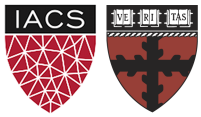
CS205, Fall 2015
Computing Foundations for Computational Science
Ray Jones



HARVARD
School of Engineering
and Applied Sciences

Homework 2
Due November 2, 2015

This homework will be an exploration of multiprocessing and multithreading, and optimization of algorithms in the multithreading regime.

Node: This problem set will require a functional Cython installation. Complete problem 1 soon, post on Piazza or ask for help in Slack if you encounter difficulties.

Get the HW2 skeleton files from GitHub

To bring your repository up-to-date with the CS205 homework repository, we will add it as a remote, then use it as the basis for your work on HW2.

First, switch to your local repository directory on your machine (or VirtualBox). Most likely, you can do this with this command:

```
cd ~/cs205-homework
```

Then, add the CS205 class HW repository as a remote with the name "upstream":

```
git remote add upstream https://github.com/harvard-cs205/cs205-homework.git
```

Fetch the changes we've made to that repository:

```
git fetch upstream
```

Check out a HW2 branch based on the upstream/master branch:

```
git checkout --no-track -b HW2 upstream/master
```

If you have uncommitted changes, you may have to deal with them. If you just want to throw them away, you can run "`git checkout -- .`", though you may just want to commit them all in a wrapup commit.

Make sure you are on the HW2 branch:

```
git branch
```

(You should see a list of branches with an asterisk next to `HW2`.)

Then, begin hacking, and `git add ...files...` and `git commit -m ''...message...''` as you go along.

When you are ready to submit your work, push it to GitHub into your remote repository (you can do this with partial results, if you want a remote backup of your work):

```
git push origin HW2
```

And then follow the instructions from Homework 0 to open a pull request from your HW2 branch to the CS205 homework repository.

Problem 1 - Set up Cython

If you have not already done so[1], clone a copy of the https://github.com/harvard-cs205/cython-install-check repository, and make sure `python run_tests.py` produces the expected output. Check on Piazza for discussions of what may be necessary.

[1] :-(

Problem 2 - Parallel vector updates [10%]

As discussed in class, there are differing approaches protecting shared resources with locks. One approach, "coarse-grained locking" uses a single (or small number) of locks, in order to keep code simple, but sacrificing concurrency. In contrast, "fine-grained locking" uses a large number of locks, normally one per shared item, incurring a larger computational overhead and complexity, but allowing higher concurrency.

In this problem, you will explore fine-, medium-, and coarse-grained locking, in order to understand their costs and benefits. We have provided skeleton files in the `P2` subdirectory for a simple problem (transferring values between two array elements).

This problem has two parts:

- Add code to `P2/parallel_vector.pyx` in the functions `move_data_medium_grained` and `move_data_fine_grained` for parallelism and locking. Your code should parallelize over 4 threads using `prange(..., num_threads=4)`, and use OpenMP locks to ensure correct behavior. Helper functions for creating and destroying locks has been provided. In the medium-grained case, the number of adjacent items managed by one lock should be controlled by an argument to the function.
- Characterize the performance of each approach (coarse, medium, and fine-grained locking) on the two cases in `P2.py`: random data exchanges and correlated data exchanges. In the latter, the two entries exchanging data are near each other. You should choose a range of values for `N` in the medium-grained case, perhaps inspired by the fact that, for the correlated-transfer case, the source and destination are no more than 10 elements away).

Submission Requirements

- `P2/parallel_vector.pyx`: New version with additional code added for 4-thread parallelism and fine- and medium-grained locking.
- `P2/P2.txt`: A text file describing your experiments with the three scales of locking, including your suggestions for a good N to choose for medium-grained locking in the two cases explored (uncorrelated and correlated transfers) and your justification for that choice (or why you were unable to find a good value).
- Any graphs referenced in `P2/P2.txt`, if needed.

Problem 3 - Performance of Instruction- and Thread-Level Parallelism [35%]

In this problem, you will optimize a compute-bound task using thread-level and instruction-level parallelism.

Multithreading

Add multithreading over rows (i.e., `i`) in `P3/mandelbrot.pyx` using a prange loop. Use a chunksize of 1, and static scheduling (`prange(..., schedule='static', chunksize=1)`). Compare performance between 1, 2, and 4 threads.

Instruction-level Parallelism

In the loop over columns (i.e., `j`), use AVX instructions to add 8-way instruction-level parallelism. You can assume that the number of columns will be a multiple of 8.

We have wrapped several AVX instructions for use in Cython in `P3/AVX.pxd`. Read over that file to get a sense of what is available. To use them, you can treat `AVX` as a python module, and call, for example, `a = AVX.mul(b, c)`, where `a, b, & c` are all `AVX.float8` variables.

As discussed in class, these instructions allow you to operate on 8 floating-point values in parallel. You may find it helpful to first separate the complex values into their real and imaginary parts using `numpy.real(in_coords)` and `numpy.imag(in_coords)`.

Compare the performance of 8-way instruction-level parallelism for 1, 2, and 4 threads.

