

   ii. version1
   This version shared minDis-the shortest path they got, but do NOT unroll the first two loops.
   Since different threads DO share the shortest path they got, thus the "user time" would be close to serial version. But accessing the shared variable critical section brings overheads.

   Shared Var | minDis_s
   Loop Unrolling | NO

iii. version2

This version shared minDis-the shortest path they got. Also unroll the first two loops, which distribute the workloads more evenly.
Since different threads DO share the shortest path they got, thus the "user time" would be close to serial version. But accessing the shared variable without race condition brings overheads.

Shared Var | minDis_s
Loop Unrolling | YES

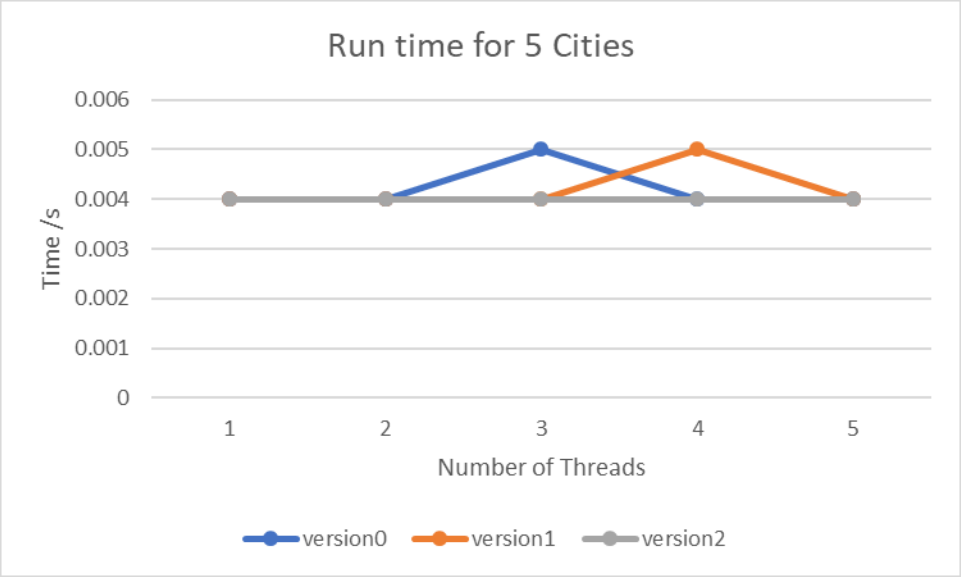
3. Experimental Result

Note: Although the Lab Instruction merely asked for the result on 5-cities and 10-cities, here we also provide the result on 15-cities, and the majority of analytic is based on 15-cities result.
The reason is that on 5-cities and 10-cities, the serial version is already FAST-ENOUGH – about 0.004s according to time command. The system overhead (program loading, thread creation) and the input/output overhead is 0.002s-0.003s, then we cannot see any useful insights through 5-cities and 10-cities results.

