CMSC 23010 Homework 1 Programming Final

Ruolin Zheng

April 10, 2019

The svn submission in the `hw1` directory includes:
`shortest_paths.c` - the Floyd-Warshall Algorithm program, Serial an Parallel version
`Makefile` - for `shortest_paths.c`
`tests.py` - the automated testing script for correctness and performance
`plot.ipynb` - the script for plotting performance statistics
`performance.txt` - performance statistics from running slurm

1 Design and Design Changes

1.1 Preconditions

The test input should not contain negative cycles. For the same reason, all test inputs generated by `tests.py` have non-negative edge weights.

1.2 Modules

Please refer to `shortest_paths.c` and the function descriptions for details. **Phase 1** serves as a parser and initializes the `dist` array. **Phase 2** implements the Serial and Parallel versions of the algorithm. **Phase 3** writes to the output file.
Dependencies include `stopwatch.h` for timing and `pthread.h` for threading.

1.3 Interface

For Serial, `./shortest_paths -f FILE`.
For Parallel, `./shortest_paths -f FILE -t NUM_THREADS`.

1.4 Algorithm for Parallel

I followed my original design (copied below,) statically assigning N/T rows (all j's for each fixed i) of the `dist` array to each worker thread. When $N > T$, T threads will be spawn at the beginning of each k loop; when $N \leq T$, I set $T = N$. This approach provides substantial ease of implementation compared to the alternative discussed below in **Improvements**.

Algorithm 1: Parallel

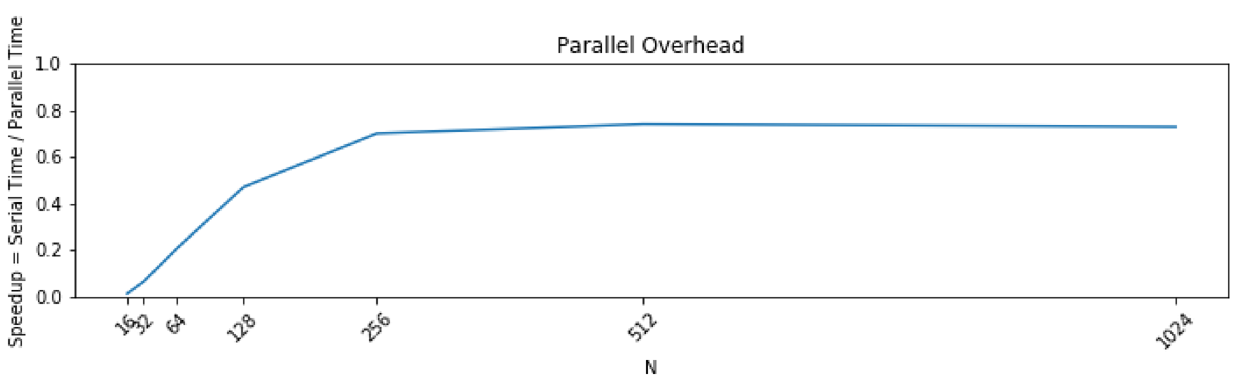
```
for k ← 1 to N do
    // spawn T threads and assign a chunk of size N/T to each
    for t ← 1 to T do
        for i ← (t − 1)N/T to tN/T do
            for j ← 1 to N do
                if dist[i][j] > dist[i][k] + dist[k][j] then
                    dist[i][j] ← dist[i][k] + dist[k][j];
    // join all T threads
```

2 Performance Analysis

For the Performance Testing on both **Overhead** and **Speedup**, I collected data from 10 trials (Serial and Parallel for every $T \in \{1, 2, \ldots, 64\}$) and averaged the results. Please find the raw data in `performance.txt`. Small fluctuations in the trend (plots) may be due to external factors such as OS scheduling and cache usage.

From *https://howto.cs.uchicago.edu/techstaff:slurm*, it appears that the computation node we use has 16 cores and 16 threads per core; this means that the thread numbers in our Performance Testing are all valid.

2.1 Parallel Overhead



Below are a table of the running time as well as a table of the Serial vs. Parallel Ratio.

