CSE 6010
Final Assignment - Option 1
Parameter Optimization with OpenMP
11/30: Corrected parentheses and clarified update on p.2

Final Submission Due Date: 11:59pm on Monday, December 9
Submit Makefile and code as described herein to Gradescope through Canvas
NO Initial Submission
NO 48-hour grace period

Choose ONE of Final Assignment - Option 1 and Final Assignment - Option 2
If you submit both, we will grade whichever project has a higher autograder score; if they tie, we will grade whichever was submitted later

In this assignment, you will use an optimization technique called particle swarm optimization (PSO) to find parameter values for a model that describes how populations of a predator species and a prey species evolve in time. PSO is a bio-inspired algorithm that iteratively evolves a set of candidate parameterizations toward lower error. You will speed up the calculations by using OpenMP.

Predator-prey model: Background and implementation

The general predator-prey model we will use is a differential-equations model, where x represents the prey and y represents the predator:

$$\frac{dx}{dt} = f_x(x, y) = ax - bxy\,,$$

$$\frac{dy}{dt} = f_y(x, y) = cxy - dy\,.$$

Here a represents the prey growth rate, d represents the predator death rate, and b and c represent the effects of predation on the depletion of the prey and growth of the predator, respectively.

You will solve the equations from initial conditions at time 0 iteratively using the simple "forward Euler" approach: calculate the updated populations as $x_{i+1} = x_i + \Delta t\, f_x(x_i, y_i)$ and $y_{i+1} = y_i + \Delta t\, f_y(x_i, y_i)$, where $x_i = x(i\Delta t)$ (the value of x at time $i * \Delta t$) and $y_i = y(i\Delta t)$ (the value of y at the time $i * \Delta t$. For this assignment, you should set $\Delta t = 0.25$ and set the initial values of x and y at time 0 to $x(0) = x_0 = 2.5$ and $y(0) = y_0 = 3$. Solving the equations iteratively leads to a time series of values for x and y. You should solve the equations for 100 time steps (leading to a total of 101 values, including the initial values).

The goal of this assignment will be to identify values for the four model parameters that can be used to produce time series that are close to a target time series. Here, you will use the model to create the target time series using a set of "true" parameter values, then seek to fit the model to that dataset by finding values for the four parameters that lead to similar (low-error) time series.

Particle swarm optimization: Background and implementation

PSO is an optimization method that does not require the use of derivatives. It maintains a number of candidate solutions, called "particles." We will use PSO to obtain values of the four parameters of the predator-prey model above, so that each particle will include a set of values for a, b, c, and d. PSO updates the particles iteratively; we will do so for a set number of iterations. Evolution of the particles is accomplished using a "velocity" stored with each particle, with a component associated with each parameter; the parameter values collectively are referred to as the "position" of the particle. The set of velocity values for a particle j will be updated based on the best set of parameters the particle has ever found (the set of parameters it has seen that produce the lowest error E_j) and the best set of parameters found across **all** particles.

Particle initialization: The initial position of each particle will be set by selecting initial values for each parameter. For a given parameter, a uniform random value between the upper and lower bounds specified for that parameter in the input file will be generated. In addition, for each particle, each of the four initial velocity components will be set to a uniform random value between 0 and 1. After initializing each particle, it will be necessary to calculate its error and to store that value also as the "local best" error, along with the values of the parameters (the position of the particle), in case the initial position turns out to have the lowest error that particle will see.

Error metric: To quantify the quality of a particle's current position, we will use a simple error metric that consists of the sum of the absolute differences between the true and predicted values for prey and predator at each time point in the time series generated using the true parameter values and the particle's current position: if the time series include $N + 1$ data points arising from initial conditions together with N time iterations, then the error value for particle j is defined as $E_j = \sum_{i=0}^{N}\left(\left|x_i^{true} - x_i^j\right| + \left|y_i^{true} - y_i^j\right|\right)$. (Note that the error will be measured as a property of the particle as a measure of distance from the dataset; there is no reference to the true parameter values used to generate the dataset.)

Particle update: To update the position, the velocity is first updated, using the error associated with the particle calculated following the procedure described above, and then the velocity is used to update the position. Along with the error, it is necessary to have each particle's local best error and the corresponding set of parameter values, together with the global best error achieved by any particle and the corresponding parameter values.

