CS 4380 / CS 5351
Parallel Programming / Parallel Processing
Fall 2020
Colten Murray

Project 3

Due: *Thursday, October 15 at 3:00 PM*

3.0 Collatz [30u/20g points]

Question 3.1a) What compute times do you get (in seconds)? List them in a table where the first column shows the number of threads used (increasing from top to bottom) and the second column lists the compute time. Use two digits after the decimal point for the compute times.

Upper bound: 50000000

Threads	Compute Time (s)
1	18.73
4	5.04
8	2.83
12	1.91
16	1.50
20	1.33
24	1.01
28	1.00
32	0.92
36	0.79
40	0.72
44	0.84
48	0.66
52	0.71
56	0.70
60	0.66
64	0.72

Question 3.1b) What is the highest observed speedup (relative to the pthread code running with one thread)?

Both 48 threads and 60 threads had a speedup of 28.38.

Question 3.1c) The highest speedup is significantly higher than the number of cores in a compute node of Lonestar 5. What makes such a high speedup possible?

The cores in the Lonestar 5 Compute nodes are able to run multiple threads.

Question 3.1d) Sometimes, the highest speedup is not achieved with a thread count that is an inte-

ger multiple of the core count. Provide a good plausible reason for why that is.

Load imbalance may cause the most parallelized solution to not be the best solution to the work needed.

Submit the **`collatz_pthread.cpp`** source file on TRACS. As always, the questions and answers need to be included in the project report.

3.1 Fractal [30u/20g points]

Question 3.2a) What compute times do you get (in seconds)? List them in a table where the first column shows the number of threads used (increasing from top to bottom), the second column shows the compute time for the smaller input, and the third column shows the compute time of the larger input. Use two digits after the decimal point for the compute times.

Threads	Compute Time (width =512)	Compute Time (width =1024)
1	9.54 s	38.13 s
8	1.36 s	5.33 s
16	0.73 s	2.87 s
24	0.57 s	2.15 s
32	0.43 s	1.58 s
40	0.49 s	1.71 s
48	0.38 s	1.32 s
56	0.47 s	1.53 s
64	0.36 s	1.18 s

Question 3.2b) What is the highest observed speedup for the larger input, the corresponding efficiency (in percent), and at what thread count is the lowest compute time achieved?

The highest observed speedup is 32.37 at thread count 64. The efficiency of this speed up is 50.57%. This thread count had the lowest compute time at 1.18 seconds

Question 3.2c) The cyclic partitioning potentially results in false sharing in the *pic* array. Looking at the code (rather than the compute times), is it likely that significant false sharing will occur? Explain why or why not.

Yes, in the code the pic array is unprotected and it is possible for multiple threads to access the same array index at the same time causing indeterminacy.

Question 3.2d) Why is no barrier needed before starting the timer?

The other threads are not launched until the timed section of the code starts

Name the source file **`fractal_pthread.cpp`** and submit it on TRACS.

3.2 Ray Tracer [40u/30g points]

Question 3.3a) What compute times do you get (in seconds)? List them in a table where the first column shows the number of threads used (increasing from top to bottom), the second column shows the compute time for the smaller input, and the third column shows the compute time of the larger input. Use two digits after the decimal point for the compute times.

Threads	Compute Time (width =2000)	Compute Time (width =4000)
1	1.032323 s	4.08 s
8	0.153434 s	0.60 s
16	0.13 s	0.47 s
24	0.10 s	0.23 s
32	0.10 s	0.32 s
40	0.08 s	0.32 s
48	0.11 s	0.27 s
56	0.08 s	0.30 s
64	0.10 s	0.24 s

Question 3.3b) What is the highest observed speedup for the larger input and why is it so much lower than the corresponding speedup obtained on the fractal code?

The highest observed speedup is 17.74 at thread count 24. The fractal code had a higher speedup because the code is more parallelized than the raytrace code.

Question 3.3c) Explain why running the *prepare* function serially is not a significant problem (in terms of Amdahl's law) for the given inputs.

The prepare function takes only one loop as the raytrace function contains many loops. In this comparison, the serial prepare function is a small percent of the total code ran in the timed section.

Question 3.3d) Which aspect of the *ball_y* computation makes it hard to find a closed-form solution so the values do not have to be precomputed (like in fractal)?

The ball_y computation is hard to parallelize because once the ball_y value passes the limit of –semiwidth, the ball_y value is reset and the value that it is incremented by is flipped.

Name the source file **`raytrace_pthread.cpp`** and submit it on TRACS.