Multicore Processors - Spring 2019
Programming Assignment 1

Daniel Rivera Ruiz
Department of Computer Science
New York University
drr342@nyu.edu

1 Modified Traveling Salesman Problem

1.1 5 cities

Table 1 shows the average running time (measured over 5 repetitions using the `time` command) for the `ptsm` program with a problem size of 5 cities. Figure 1 presents a bar chart for these results and a linear plot for the Speedup calculated as $S_n = t_1/t_n$ where n is the number of threads.

Table 1

Threads	Time $t(s)$	Speedup S
1	0.0032	1.000
2	0.0036	0.889
3	0.0032	1.000
4	0.0042	0.762
5	0.0040	0.800

1.2 10 cities

Table 2 shows the average running time (measured over 5 repetitions using the `time` command) for the `ptsm` program with a problem size of 10 cities. Figure 2 presents a bar chart for these results and a linear plot for the Speedup calculated as $S_n = t_1/t_n$ where n is the number of threads.

Table 2

Threads	Time $t(s)$	Speedup S
1	0.0642	1.000
2	0.0338	1.899
3	0.0240	2.675
4	0.0190	3.379
5	0.0162	3.963
6	0.0142	4.521
7	0.0154	4.169
8	0.0120	5.350
9	0.0128	5.016
10	0.0108	5.994

Figure 1: Modified Traveling Salesman Problem with 5 cities.

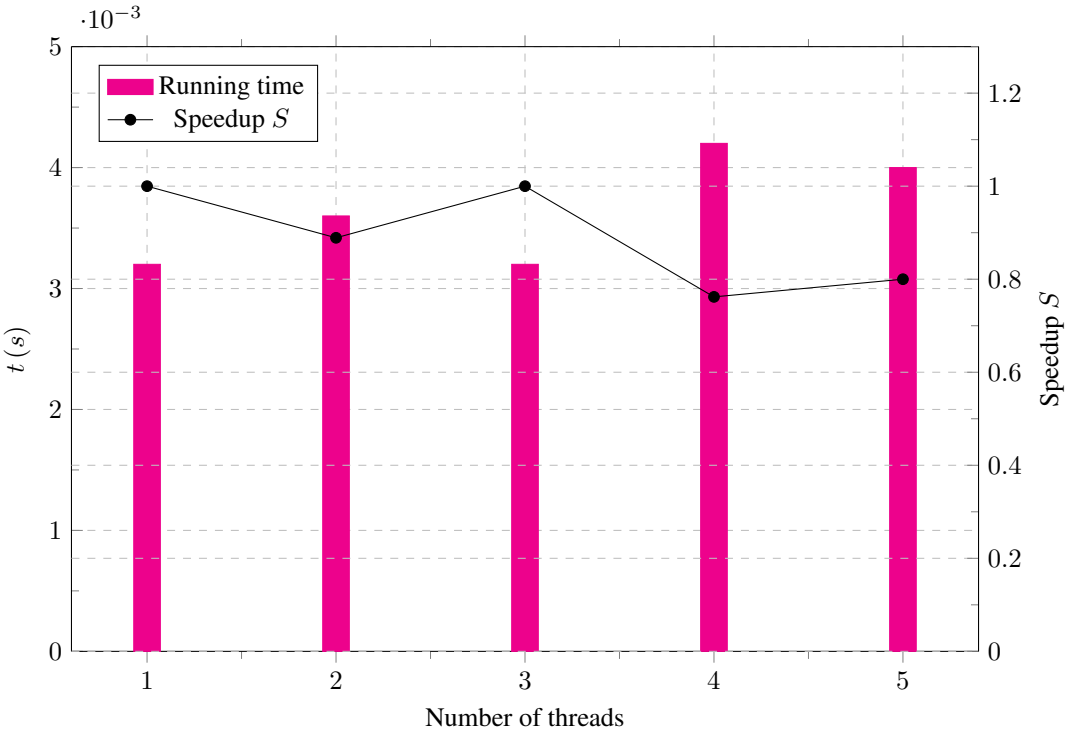
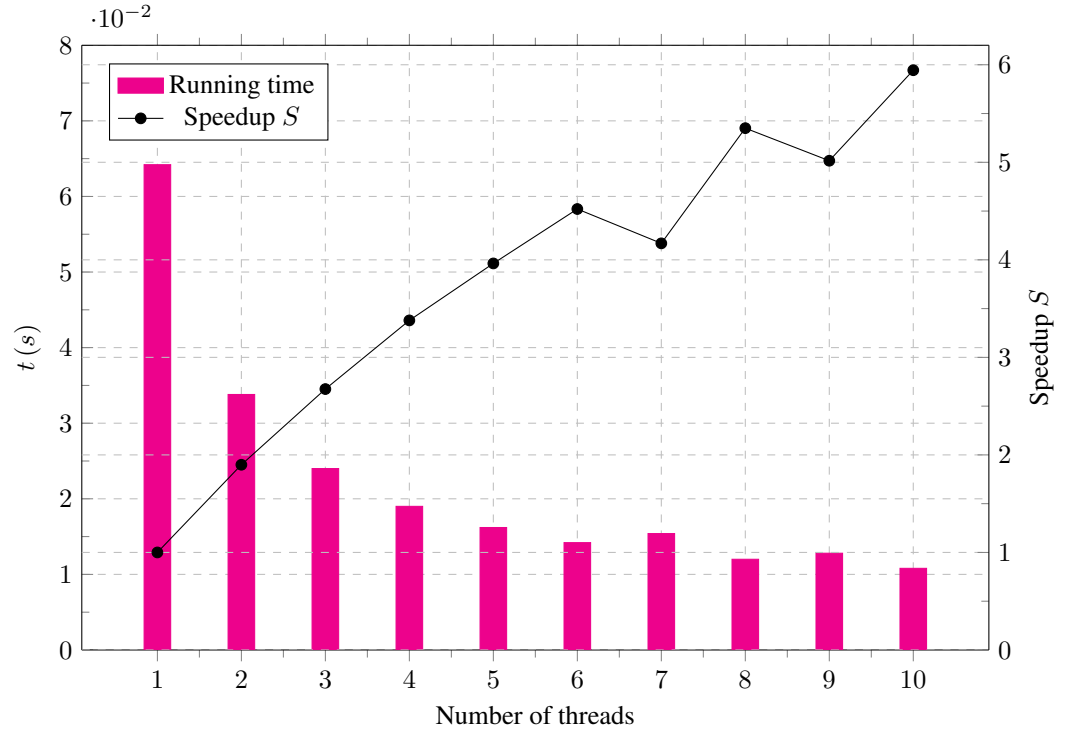


