MPCS 51100 - Fall 2019
PSet 3
Threading

Due Date: Nov 18, 2019 @ 5:30pm on Canvas

Submission Requirements (10 pts)

- There are no output formatting requirements, though several of the problems do require specific functional interfaces to help make grading a little easier.
- Submit all files in a single zip file, with filenames (e.g., p1.c, p2.c) as prescribed by the problem set so that your code can be compiled and run by the grader easily.
- In your top directory, populate the README.txt file to list your name, the assignment, and a discussion of any shortcomings your code may have. You will receive more partial credit if you are able to clearly identify faults or problems with your code, rather than letting us find them ourselves.
- Include a Makefile. Your code should compile with gcc using the following flags: `gcc -Wall -O2 -fopenmp -lm`.
- When using pthreads, you may use the `omp_get_wtime()` function for timing purposes.
- The academic integrity policy for this assignment is as described in the course syllabus.
- We do not require any particular coding standard and we do not have strict style requirements. However, your code should be human readable, which means it should be very clean, well commented, and easy for the grader to understand.

1 Serial Matrix Product - p1.c (5 points)

Write a function in p1.c that computes the product ($C = AB$) of two $N \times N$ integer matrices. Write a main program in p1.c that randomly initializes A and B and executes the function, using timers to measure performance. There are multiple valid ways of computing matrix products that you are free to choose from, though to simplify parallelization in later problems you are encouraged to use as simple a method as possible. Your function should have the following prototype:

```
// Computes the 2D matrix product C = AB, with matrix dimensions N x N
void matrix_product_serial( int N, int ** C, int ** A, int ** B );
```

2 Matrix Product via pthreads - p2.c (5 points)

Copy your serial matrix product function from Problem 1 to p2.c. Then, create a new parallel version of your matrix product function using pthreads. Write a main program in p2.c that randomly initializes A and B and executes both your serial and parallel versions, using timers to measure the performance of each. Include a test that checks whether the parallel result matches the serial result given the same input matrices and prints out whether the test was a success or a failure. Note that there is a non-trivial cost for creating and joining threads, so be sure to design your parallelization scheme so as to minimize the number of times threads are created/joined. Your parallel matrix product function should have the following prototype:

```
// Computes the 2D matrix product C = AB, with matrix dimensions N x N
// using nthreads number of pthreads
void matrix_product_pthreads( int N, int ** C, int ** A, int ** B, int nthreads );
```

3 Matrix Product via OpenMP - p3.c (5 points)

Repeat the previous problem, but this time using OpenMP instead of pthreads. Use the following prototype:

```
// Computes the 2D matrix product C = AB, with matrix dimensions N x N
// using nthreads number of OpenMP threads
void matrix_product_openmp( int N, int ** C, int ** A, int ** B, int nthreads );
```

4 Matrix Product Comparison - p4.pdf (5 points)

Compare the performance of your pthreads and OpenMP matrix product methods for a matrix of dimension $N = 2,000$ using various numbers of threads. Create a strong scaling plot showing speedup on the y-axis and number of threads on the x-axis for your two methods. Speedup is computed as:

$$\text{Speedup} = \frac{\text{Runtime with 1 thread}}{\text{Runtime with n threads}} \tag{1}$$

Submit your plot in p4.pdf. Be sure to label your axes and series, and give the graph a title.

5 Serial Julia Set - p5.c (5 points)

In the next several problems, you will be creating codes that plot Julia sets $f(z)$, which are defined by the following equation for complex values z and c:

$$f(z) = z^2 + c \tag{2}$$

More information on Julia sets is available at `https://en.wikipedia.org/wiki/Julia_set`. For the purposes of this assignment the important thing to know is that we can use an iterative algorithm to determine if any given point z in complex space is within a Julia set defined by some fixed value of c. Using this

iterative test, we can create a plot of a Julia set by testing many points in a domain and plotting the results. Algorithm 1 below gives the full algorithm you'll need to plot a Julia set for a particular choice of c, on the real domain of $-1.5 < x < 1.5$ and imaginary domain of $-1.0 < y < 1.0$. For reference, a plot of this particular Julia set is given in Figure 1 below so you know what to aim for.