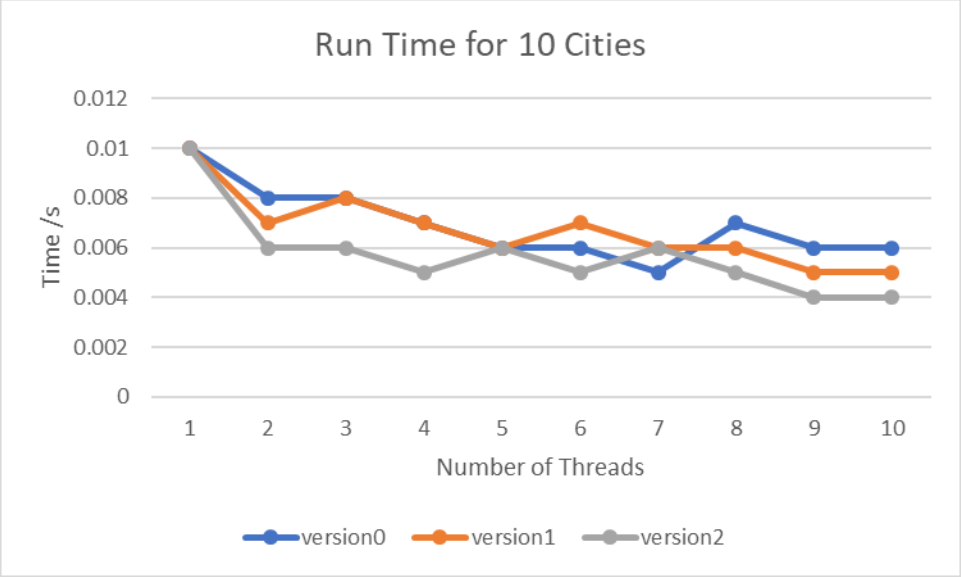
a. Environment

All the tests are done on crunchy3, with g++6.2.0, optimization level -O2.

