Homework 4: Cilk Primer

Due: 10:00 P.M. (ET) on Thursday, February 27th, 2025

Last Updated: February 11, 2025

In this homework you will experiment with parallelizing programs with Cilk. You will learn how to use Cilksan to detect and solve determinacy races in your multithreaded code, and how to measure a program's parallelism using the Cilkscale scalability analyzer.

Contents

1 Getting started

Getting the code

You can get this assignment's code using Git:

```
$ git clone git@github.com:CSE491-spring25/homework4_<username>.git homework4
```

Submitting your solutions

Submit your write-up on Gradescope and your code via Git before the deadline stated at the top of this handout. For each question we ask, give a short response of 1–3 sentences. Paste program outputs where necessary.

Reporting performance

To test the performance of parallel code, you should use **`telerun --cores 8`**, a variant of `telerun` that submits jobs to cloud machines with 8 cores. It's important to use `telerun --cores 8` for all performance measurements involving parallel code.

2 Recitation: Parallelism and race detection using Cilk

Please answer the checkoff questions on your own and show your responses to a TA after you have completed *all* checkoffs in this section.

2.1 Introduction to Cilk

The `cilk_spawn` and `cilk_scope` keywords

Compile the `fib` program in the `fib/` subdirectory. Then, time the execution for finding `fib(45)` using the `time` command:

```
$ telerun --cores 8 time -v ./fib 45
```

You will get 3 different times as outputs, labeled `Elapsed (wall clock) time`, `User time`, and `System time`. The `Elapsed time` is wall-clock time, which is the total elapsed time from beginning to end of the execution. The `User time` is the amount of time a CPU spent in user mode. The `System time` is the amount of time a CPU spent in the kernel. The sum of `User time` and `System time` is the actual CPU time of the command. Since the current `fib` is a serial program, you will find that wall-clock time is slightly higher than CPU time.

Next, we want to parallelize the program to take advantage of the other 7 processor cores on the `telerun` machines. You can do this by adding `cilk_spawn` in front of function calls that you want to execute in parallel. You also need to wrap it in a `cilk_scope` to wait for all spawning tasks to finish before returning the result. These keywords can be used when the Cilk header is included:

```
01 #include <cilk/cilk.h>
```

You can specify the number of Cilk workers that execute a program by setting the environment variable `CILK_NWORKERS`. You can, for example, set it to 8 workers when running a program by prepending `CILK_NWORKERS=8` to the command you wish to run. If you do not specify it, the number of workers will default to the number of cores in the system where the program is being run. Setting the number of Cilk workers is especially useful for evaluating execution time as a function of workers. For example, you may run `./fib` with 4 workers on the 8 core machine with the following command:

```
$ telerun --cores 8 time -v CILK_NWORKERS=4 ./fib 45
```

Checkoff Item 1: Parallelize `fib` using `cilk_spawn` and `cilk_scope`, and report the execution time when using 4 and 8 Cilk workers.

We would ask you to run it with 1 worker, but *spoilers* it takes over a minute and times out.

You might find that the new version is not faster than the first one (in terms of `real` time). Furthermore, the CPU time is much higher than wall-clock time, because the program uses multiple processors to run. Under what circumstances would the parallel version be slower? Try to fix your program using "coarsening" to get a parallel speedup.

Checkoff Item 2: Describe your approach to coarsening the program, and report your parallel execution time for the coarsened version of `fib` on 1, 4, and 8 Cilk workers.

The `cilk_for` keyword

The `transpose/` subdirectory contains the source code for `transpose`, an in-place matrix-transpose program. As you saw in lecture, you can replace the `for`-loops with `cilk_for`-loops to parallelize `transpose`. Running `./transpose 10000` will transpose a 10000×10000 matrix.

Checkoff Item 3: Parallelize `transpose` using `cilk_for`, and report your execution time with input size 10000 for 1, 4, and 8 Cilk workers.

2.2 The Cilksan race detector

The Cilksan race detector allows you to check whether your Cilk program has a determinacy race. It provides detailed output that specifies the line numbers of two memory accesses, often a read and a write, that were involved in the race.

Compile and run the `qsort-race` program in the `qsort-race/` subdirectory. This code was parallelized by naively adding `cilk_spawn` and `cilk_scope` to a serial quicksort program. This code has a race!

