COMP 605 - HW02

Akhil Perimbeti

March 23, 2022

Parallel Programming using OpenMP

NOTE: All code was implemented using parallel for-loops with reductions (No critical sections) - Adding a reduction clause removes flow dependencies speeding up the program.

PART A: Matrix Multiplication (ijk-form) using Parallel Threading

Inner loop (ijk-form): The values of Matrix A are iterated through row-wise $(i, *)$. The values of Matrix B are iterated over column-wise $(*, j)$, and the values of the result Matrix C are iterated as fixed points (i,j). This ijk-form of matrix multiplication has around 1.25 misses per inner loop iteration.

Dimensions: $A = [3000] \times [3000]$, B = [3000]×[3000], **Result matrix C = A*B** =[3000]×[3000].

# of Threads	Execution speed (seconds)
1	226.60627
2	127.73095
4	60.597486
8	27.695219
16	14.095281
32	7.883630
64	5.318834
128	3.502422
256	2.350745
512	2.210789

Table 1: Summary of Results (PART A)

Measuring Speedup

Speedup is the factor by which the time to solution can be improved compared to using only a single processor. In computer architecture, Amdahl's law is a formula which gives the theoretical speedup in latency of the execution of a task at fixed workload that can be expected of a system whose resources are improved. Amdahl's law is often used in parallel computing to predict the theoretical speedup when using multiple processors. However this is too complicated for now, and we will simply apply a method to see the factor of speedup.

As the number of threads/processes increases by a factor of 2, the execution time (for the parallel region) also seems to reduce by approximately a factor of 2. So the speedup factor in this case would be ≈ 2. It is interesting to note that — higher threads more slow because large overhead (EXPAND)

PART B: Monte-Carlo simulation for estimating an irrational number (π)

It is possible to estimate the value of π by generating a large number of random points/darts and record how many fall within the circle enclosed by the unit square. After generating a large number of uniformly distributed random points anywhere from (0,0) to (1,1). Keeping track of the number of points that fall within the circle (N_{inner}) and the total number of points generated(N_{total}), the value of π can be approximated as follows:

$$\pi \approx 4 \cdot \frac{N_{inner}}{N_{total}}$$

Thread-Safe Random Number Generator: **erand48()**

The *drand48()* and *erand48()* functions return non-negative, double-precision, floating-point values, uniformly distributed over the interval [0.0,1.0). The *erand48()* function uses a linear congruential algorithm and 48-bit integer arithmetic to generate the random numbers and is the **thread-safe** version of *drand48()*. (A linear congruential algorithm (LCG) is an algorithm that yields a sequence of pseudo-randomized numbers calculated with a discontinuous piecewise linear equation.) Using the function *rand()* for random number generation is not thread-safe since it uses hidden state that is modified on each call. This might just be the seed value to be used by the next call, or it might be something more elaborate. So, the function *rand()* is not thread-safe (the state must be explicit) ; therefore, it cannot be used get reproducible behaviour in a threaded application.

N_{total}	# of Threads	π Approx.	$\lvert Error \rvert$
10,000	4	3.11400	0.027593
100,000	8	3.128400	0.013193
1,000,000	16	3.140736	0.000857
10,000,000	32	3.141346	0.000247
100,000,000	256	3.141556	0.000036

Table 2: Summary of Results (PART B)

As we can see in the results of Table 2 above, as the number of total points (N_{total})$and thread count increases, the approximated s$
Notice the value for absolute error are dropped significantly as the number of points and number of threads are increased.

PART C: Integral Calculation using Composite Trapezoidal Rule to approximate π

The trapezoidal rule for approximating an integral is:

$$\int_a^b f(x)\,dx \approx \frac{b-a}{2} \cdot (f(a) + f(b))$$

Given that n = number of sub-intervals/trapezoids, and $h = \frac{b-a}{n}$ $\longrightarrow$

$$\int_a^b f(x)\,dx \approx \frac{h}{2} \cdot (f(a) + f(b)) + h \cdot \sum_{i=1}^{n-1} f(a + (i \cdot h)$$

We can now approximate π by approximating the following integral using the composite trapezoid rule:

$$\int_0^1 \frac{4.0}{1+x^2}\,dx = \pi$$

So for my results for Problem 3 are displayed in two different tables. The first table (Table 3) shows the convergence of the Composite Trapezoid Rule, as we can see that when the number of trapezoids increases, the absolute error between the approximated solution and the exact solution grows smaller and smaller. The second table (Table 4) shows the speedup when using mutliple threads. In order to properly see the effect that increasing the number of threads had on the execution time of the parallel region, the total number of trapezoids was set to 100,000 and only the total number of threads were changed. As we can see in the results from this table below, the speedup was quasi-linear, as the execution time decreased approximately by a factor of two, when the number of threads were increased by a factor of 2. (Please refer to tables 3 and 4 on the next page).

N (# of trapezoids)	# of Threads	π Approx.	$\lvert Error \rvert$
10	1	3.1665	0.0249073
20	2	3.128400	0.013193
40	4	3.1428358	0.0012431
80	8	3.1426872	0.0010945
160	16	3.14173	0.000698
320	32	3.141582	0.000392

Table 3: Summary of Results (PART C) - Convergence

N (# of trapezoids)	# of Threads	Execution speed (seconds)
100000	1	12.01456
100000	2	6.45021
100000	4	3.11258
100000	8	1.32667
100000	16	0.58259
100000	32	0.22389

Table 4: Summary of Results (PART C) - Speedup