The following process is used to update each particle; the example is shown for parameter a, but the same approach is used for b, c, and d. Here we use the notation p_a to denote a's position and v_a to denote a's velocity. In addition, we denote a uniform random number between 0 and 1 as $z(0,1)$; a different random number should be used every time the calculation is performed.

$$v_a \leftarrow \chi\left(v_a + \phi_{local}\left(z(0,1) * \left(a_{best,local} - p_a\right)\right) + \phi_{global}\left(z(0,1) * \left(a_{best,global} - p_a\right)\right)\right)$$

We will set the values of the three hyperparameters as follows: $\phi_{local} = \phi_{global} = 2.05$ and $\chi = 1/\left(\phi_{local} - 1 + \sqrt{\phi_{local}^2 - \phi_{local}}\right) \approx 0.73$; these values will not be changed during any use of the algorithm. Then, the position is updated using the updated velocity value.

$$p_a \leftarrow p_a + \gamma v_a$$

We will fix the value of γ as well: $\gamma = 0.05$.

Bounds enforcement: As a result of the particle update, it is possible that any of the components of the position (consisting of the updated values for a, b, c, and d) could exit the bounds specified in the input file. For this assignment, if any updated parameter value is outside its bounds, for this assignment you should set it to a value randomly selected within the bounds.

Specifications:

- Input data for the system will be provided through the file provided as the first command-line argument; examples will be provided. This file will include four lines, each with three doubles values, delimited by spaces:
    - Line 1: true value, lower bound, upper bound for parameter a
    - Line 2: true value, lower bound, upper bound for parameter b
    - Line 3: true value, lower bound, upper bound for parameter c
    - Line 4: true value, lower bound, upper bound for parameter d

  An example file will be provided for your use. The assignment will be graded with different input files to test for correctness under a variety of conditions, so you should expect some variations to be tested, including the true parameter values as well as the bounds. In particular, note that by setting the true value and the bounds to the same value, we can essentially limit the number of parameter values the algorithm will find.
- Command-line arguments: Your code should take two required and one command-line arguments:
    - First (required), the name of the file containing the true values and bounds for each parameter.
    - Second (required), the number of threads to use in the parallel region(s).
    - Third (required), a "base" seed value used to generate the random numbers for each thread. (Note that the seed for each thread should be different; use (seed + thread_number).
    - Fourth (optional), the character "p" (without quotation marks), to print additional output.
- Data structures: The data structure(s) you should use for this assignment are not specified, but you should use "good" data structures that will help you implement the PSO algorithm.
- PSO specifications:
    - Set the number of PSO iterations to 50.
    - Set the number of particles to 10,000 (10^4).

- Parallelization:
  - Your code should implement parallelization at one or more appropriate places to achieve a performance improvement with multiple threads.
  - The number of threads should be set to the value given on the command-line.
  - Immediately before and immediately after the parallel region(s) in your code (for example, including the main iterations), call the function `omp_get_wtime()` to obtain the times and accumulate the elapsed time within the parallel region (as in Workshop 5), so that you can later output the total time spent within the parallel regions.
  - In addition to correctness, your code will be tested to ensure speedup is achieved with more threads under suitable conditions.
  - As in Workshop 5, depending on your operating system and other system properties, you may not see any speedup with multiple threads. If this is the case for you, please try the autograder environment, which should demonstrate speedup.
- Generating random numbers with multiple threads:
  - Because you will use multiple threads in this program, you will need to use the thread-safe function `rand_r()` to generate random numbers. Note that `rand_r()` takes as an argument the ***address*** of the seed of type unsigned int.
  - You should also use a different seed for each thread. This can be accomplished by incorporating into each thread's seed the thread ID (which can be obtained using `omp_get_thread_num()`). Here you should use the "base" seed passed as a command-line argument plus the thread_number.
  - In contrast to `rand_r()`, the `rand()` function we have used before uses a global seed value, so if multiple threads call it they might interfere with access to the seed.
- Output to the screen:
  - In all cases: At the end of the program, output the following lines.
    - Line 1: The global best error value, output using %.5f.
    - Line 2: The values of the parameters associated with the global best in the order a, b, c, and d. Output each value using %.5f and separate the values by a space with no commas or other text.
    - Line 3: The cumulative time your code spent in the parallel region (see above). Write this single value followed by a newline character using the %f format specifier.
  - When the command-line argument "p" (without quotation marks) has been provided, you should add the following additional output after the "standard" output described above: Output the global best error value for each iteration using %.5f followed by a newline character. Include the global best error value obtained from the initial values as well as for N additional iterations, for a total of $N + 1$ values output.