Before you run Cilksan, take a look at the quicksort code and see if you can identify the determinacy race. See if you can expose the race condition by running a few tests on `telerun`, too. Don't be discouraged if you are unable to expose the race by running tests — but also do not allow yourself to be fooled! This code does, indeed, have a race. Like many subtle determinacy races, the one present in quicksort is difficult to elicit and identify without the use of tools.

We can use Cilksan to detect the race by recompiling the code as follows:

```
$ make -B CILKSAN=1
```

The `-B` flag tells `make` to rebuild all targets. It is practically the same as running `make clean` right before.

When running the newly compiled binary, Cilksan outputs its report to `stderr`. We can now expose this race on a fairly small input of 10 elements:

```
$ ./qsort-race 10 1
```

Checkoff Item 4: Use Cilksan to find this race. Then, fix the race and use Cilksan to confirm that no more races exist. Report the line numbers in the code where the read/write race occurs, and give a brief description of what was happening to cause this race.

2.3 The Cilkscale scalability analyzer

The `qsort/` subdirectory contains `qsort`, another parallel quicksort program. This version of quicksort should not have any races, but you should of course verify this by using Cilksan.

You can use Cilkscale to analyze the scalability of this quicksort program. We can build `qsort` with Cilkscale as follows:

```
$ make -B CILKSCALE=1
```

Since Cilkscale uses timing measurements to compute parallelism, it is usually a good idea to run it on a quiesced machine — such as those provided in the telerun job queue. For this assignment, however, we recommend you run Cilkscale locally on your machine instead of using `telerun` to avoid clogging the queue.

Checkoff Item 5: Report the parallelism computed by Cilkscale on the quicksort program for a few different sized inputs.

Cilkscale's command-line output includes work and span measurements for the Cilk program in terms of empirically measured times. Parallelism measurements are derived from these times. For some programs, such as `fib`, there may be some variability in the reported parallelism numbers.

Checkoff: Explain your responses to Checkoff Items 1–5 to a TA. Hop in the queue and read the following explanation while you wait.

A simple struct `wsp_t` contains the number of nanoseconds for work and span. This data is collected immediately before and after the `sample_qsort()` execution at lines 124 and 126 of `qsort.c`. Then these two measurements are subtracted and dumped (in line 134) to `stdout` in CSV format, with the first column being the label of the measurement. At the end, the same measurements are output for the program as a whole with an empty label. The final measurement includes all the setup and teardown code, which pollutes the measurement we are interested in. Because the dump to `stdout` can interleave with other program output, you might want to set the environment variable `CILKSCALE_OUT=<filename>.csv` to redirect Cilkscale output to a specific file (you will only be able to access that file when running Cilkscale instrumented programs locally — `telerun` currently doesn't return output files).

In addition to a span column, you are also seeing a "burdened span" column. Burdened span accounts for the worst possible migration overhead, which can come from work-stealing and other factors. If you are interested in how Cilkscale works, you can check out the paper on Cilkview, Cilkscale's predecessor.

3 Homework: N queens problem and reducers

The N queens problem is to place N queens on a $N \times N$ chessboard such that no queen attacks another (i.e., no two queens are found in any row, column, or diagonal). Review part of Lecture 2: Bit Hacks to get a feel for how a backtracking procedure works for the N queens problem.

For this homework, our recursive implementation in `queens.c` (only 12 lines of code!) looks for all possible solutions and appends them to a list. It works for $N = 8$, a standard 8×8 chessboard. We chose $N = 8$ for two reasons:

1. *Tractable number of solutions.* $N = 8$ leads to only 92 distinct solutions. $N = 16$, for example, leads to $14,772,512$ distinct solutions.
2. *Simple board representation.* The 8×8 squares of the chessboard fit into the bits of a `uint64_t`.