N	16	32	64	128	256	512	1024
Serial	0.0112	0.0782	0.554	3.9118	28.0513	200.2776	1524.5271
T=1	0.8575	1.1984	2.6515	8.2859	39.965	269.9368	2085.9608

Table 1: Parallel Overhead Running Time (Unit: sec)

N	16	32	64	128	256	512	1024
$\frac{\text{Serial}}{\text{Parallel}}$	0.0131	0.0653	0.2089	0.4721	0.7019	**0.7419**	0.7309

Table 2: Parallel Overhead Ratio

2.2 Parallel Overhead Analysis

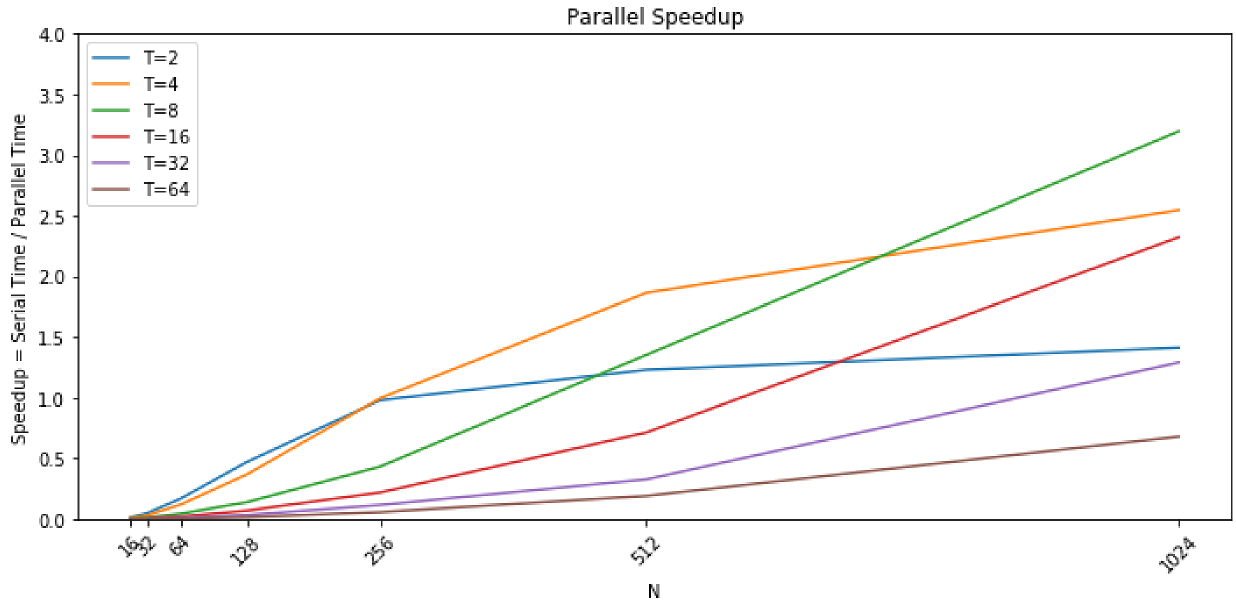
2.2.1 Original Hypotheses

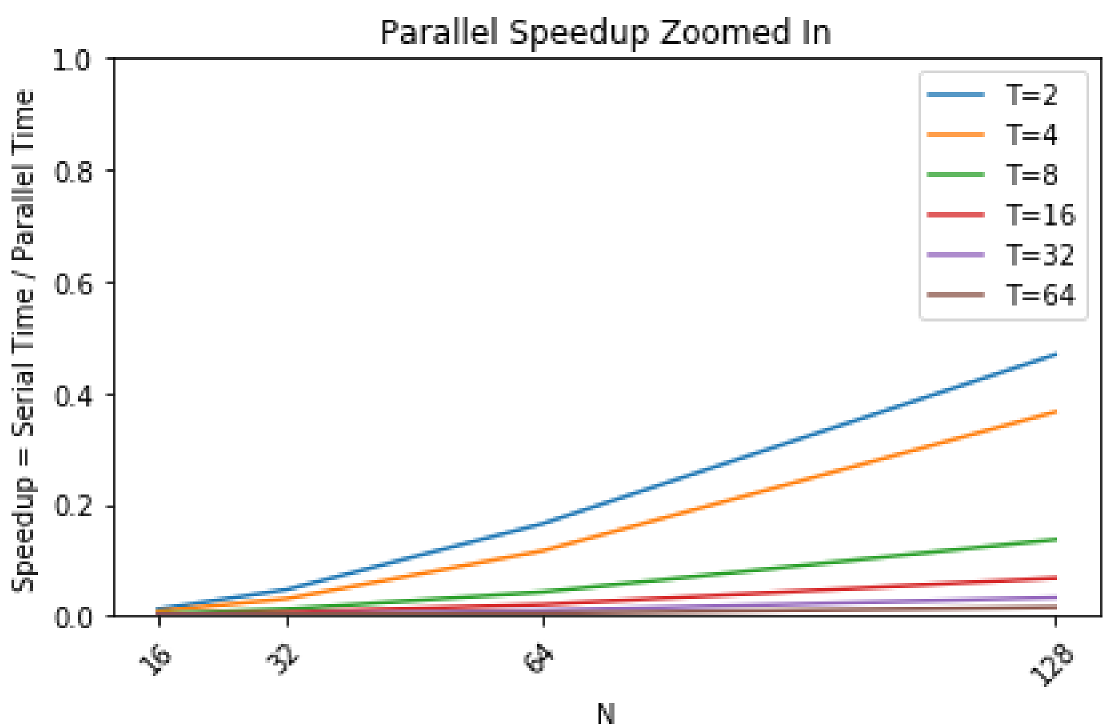
(Copied from the Design Document) With $T = 1$, **Parallel** incurs a constant overhead cost of spawning a single thread for N times, and thus should be slower by an amount proportional to N than **Serial** for all N.