To receive full credit, your code must be well structured and documented so that it is easy to understand. Be sure to include comments that explain your code statements and structure. Your code should avoid redundancy and overly complex solutions, and it should achieve speedup for suitably large cases.

Submission (worth 12 points): Submit your code to the Submission on Gradescope. You may submit any number of files named in any way you like, but you will be required to submit a Makefile that will be used to compile your program, and the name of your executable must be **pso** (no extension). Do not zip the files or upload a directory; instead, submit your files directly. **Only submit the source code files that you wrote yourself; do not include the executable or provided files.**

Sample input and output: A sample input file "fit_a.txt" is provided on Canvas. Here is an example of output for specified command-line arguments (your solution may not get exactly the same numbers):

```
./pso fit_a.txt 4 0 p
0.000000
0.800000 0.200000 0.400000 0.600000
0.226206
0.024904
0.024904
0.024904
0.013726
0.013726
0.013726
0.006707
0.003357
0.002294
0.002294
0.002294
0.002294
0.000455
0.000455
0.000455
0.000004
0.000004
0.000004
0.000004
0.000004
0.000004
0.000004
0.000004
0.000004
0.000004
0.000004
0.000004
0.000004
0.000004
0.000004
0.000004
0.000004
0.000004
```

```
0.000004
0.000004
0.000004
0.000004
0.000004
0.000004
0.000004
0.000004
0.000004
0.000000
0.000000
0.000000
0.000000
0.000000
0.000000
0.000000
0.000000
0.000000
```

Some hints:

- Start early!
- Identify a logical sequence for implementing the desired functionality. Then implement one piece at a time and verify each piece works properly before proceeding to implement the next. Working through the tests outlined below in sequence may be helpful.
- Complete a serial version of the code before adding parallelization (but plan ahead!).
- Be sure to test for speedup using the autograder.
- Debugging optimization problems and parallel implementations can be challenging, as errors are not always obvious from observing the output. Coming to office hours to discuss your implementation and walk through the logic of your program will be the best way to resolve this type of issue, so please plan ahead to have enough time to debug the assignment before the deadline.
- For this assignment, there is no need to attempt to parallelize file I/O.

Grading

Program correctness will be assessed using the autograder on Gradescope; valgrind will be used to detect memory errors, including failure to free all allocated memory. Detailed grading specifications are described below. Numbers in parentheses indicate the total points available for a specific component.

1. **Autograder (8)**
    a. **Test 0: Compile the program (0)**: This test is not worth any points, but it will show the output if errors are encountered compiling your program on the autograder. This output should be helpful if your code compiles on your machine but not on the autograder. Fix these errors before moving on to anything else!

   b. **Test 1: Correctness (6):**
      i. (1) Global error is non-increasing in all test cases
      ii. (2 cases, 2 each) Low error is achieved when fitting a parameter individually and the relative error in the parameter value found is also low
      iii. (1) Low error is achieved when fitting all parameters
   c. **Test 2: Speedup (2):** Achieve a speedup factor of 2 or greater when using 4 threads.
2. **Program structure and implementation (2.5)**: Graded manually. Points are earned for implementing the algorithms and data structures correctly as described in the assignment and making reasonable decisions when designing your program.
3. **Program style and documentation (1.5)**: Graded manually. Points are earned for using abstractions to avoid redundant code and making your program readable by using appropriate comments, variable names, whitespace, indentation, etc. Write your code so someone else can easily follow what is going on.

The autograder will run immediately when the program is submitted and should present the results quickly. The manually graded components of the assignment will not be available until after the grades are published. Autograder feedback will only be available with the final submission for the assignment, as the initial submission will not be graded for correctness.

Note that autograder points may be deducted if the instructions of the assignment are not followed, for example, if the program does not set the number of threads to the value of the command-line argument. Additionally, autograder points will be manually deducted if the program output is manually set to pass the autograder without correctly implementing the solution.

Because the PSO algorithm is likely to be effective even with some differences from this assignment's description, a number of manual checks will be included. Examples include but are not limited to the following:

- Use of `rand_r()` with thread-specific seed values (with a different seed for each thread).
- Appropriate data structures that are not too complex and that avoid redundancy.
- Appropriate definition of the error function.
- Proper updates of the local and global best error values and parameter (position) values.
- Specified number of particles used.
- Modularity and minimization of code redundancy.