As shown in the relevant slides for Recitation 2, we can represent the board as 3 bit vectors `down`, `left`, and `right` of size N, N, and N only. This result was invented by Tony Lezard and can be found in an email from 1991:

```
Path: gmdzi!unido!mcsun!uknet!slxsys!ibmpcug!mantis!tony
From: to...@mantis.co.uk (Tony Lezard)
Newsgroups: rec.puzzles
Subject: Re: 8 Queens (NO *SPOILER*)
Message-ID: <eeFmBB1w164w@mantis.co.uk>
Date: 18 Nov 91 18:47:49 GMT
References: <1991Nov16.033939.70781@cs.cmu.edu>
Organization: Mantis Consultants, Cambridge. UK.
Lines: 40
redm...@cs.cmu.edu (Redmond English) writes:
> I wrote a little C program a while ago, which after exhaustively testing
> every position with exactly one queen on each rank (taking about 1 minute
> on an 8MHz 68000 machine), came up with 92 solutions.
Erk. Exhaustive testing? Count me out on that. The following program,
written in ANSI C, uses a backtracking algorithm and gives the correct
answer instantaneously:

#include <stdio.h>
void try(int, int, int);
static int count = 0;

void main() {
   try(0,0,0);
   printf("There are \%d solutions.\n", count);
}

void try(int row, int left, int right) {
   int poss, place;
   if (row == 0xFF) ++count;
   else {
      poss = ~(row|left|right) & 0xFF;
      while (poss != 0) {
         place = poss & -poss;
         try(row|place, (left|place)<<1, (right|place)>>1);
         poss &= ~place;
      }
   }
}


ObPuzzle: Generalize the above to n queens.
--
Tony Lezard IS to...@mantis.co.uk OR tony\%man...@uknet.ac.uk
```

```
OR EVEN ar...@phx.cam.ac.uk if all else fails. Great! Kept my .sig down to two
lines!
```

A more updated version of the code, written by Prof. Leiserson, can be found in the course website.

Our `queens` implementation uses three bit vectors, where `0`s represent available squares and `1`s represent unavailable squares. Therefore, the `down` (equivalent to `row`), `left`, and `right` bit vectors start as all `0`s and get filled with `1`s as the queens are placed.

For Fun: Here is Marcel van Kervinck's one-liner for solving the queens problem. Try running it and deciphering it if you want!

```
01 t(a,b,c){int d=0,e=a&~b&~c,f=1;if(a)for(f=0;e-=d,d=e&-e;f+=t(a-d,(b+d)*2,(c+d)/2));
02 return f;}main(q){scanf("%d",&q);printf("%d\n",t(~(~0<<q),0,0));}
```

3.1 It's a race, but no one is winning.

Compile and run `./queens`. Record the serial execution time.

Let's see if we can get any speedups from parallelization. In the `queens` function, add `cilk_spawn` and `cilk_scope` keywords to parallelize the recursive calls to `queens`. Compile and run `./queens` with `telerun --cores 8`.

When running a Cilk program, you may specify the number of worker threads you'd like to use by setting the environment variable `CILK_NWORKERS`.

Write-up 1: What happened when you ran `./queens`? Did parallelization do what you expected?

Something went wrong. If it didn't, try running it again: the failure can be non-deterministic. Let's use the Cilksan tool to detect the problem. Cilksan will take too long to finish with the current build, so enabling Cilksan will disable the loop in `main` which runs `run_queens I` times.

Now compile with Cilksan. The compiler flag `-fsanitize=cilk` triggers compilation with Cilksan. You can build `queens` with it by running `make -B CILKSAN=1`. The resulting binary will run with Cilksan code built-in.

Write-up 2: Was Cilksan able to detect the problem? Describe the race condition briefly in words, and report the relevant line numbers for the read and write involved in the race.

For more information on the determinacy race detection algorithm used by Cilksan, please refer to the determinacy race detection paper.

3.2 Fixing the race

Remove `cilk_spawn` and `cilk_scope` keywords from the `queens` function. Before re-parallelizing our code, we want to make any accesses to the list of solutions, `board_list`, thread-safe.

Here's one strategy that doesn't require locks: instead of passing `board_list` to the recursive calls to `queens`, we can create a bunch of temporary `BoardList` objects and pass a different one to each recursive thread. When the threads synchronize, all of the temporary lists will have been filled, and we can concatenate them all together with the original `board_list`.

In `board.c`, implement

```
01 void merge_lists(BoardList* left, BoardList* right);
```

The function should merge `right` into `left`. For example, say `left` has 3 nodes and `right` has 5 nodes. After a call to `merge_lists(left, right)`, `left` should have 8 nodes and `right` should be empty. We'll use this call to concatenate several lists together. Remember to handle cases where one or both of the lists are empty and to update all 3 fields of `BoardList`.