2.2.2 Actual Observation and Reasoning

As shown in the plot, the Serial vs. Parallel Ratio is consistently below 1.0 (reaches 0.74 when $N = 512$,) which means that Parallel incurs overhead by spawning the single thread and is thus slower than Serial. This part is consistent with my original hypothesis.
The plot also shows that although Parallel is significantly slower than Serial for small values of N, it becomes faster (but still slower than Serial) as N increases. This piece of evidence helps correct my original hypothesis: Although Parallel incurs an overhead proportional to N for the N times it spawns a single thread, this overhead becomes less significant as N, the size of the input increases. This is plausible because the increase in time scale of the actual computation work will downplay the overhead cost.
Conclusion: With $T = 1$, **Parallel** incurs an overhead of spawning a single thread for N times. While this renders it slower than **Serial**, the overhead cost appears less significant as N scales.

2.3 Parallel Speedup





Below are a table of the running time as well as a table of the Serial vs. Parallel Ratio.

N	**16**	**32**	**64**	**128**	**256**	**512**	**1024**
Serial	0.0112	0.0782	0.554	3.9118	28.0513	200.2776	1524.5271
T=2	0.9633	1.6792	3.365	8.3531	28.6877	163.0881	1080.2463
T=4	1.4049	2.5879	4.777	10.7052	28.2351	107.4706	598.7194
T=8	3.4275	6.6999	13.0596	28.7295	65.1992	148.2036	477.0281
T=16	6.744	13.3825	27.7758	57.9622	129.9037	281.2457	656.4667
T=32	6.8846	29.3041	60.3712	120.9967	248.0894	617.09	1182.7523
T=64	6.9785	29.7027	130.0029	260.7866	504.1353	1065.8168	2247.5759

Table 3: Parallel Speedup Running Time (Unit: sec)

N	**16**	**32**	**64**	**128**	**256**	**512**	**1024**
T=2	0.0116	0.0466	0.1646	0.4683	0.9778	1.2280	**1.4113**
T=4	0.0080	0.0302	0.1160	0.3654	0.9935	**1.8636**	**2.5463**
T=8	0.0033	0.0117	0.0424	0.1362	0.4302	1.3514	**3.1959**
T=16	0.0017	0.0058	0.0199	0.0675	0.2159	0.7121	**2.3223**
T=32	0.0016	0.0027	0.0092	0.0323	0.1131	0.3246	1.2890
T=64	0.0016	0.0026	0.0043	0.0150	0.0556	0.1879	0.6783

Table 4: Parallel Speedup Ratio

2.4 Parallel Speedup Analysis

2.4.1 Original Hypotheses

(Copied from the Design Document) **Parallel** should be T factors faster than **Serial** for all N. However, Parallel incurs a total overhead of spawning TN threads; thus, for larger N, **Parallel**'s speedup may be less significant.