Figure 2: Modified Traveling Salesman Problem with 10 cities.



2 Conclusions

The `crunchy5` server consists of 64 logical cores distributed among 4 *AMD Opteron 6272* processors, each one with 8 physical cores enabled for hyperthreading. Since our application relies solely on OpenMP, the code can only be parallelized within a single processor, i.e. the number of logical cores available for the application is 16.

If we take a look at figure 1, we notice that there is no improvement when the number of threads is increased, and there's even a small slowdown in some cases. The intuition behind this is that the problem size is not big enough to overcome the overhead associated to thread generation. With only 5 cities, the total number of paths to be explored is $4! = 24$ at most, which is not a lot of work even for one thread. Therefore, distributing this among several threads doesn't really yield any improvement.

On the other hand, figure 2 shows a clear improvement in the running time as more threads are introduced. With 10 cities, the number of paths to be explored increases drastically to $9! = 362\,880$. Therefore, generating more threads is actually worth the effort because there is enough work for all of them. In fact, if we were to generate even more threads (up to 16, which is the limit imposed by the hardware) we would get even higher speedups. For this problem in particular, using 16 threads yields an average running time of $0.009\,s$, which is roughly equivalent to a $7\times$ speedup with respect to the one thread implementation. Further increasing the number of threads (to say 32) doesn't really yield any major improvements because the hardware limitations will enforce serialization of some of the threads.

This upper bound on the speedup value is directly related to Amdahl's law. If we take the maximum speedup $S = 7$, the number of cores $p = 16$ and solve for the inherently sequential portion of the code F, we get $F \approx 8\%$. This means that in a sequential implementation (one thread), the portion of the code that cannot be parallelized, i.e. input file reading and weight matrix initialization, accounts for about 8% of the overall execution time.

Even though there isn't enough data to confirm this, there seems to be a trend where implementations with an odd number of threads have slightly less performance than expected. In general, choosing a number of threads that is a power of 2 is a good rule of thumb, since it will make the best use of resources (e.g. reduce operations based in binary trees) plus it will most likely be a multiple of the number of cores in the system.