Algorithm 1 Normal Julia Set Calculation For $c = -0.7 + 0.26i$

```
1: Input N
2: Input empty N × N grid of integers P                          ▷ Holds results
3: c_x = −0.7
4: c_y = 0.26
5: Δ_x = 3.0 / N
6: Δ_y = 2.0 / N
7: for 0 < i < N do
8:     for 0 < j < N do
9:         z_x = −1.5 + Δ_x i                                    ▷ Real component of z
10:        z_y = −1.0 + Δ_y j                                    ▷ Imaginary component of z
11:        iteration = 0
12:        MAX = 1000
13:        while (z_x^2 + z_y^2 < 4.0) AND (iteration ≤ MAX) do
14:            tmp = z_x^2 − z_y^2
15:            z_y = 2 * z_x * z_y + c_y
16:            z_x = tmp + c_x
17:            iteration = iteration + 1
18:        end while
19:        P[i][j] = iteration
20:    end for
21: end for
```

In p5.c, write a function that computes a Julia set in serial, using the following functional interface:

```
// Computes an N x N julia set, storing the results in the 2D pixels matrix
// (pixel matrix allocated by caller, not within function)
void julia_set_serial( int N, int ** pixels );
```

Include a test main in p5.c that calls your function on a $N = 1000$ problem. Include timers in your application to test how long the work region of the application takes to complete.

Your code should be capable of writing the finished Julia set pixel grid to file, by way of the following function:

```
// Saves an integer matrix of size N x N to file
void save_integer_matrix( int N, int ** pixels, char * fname );
```

Finally, write a plotting script in the language/program of your choice (e.g., python, matlab, julia, etc), and generate a plot of your set. Include the script in your submission as `julia_plotter.<extension>`, and add comments on the script any instructions that might be necessary to run the script (arguments, expected input filenames, etc) so that the graders can run your julia set code and generate a plot from your script so as to check that everything is working as expected.

Include your final plot in a file called `p5.pdf`.

6 Julia Set via pthreads - p6.c (10 points)

Copy your serial code from Problem 5 to p6.c. Then, create a new parallel version of your matrix product function using pthreads, with the following interface:

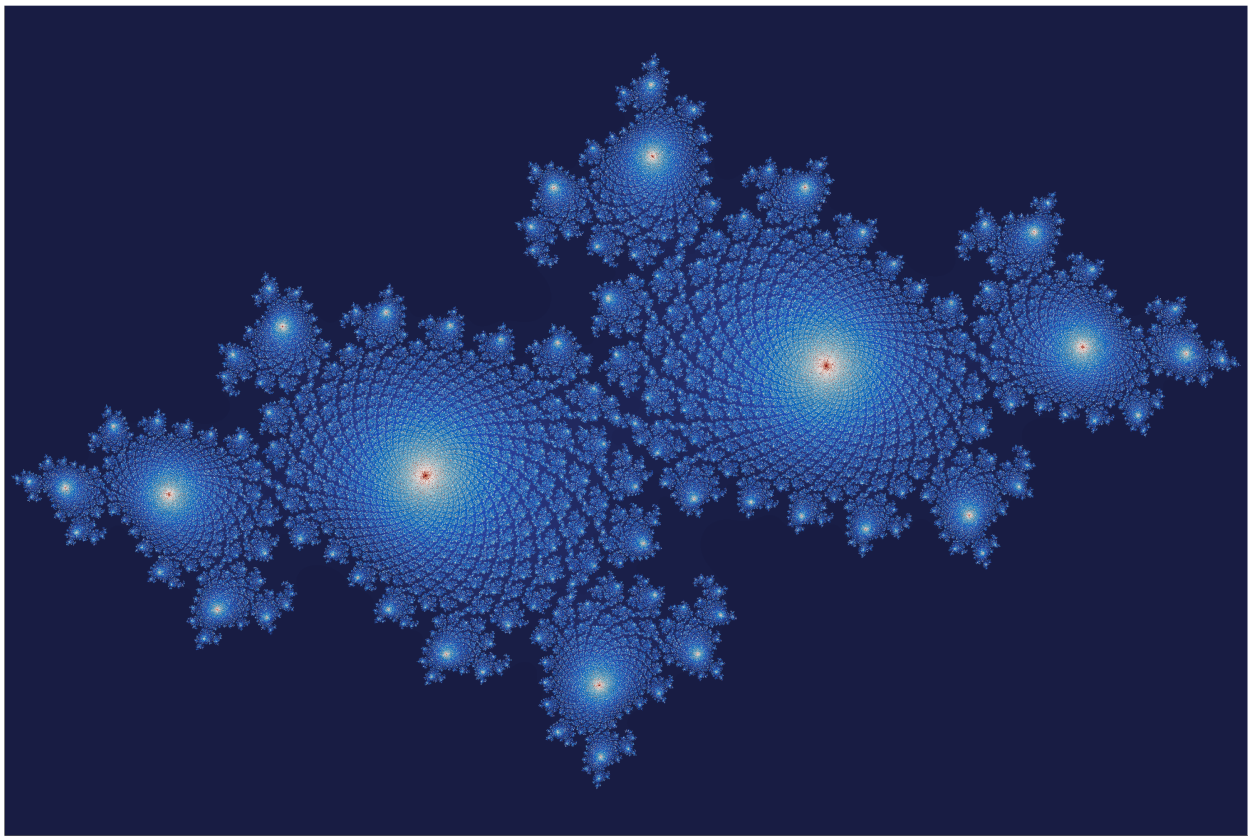


Figure 1: A Julia Set $f(z) = z^2 + c$, where c = -0.7 + 0.26i

```
// Computes an N x N julia set, storing the results in the pixels matrix
// Using nthreads number of pthreads
// (pixel matrix allocated by caller, not within function)
void julia_set_pthreads( int N, int ** pixels, int nthreads );
```

Use a static decomposition of work between threads, such that all threads will handle (approximately) the same number of pixels. Similar to the previous problem, include a test main in p6.c that calls your function on a $N = 1000$ problem. Include timers in your application to test how long the work region of the application takes to complete.

Include your final plot, generated from a run with at least two pthreads, as `p6.pdf`.

7 Julia Set via OpenMP - p7.c (5 points)

Repeat the previous problem in p7.c, but this time using OpenMP instead of pthreads as the parallelization technique. Use the following functional prototype:

```
// Computes an N x N julia set, storing the results in the pixels matrix
// Using nthreads number of OpenMP threads
// (pixel matrix allocated by caller, not within function)
void julia_set_openmp( int N, int ** pixels, int nthreads );
```

Unlike the previous matrix multiply problems in which work could be efficiently distributed among all threads statically, computing the Julia set introduces problems of load balancing. Specifically, each pixel in the domain will take a variable amount of work to compute. Some pixels will only require a single iteration,

while others will take thousands of iterations to complete. This means that a static decomposition (as you did with pthraeds in the previous problem) may result in some threads taking much longer than others to finish, resulting in inefficiencies. To address these load balancing issues, you should use the OpenMP dynamic load balancer. Experiment with work chunk sizes to determine what is approximately optimal on your system when using all available threads.

Include your final plot, generated from a run with at least two OpenMP threads, as `p7.pdf`.

8 Julia Set Comparison - p8.pdf (10 points)

Test your pthreads and OpenMP Julia set applications against each other for a matrix of dimension $N = 1,000$, using various numbers of threads. Create a strong scaling plot showing runtime on the y-axis and the number of threads on the x-axis for your two methods, and submit your results in p8.pdf. Be sure to label your axes and series, and give the graph a descriptive title.

Also in p8.pdf, discuss if it would be possible to implement a dynamic load balancing scheme in pthreads, and how you might go about doing this if it is possible.