Write-up 3: What is the most efficient way to concatenate two singly linked lists? What is the asymptotic running time of your `merge_lists` function?

Now, implement the strategy of passing in temporary lists to the recursive calls and merging them at the end. Compile and run `./queens` to make sure your serial code is still correct. If it is, re-parallelize the code with `cilk_spawn` and `cilk_sync`. Run Cilksan to verify that there are no races. Then record the parallel execution time.

Write-up 4: Briefly explain your implementation. How does the parallel code perform compared to the serial code? Try to explain any difference in performance.

Write-up 5: (Optional) As with fib, coarsen the recursion of `queens` so that if it hits a base case, then it uses the original implementation. Record the new parallel execution time. What is the base case you used? How does the parallel code with coarsening perform compared to the parallel code without? Try to explain any difference in performance. How does it perform compared to the serial code?

3.3 Cilk reducers

Manually making the empty lists for each recursive spawn, and then manually merging them back together was annoying. Thankfully Cilk has something called reducers to make this much easier. They work similarly to the code you just wrote - give each strand an empty list to modify, and then merge the lists together when the strands sync. Once you define for Cilk an empty list and how to merge lists, Cilk handles the rest. Reducers are more general than just lists though. To use a reducer for integer sums, you'd define zero as a blank copy, and addition as the merge step. Then you can write this beautiful race free code:

```
long_reducer sum;
cilk_for (int i = 0; i < N; i++) {
    sum += A[i];
}
```

Background

Cilk provides a unique programming construct called a ***reducer hyperobject***, or ***reducer***. Conceptually, a reducer is a variable that can be safely used by multiple Cilk strands running in parallel. In a parallel execution of a program, the Cilk runtime maintains multiple ***views*** of a reducer. The runtime eliminates the possibility of a race by ensuring that each view can be accessed by only one Cilk strand at a time. After all parallel strands have been synchronized (i.e., after a `cilk_scope` or `cilk_sync`), Cilk guarantees that all views of a reducer have been merged together into a single view. In Cilk, reducers are based on "monoids," and thus Cilk guarantees that a parallel execution of a program updates a reducer and merges views in a way that is equivalent to a serial execution of the same program.

A ***monoid*** is an algebraic structure on a set of elements T with an associative binary operator $\oplus$ (for us this is merging two lists) and an identity element e (for us this is an empty list). As a simple example, for an integer sum monoid, T is $\mathbb{Z}$ (the set of integers), $\oplus$ is $+$ (integer addition), and the identity is $e = 0$.

For example, consider a program with a reducer X with initial value x_0, and suppose that a serial execution of the program performs a sequence of updates $x_1, x_2, \ldots, x_7$ to X. The serial execution combines these updates one at a time, i.e., it produces a final value for X of

$$(((((((x_0 \oplus x_1) \oplus x_2) \oplus x_3) \oplus x_4) \oplus x_5) \oplus x_6) \oplus x_7) \ . \tag{1}$$

Because $\oplus$ is associative and has an identity element e, however, other ways to combine these updates also produce the same final result. For example, one could also combine updates as follows:

$$((((x_0 \oplus x_1) \oplus x_2) \oplus x_3) \oplus x_4) \oplus (((e \oplus x_5) \oplus x_6) \oplus x_7) \ , \tag{2}$$

and still produce the same result. The original serial execution order in Expression (1) uses a single ***view*** of X, i.e., updates occur sequentially. In contrast, the execution represented by Expression (2) uses two views, one for value $(x_0 \oplus x_1 \oplus x_2 \oplus x_3 \oplus x_4)$, and one for the value $(x_5 \oplus x_6 \oplus x_7)$. Having two views allows two processors to update X in parallel without races.

More generally, a parallel execution may create more reducer views and combine them in more complicated but equivalent ways. For example, Expression (3) uses three views, and Expression (4) uses five views:

$$(((x_0 \oplus x_1) \oplus x_2) \oplus ((e \oplus x_3) \oplus x_4)) \oplus (((e \oplus x_5) \oplus x_6) \oplus x_7) \ , \tag{3}$$

$$((x_0 \oplus ((e \oplus x_1) \oplus x_2)) \oplus ((e \oplus x_3) \oplus x_4)) \oplus (((e \oplus x_5) \oplus (e \oplus x_6)) \oplus x_7) \ . \tag{4}$$

Note that reducers can work for noncommutative operations like matrix multiplication. Cilk is careful to always merge the left and right operands correctly.