2.4.2 Actual Observation and Reasoning

(Correction to my hypothesis during Implementation Phase) In actual implementation, when $N > T$, I set $T = N$ because otherwise, some threads would remain idle and Parallel incurs unnecessary overhead in creating them. Therefore, on a N vs. Speedup plot, I'd expect the line for a specific T to lie close to that of $T' = N$ for small $N, N < T$; for larger $N, N > T$, we may observe an upward-sloping "elbow" and a sudden increase in the slope of the line.
We first zoom in on $N \in \{16, 32, 64, 128\}$ in an attempt to observe the "elbow" trend. However, from the **Parallel Speedup Zoom In** plot, for these small N values, none of the T lines reaches 1.0 because of the significant overhead in comparison to a small time scale of computation.
From the **Parallel Speedup** plot, it appears that Parallel does result in a Speedup Ratio

> 1.0 for some T values. As in **Parallel Overhead**, as N increases, Parallel becomes increasingly efficient compared to Serial, since the large amount of computation downplays the effect of the TN overhead of thread creation. In alignment with my concern that Parallel Speedup may be less significant for larger N, the plot shows $T = 2, 4$ as lines that gradually flatten at large N.
I hypothesized that the Speedup should be positively correlated with T; however, the plot does not support my hypothesis and shows the most significant Speedup at $T = 8$. With larger $T = 16, 32, 64$, the Speedup actually decreases. This may be explained by the TN overhead and external factors as mentioned above.
Moreover, theoretically, we may expect an T-fold Speedup for large N. However, in reality, the overhead of thread creation is not negligible. The most significant Speedup Ratio we can achieve is about 3x, at $T = 8$ and $N = 1024$; the most significant Speedup Ration proportional to T is 1.4x at $T = 2$ and $N = 1024$, $\frac{1.4}{2} = 0.7$.
Conclusion: Parallel becomes increasingly efficient compared to **Serial** as N increases. There is an overhead of spawning TN threads. (1) For small T, the effect is that the line gradually flattens at a max possible value; in other words, further increase in input size causes an overhead that offsets the speedup from parallelism. (2) For large T, the overhead is very prominent at smaller N, offsetting the speedup from parallelism and even renders it less efficient than those of smaller T values.

3 Improvements

3.1 Alternative Implementation

An alternative implementation (as discussed in class) would spawn T threads before entering the k loop and use a `pthread_barrier_t` object to ensure that all threads wait until the current k iteration is finished before moving onto $k + 1$. All T threads are joined after exiting the k loop.
Compared to my implementation, this incurs less overhead (T vs. NT in spawning threads.) However, using `pthread_barrier_t` may also incur an overhead (constant or dependent on T.) This alternative approach is also more difficult to implement.

3.2 Cache Usage

Laying out the `dist` array in a continuous 1D chunk might result in better caching usage and improve performance. Accessing by rows (as in my implementation) makes better use of the cache than accessing by columns because of the row-major array design (the cache can load in an entire row.)

4 Instructions on Running the Code

Please make sure `hw1` and `utils` are in the same folder. Please run the following commands within the `hw1` directory:
`make`
`./tests.py -i` generate test inputs in a tests/ folder
`./tests.py -r` run serial and parallel for T = 1, 2, 4, ..., 64 for all test inputs
`./tests.py -o` compare the outputs from serial and parallel
To run individual cases:
`./shortest_paths -f tests/test512.txt` run serial
`./shortest_paths -f tests/test512.txt -t 8` run parallel with 8 threads
This will generate `test512_s_res.txt` and `test512_p_res.txt` in the `tests/` folder, which can then be compared with `diff tests/test512_s_res.txt tests/test512_p_res.txt`.

4.1 Notes on the Testing Interface

`tests.py` is an automated script for both correctness and performance testing. It makes use of the `subprocess` module to run the C program and terminal commands. By default, it run tests on every $N \in \{16, \ldots, 1024\}$ and $T \in \{2, \ldots, 64\}$.
`-h` displays the help menu.
`-i, --generate-inputs` generates test inputs for $N \in \{16, \ldots, 1024\}$ in the `tests/` folder, named as `test[N].txt`.
`-r, --run` runs `./shortest_paths -f tests/test[N].txt` for serial and `./shortest_paths -f tests/test[N].txt -t [T]` for parallel for $T \in \{2, 4, \ldots, 64\}$ for every N. Per the C program specification, this generates `test[N]_s_res.txt` and `test[N]_p_res.txt` in `tests/`.
`-o, --check-outputs` runs `diff tests/test[N]_s_res.txt tests/test[N]_p_res.txt` and checks if the return is an empty string `''` for every N; if so, the correctness test passes.