9 Shared Pi - p9.c (15 points)

In previous problems, the end result of all of our computations was a large matrix of data, with each individual element being thought of as an independent task. That is, threads were assigned to different tasks, with each task involving writes to a unique location in memory that others threads could not accesses, so no thread synchronization was required.

In this problem, you will be implementing several slightly different Monte Carlo methods for computing π in parallel using OpenMP. As the end product of our computation this time is a single number, synchronization at some level will be required. You will test a variety of methods for coordinating work between threads.

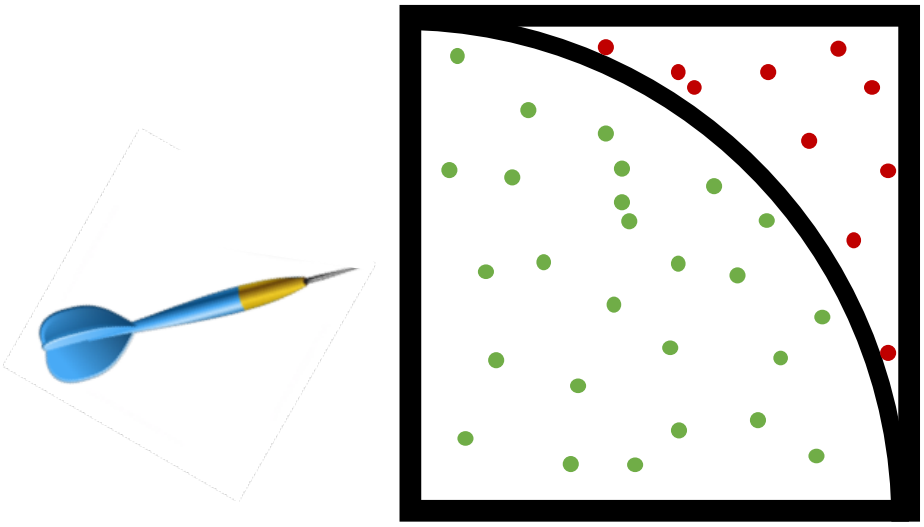
We will use the Monte Carlo "dartboard" method for computing pi. In this method, you will randomly sample points from a uniform distribution within a square domain (i.e., throw darts randomly at a square target). Then, once your darts have landed, you will check whether or not the hit point is located within a quarter circle centered at the bottom left corner of the square, and tally it as a hit if it is indeed within the circle. This random sampling allows us to compute π using the following relationships:

$$\frac{\text{Number of darts that hit within quarter circle}}{\text{Number of darts thrown}} = \frac{\text{Area of quarter circle}}{\text{Area of square}} = \frac{0.25\pi r^2}{r^2} = \frac{\pi}{4} \tag{3}$$

While all darts are independent, we want to come up with a final value for the "Number of darts that hit within the quarter circle" state variable in Equation 3, which we will refer to as the `num_hits` variable. If we were to parallelize the code with all threads adding to `num_hits` in an uncontrolled manner, we might have write conflicts and undefined behavior may occur.

In this problem, implement a serial function that handles the Monte Carlo dartboard pi computation and returns an estimate for pi. Then, once you've tested your function and are confident it is working, make 5 copies of this function. You will parallelize each copy, with each one using a different thread synchronization method from the list below:

1. Atomics – Use OpenMP atomics on the state variable `num_hits` so that the incrementing operation is protected.
2. Critical – Use an OpenMP critical region around the `num_hits` incrementation instruction so that the operation is protected.
3. Manual Reduction A – Instead of having a single `num_hits` state variable, allocate an array of state variables (of length `nthreads`) outside the parallel region. Then, have each OpenMP thread increment the state variable that corresponds to its thread id (as queried by `omp_get_thread_num()`). Finally, once you have exited the parallel region of the code, sum the state variables together to compute your final answer for `num_hits`.



4. Manual Reduction B – Inside of the parallel region, but outside of the OpenMP for loop, declare a new local `num_hits_local` state variable that is private. Inside the OpenMP for loop, have threads increment their local state `num_hits_local` variables instead of the global one. Finally, once the OpenMP for loop has completed, but you are still inside the parallel region, use an OpenMP atomic to protect an add of the local state variable(`num_hits_local`) to the global state variable (`num_hits`) for all threads.