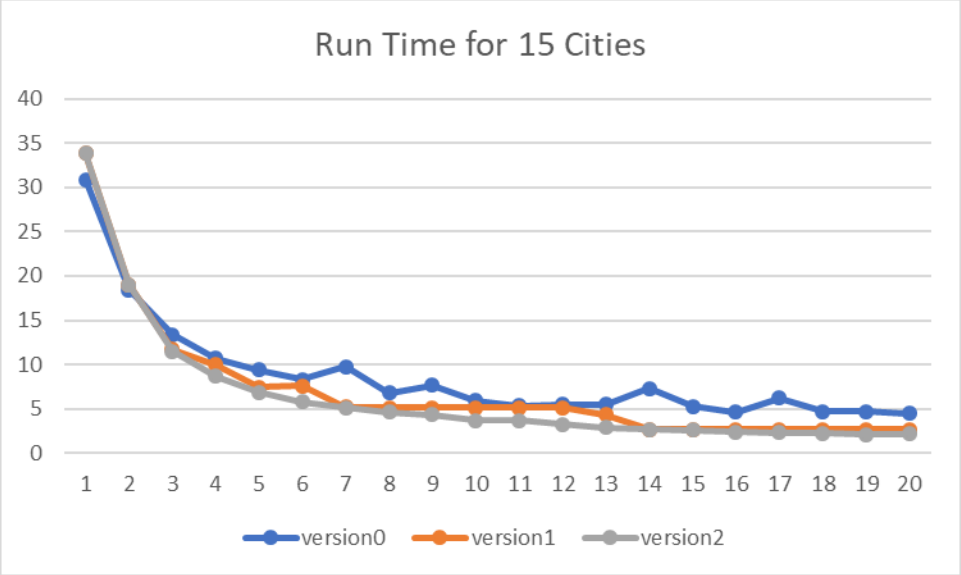
b. Run Time for 5 Cities



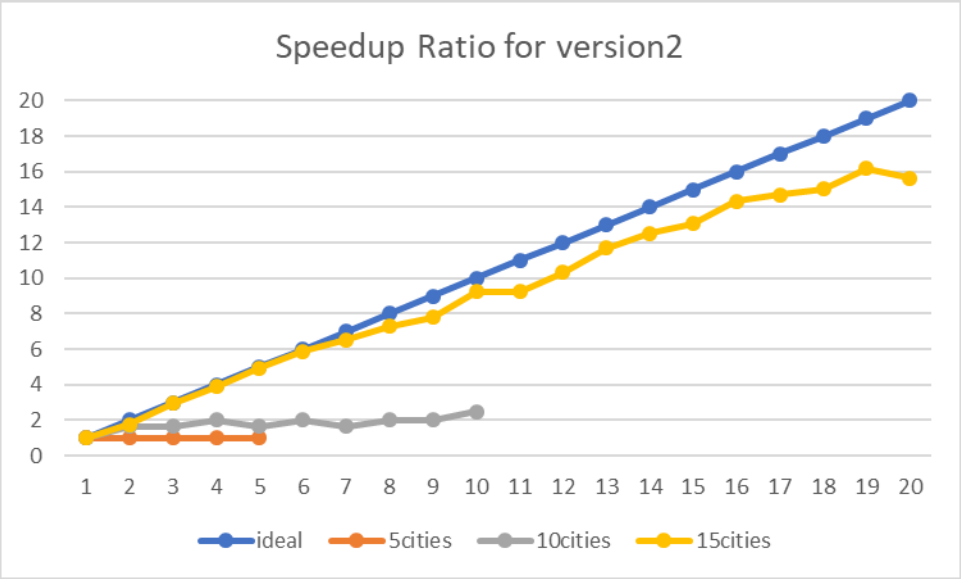
c. Run Time for 10 Cities



d. Run Time for 15 Cities



e. Speedup Ratio for version2 on {5,10,15} Cites



4. Result Analytic
   a. Why cannot see performance gain on 5-Cities (3.b) and 10-Cities (3.c) ?

      By checking /proc/cpuinfo, we know that crunchy3 has 32 cores, thus the performance is not limited by the number of processor. By checking the output from time command, we can see that the sys-call overhead (program loading, thread creation, file reads) is 0.002s-0.003s, and the total run time for serial version is about 0.004s for 5-Cities, and 0.01s for 10-Cities, thus the original program is already fast enough so that we can not gain much by using multi-threads.

      For the test on 15-Cities we can see that all the three versions scales well, and version2 can achieve a maximum of 16.2x speed-up with 19 threads.

   b. Effect of deeper loop unrolling – workload balancing

      If we merely parallelize the first loop. We would face a problem of unbalanced-workload (described in section 2.b) – the performance of the whole program is restricted by the thread with the heaviest workload. In our case, the first loop only has a loop count of numCities – 1, thus most of time it is hard to have balanced workload, without a deeper loop unrolling.

      We can notice this effect clearly by comparing the result between version1 and version2 on 15 Cities Result (section 3.d), with number of threads 7-13: version1 (without a deeper loop unrolling) runs in approximately same time – about 5.15s, while version2(with a deeper loop unrolling) is gaining performance with number of threads increasing – from 5.15s to 2.17s.

c. Effect of deeper loop unrolling – more scalability

If we merely parallelize the first loop. We would face another problem that we cannot gain more performance if we have more threads than numCites. (also described in section 2.b).

We can notice this effect clearly by comparing the result between version1 and version2 on 15 Cities Result (section 3.d), with number of threads more than 15: version1 (without a deeper loop unrolling) runs in approximately same time – about 2.69s, while version2(with a deeper loop unrolling) is still gaining performance with number of threads increasing – from 2.59s to 2.16s. Meanwhile we can achieve a maximum of 16.2x speed-up with 19 threads for version2, according to 3.e.

d. Why cannot scale linearly with more threads?

As we can see from test 3.e, version2 scales almost linearly with less than 16 threads, and tend to meet a limit after that. There are several factors that limit our scalability:

i. The critical section to access shared variable:
To synchronously access and update the shared variable which stores the shortest path globally, I use a critical section to protect it. If we have too much threads they might contend on it.

ii. Some branches are not pruned, comparing to serial version
A pruning decision is made on the condition that the visited path length exceeds the shortest path found so far. However, some branches that was pruned in serial program is not being pruned with multi-threading, because the branch is being visited before the global shortest path has been updated. We can notice this effect by checking the “user time” by time command – The multi-threading version has a slightly larger total “user time” comparing to the single thread version.

iii. The workload is hard to be balanced with too much threads
Although, through a deeper loop unrolling, we try to make the workload as balance as possible, there’s still a chance that some threads have heavier workloads if there is less pruning on certain branches. And this chance would increase with more threads.

iv. The overhead of thread creation and sync
Always exists, and increase with more threads.