Submission Requirements

- `P3/mandelbrot.pyx`: New version with additional code added for thread and instruction-level parallelism.
- `P3/P3.txt`: Your results for 1, 2, and 4 threads, with and without instruction-level parallelism.
- Any graphs referenced in `P3/P3.txt`, if needed.

Problem 4 - Image Processing [20%]

In this problem, you'll use Python threading to accelerate an image-processing task, and explore the communication primitives provided by the `threading` module for controlling thread behavior and cooperation.

First, download this file and save it as image.npz in the P4 directory.

In the P4/filtering.pyx Cython file, you will find code implementing a 3x3 median filter with edge-clamped boundaries. You should not need to modify this Cython file for this problem.

The code in P4/driver.py uses this code to compute the repeated application of a 3x3 median filter. You will use Python threads from the `threading` module to speed up this process.

Modify the `py_median_3x3()` function to operate with more than one thread (it currently ignores its `num_threads` argument). For N threads, the nth thread should process every Nth line, starting with line n (with 0 indexing). Use some combination of Semaphores, Condition Variables, Events, or other communication primitives from the Python `threading` module to ensure threads do not start an iteration of filtering before the data for that iteration is ready.

Note: to correctly compute iteration i for thread n, threads n, $n-1$, and $n+1$ must have completed iteration $i-1$.

For full credit, do not allocate new buffers for each thread (i.e., use the `tmpA` and `tmpB` arrays as-is).

For extra credit, allow threads to begin an iteration of filtering asynchronously (i.e., thread n iteration i can start as soon as threads n, $n-1$, and $n+1$ have completed iteration $i-1$, without waiting for any other threads to complete iteration $i-1$).

Compare performance for 1, 2, and 4 threads, and discuss this in your writeup. Also describe the method you chose to control cooperation between threads.

Submission Requirements

- `P4/driver.py`: New version with additional code added for N-thread parallelism.
- `P4/P4.txt`: Your writeup comparing 1, 2, and 4-way parallel performance, and discussion of your implementation choices to control cooperation between threads.
- Any graphs referenced in `P4/P4.txt`, if needed.

Problem 5 - Simulation and domain decomposition [35%]

In the `P5` directory, we've provided a simple physics simulator and animation system, which can be run with `"python driver.py"`. The physics engine is in `P5/physics.pyx`. It runs serially, and uses a $O(N^2)$ algorithm for detecting collisions between objects.

This problem has three parts:

Suggestion: As you work on this problems, turn on `boundscheck=True` at the top of `physics.pyx` when adding new code, and change it to `False` when you are sure your code is correct.

Another suggestion: During testing, set the number of objects to 500, with radius 0.01. This allows you to visually verify that the simulation is functioning as expected. Restore the original values (10000, 0.002) before performance testing.

Multithreading

Add code to `physics.pyx` for multithreading using `prange()`, with four threads, and static scheduling. Compare performance with 4 and with 1 thread (i.e., serial performance), and discuss the reasons for any performance difference. Use a large chunksize, for example, 1/4 of the number of objects.

Note: do not worry about simultaneous updates to object velocities (i.e., no need to use locks for this version).

Spatial decomposition

Add code to subdivide the simulation space into a regular grid, with grid spacing of $\texttt{R}/\sqrt{2}$, so that when there is no overlap, only one object can land in a single grid square. We have already provided code to allocate, initialize, and pass the grid to the `update()` function. During the simulation, store the index of objects in this grid, based on their center's location, and keep it up to date as objects move. Note that objects may leave the $[0, 1]^2$ space, and may overlap each other enough to be in the same grid. You do not have to handle collisions with such objects correctly, but these events should not cause a crash, and all objects should continue to update (bouncing off walls and moving according to their velocities).

Use this grid to convert the algorithm from an $O(N^2)$ algorithm to an $O(N)$ algorithm. Note that an object in a grid location could collide with other objects up to two grid squares away.

Compare this version to the non-gridded version, for both 4 and 1 threads, and discuss the reasons for performance differences between all four versions.

Spatially Coherent Sorting

Add code in `driver.py` to sort balls by location each time the animator draws a new frame (so only every $1/30th$ of a second). You can choose whatever sorting strategy you want, but it should result in good spatial coherence (i.e., objects nearby in space should be nearby in memory), and should not take too much computation to compute. Good choices are Hilbert or Morton ordering.

Discuss how sorting affects performance for 1 and for 4 threads, with the grid enabled.

Note: Be sure to update the grid indices and sort velocities for objects, as you sort. (You may find `numpy.argsort()` useful.)

Locking

Finally, add locking for each object, to prevent simultaneous updates to object velocities when they collide. Each object should have its own lock (i.e., fine-grained locking). Be sure to avoid deadlocks.

Compare performance with and without locking, for four threads with gridding and sorting.

Submission Requirements

- **`P5/physics.pyx`: New version with additional code for multithreading, gridding, and locking.**
- **`P5/driver.py`: New version with additional code for keeping objects sorted for coherence.**
- **`P5/P5.txt`: Your writeup comparing performance in each of the subproblems above, and an explanation of your choice for spatial sorting.**
- **Any graphs referenced in `P5/P5.txt`, if needed.**