5. Automatic Reduction – Use the OpenMP `reduction()` clause when entering the parallel region to automatically reduce the state variable `num_hits` across threads.

Have your main function time all of the five different methods for 10 million samples each using all available threads on your system. Additionally, have your main function print the value of Pi estimated by each method as well as the error as compared to the true value of Pi.

Note: be sure to take care when sampling numbers in parallel, as the C standard `rand()` function is not thread safe!

10 Shared Pi Analysis - p10.pdf (10 points)

In p10.pdf, compare the runtimes of your 5 different Pi protection strategies for 10 million samples each when running with multiple threads (the max possible for your system). Rank the methods in terms of performance, and discuss the following questions

1. Why are the fastest method(s) fast?

2. Why are the slowest method(s) slow?

3. The "Manual Reduction A" and "Manual Reduction B" methods above appear very similar. If you notice a performance discrepancy between the two, why?

4. The "Atomics" and "Critical" methods may appear similar in that they are both instructing OpenMP to protect access to the `num_hits` variable. If you notice a performance discrepancy between the two, why?

Additionally in p10.pdf, describe briefly how you would choose to efficiently implement the Monte Carlo dartboard Pi method if using pthreads instead of OpenMP. Also discuss how you would control access to the `num_hits` state variable via pthreads so as to maximize efficiency.

11 Parallel Prefix Algorithm - p11.c (15 points)

So far in this problem set, the problems you have solved have featured a large number of independent tasks which naturally expose a lot of parallelism. In some cases however, it is not as obvious how to parallelize an algorithm. Take for instance the prefix sum algorithm, which operates on an array of numbers, e.g.,:

```
A = [8 4 1 2 6 4 1 2 7 3]
```

and generates a new array of numbers, with each entry serving as the sum of all previous entries (inclusive) in the input array, e.g.,:

```
B = prefix_sum(A) = [8 12 13 15 21 25 26 28 35 38]
```

We can write this simple algorithm in C as:

```
void prefix_sum_serial(int * A, int * B, int n)
{
    int i;
    B[0] = A[0];
    for( i = 1; i < n; i++ )
        B[i] = B[i-1] + A[i];
}
```

At first glance, it would seem impossible to parallelize this algorithm, as each index in the output array is dependent on the previous index, making it an inherently serial process. If we were to simply add an

```
#pragma omp parallel for
```

above the loop in this code, we would not be guaranteed to get the correct output. Even if we were to protect access with atomic operations the loop dependency of the algorithm would not be respected, as we wouldn't be able to guarantee that iteration $i-1$ finishes before iteration i.

However, parallelism is indeed possible, at the expense of a net increase in work. One method to parallelize this algorithm is to split it into several stages, as follows:

1. The problem is decomposed into a set of small subdomains, with each subdomain being assigned to a thread.
2. All threads independently execute the prefix algorithm on their subdomain of the problem, ignoring any numbers preceding their subdomain. They store the total sum for their subdomain in a location other threads can access.
3. All threads wait until the previous step has been completed by all other threads.
4. All threads compute the sum of all subdomains preceding them by way of adding together the total subdomain sums computed in step (2).
5. All threads add the sum computed in step (4) to all elements in their subdomain, and store their results in the output vector.

Implement this parallel prefix sum algorithm in p11.c, using either pthreads or openmp. Write a test main that tests the serial algorithm against your parallel implementation on a randomized array of 100 million elements and compares their runtime performance. After collecting runtime data, your test main should also compare the serial and parallel solutions so as to prove that your parallel function is working correctly, and print whether the test is successful or not.

Generate a plot of speedup vs. number of threads for your parallel code. In problem 4 (matrix multiply), we expected our speedup plot to increase (somewhat) linearly as we added threads to the problem. For example, we expected a speedup of approximately 2.0 when running with 2 threads. Do you also expect to see this behavior for your parallel prefix code? Why or why not? Submit your plot and discussions in p11.pdf.