In summary, reducers exhibit several attractive properties:

- Multiple strands can access a reducer without races.
- Reducers are shared without the need for mutual exclusion locks.
- Reducers can be used without significantly restructuring existing code.
- Defined and used correctly, reducers retain serial semantics. The result of a Cilk program that uses reducers is the same as that of the serial version, and the result does not depend on the number of processors or how the work is scheduled.
- Reducers are implemented efficiently, incurring little or no runtime overhead for synchronization. The cost of accessing a reducer is a hash-table lookup.

Implementing your own reducer

Making a reducer in Cilk is simple: you need a function to set the identity, and a function to merge two views. Here's an example reducer to add up an array of long integers. Currently the arguments have to be passed in as `void*` and then cast to the reducer type.

```
//Set the reducer to zero, the identity element
void zero_long(void *view) {
    *(long *)view = 0;
}
//left += right, the merge step
void add_long(void *left, void *right) {
    *(long *)left += *(long *)right;
}
...
long cilk_reducer(zero_long, add_long) sum;
cilk_for (int i = 0; i < N; i++) {
    sum += A[i];
}
```

Since list concatenation is associative, we can use a reducer to assemble the list of boards that `queens()` generates. Let's implement our own `BoardListReducer` in 5 easy steps:

1. Define the behavior of the reducer in 2 functions:

   ```
   // Evaluates *left = *left OPERATOR *right.
   void board_list_reduce(void* left, void* right) { ... }

   // Sets *value to the the identity value.
   void board_list_identity(void* value) { ... }
   ```

   Remember to cast the `void*` arguments to `BoardList*` before using them. Note that you are defining these functions so that they can be used by the Cilk runtime system. You will not call them explicitly in your code.

   In the future remember that right is deleted after reducing. Any memory allocated by it needs to be freed. That won't matter here as we're merging all of right's nodes into left, so we shouldn't delete them.

2. Define the reducer type (not strictly necessary but helpful):

   ```
   typedef BoardList cilk_reducer(board_list_identity, board_list_reduce) BoardListReducer;
   ```

3. Initialize the reducer as a global (or local, but passing pointers to reducers can be tricky, see how to do it here) variable:

   ```
   BoardListReducer board_list_reducer;
   ```

4. Inside parallel calls, add items to the board list reducer as if it was a board list:

   ```
   append_node(&board_list_reducer, cur_board);
   ```

   Note that `board_list_reducer` acts just like a `BoardList` - you can access its size, head, or tail like `board_list_reducer.head`, and getting its address yields a `BoardList*`, not a `BoardListReducer*`.

5. Once all threads modifying the reducer have been synced, you can safely use the final value of the reducer.

Additional documentation on OpenCilk reducers can be found here.

Note that you are no longer supposed to pass a `BoardList* board_list_ptr` when calling `queens` (this does not use the reducer you just implemented!). Instead just append to your global reducer. Cilk will access the underlying view and use the `board_list_identity()` or `board_list_reduce()` functions to create a new view of the underlying `BoardList` object or merge two existing views, respectively, when it needs to.

Write-up 6: Use a reducer to parallelize `queens`. Verify that the answers you're getting are consistent with the serial code from before. Validate you have no races with `make -B CILKSAN=1 && ./queens`

Write-up 7: Build without instrumentation with a normal `make -B`. Record the parallel execution time using `telerun --cores 8 ./queens`. How does the parallel code with reducers perform compared to the parallel code without reducers?

4 Homework: Parallelizing Floyd-Warshall

The all-pairs shortest path problem is a classic algorithms problem. The goal is to compute the minimum distance between every pair of vertices in a graph, where distance is defined as the sum of edge weights in a path that connects a pair of vertices. A standard algorithm for solving this is known as Floyd-Warshall, which initializes a distance matrix with the weighted graph adjacency matrix and iteratively "relaxes" its edges. If you don't recall this algorithm, now is a good time to pull out your handy copy of *Introduction to Algorithms* by CLRS.[1]

[1] Recall any of the authors?

The relaxation step of Floyd-Warshall is given by:

$$D(i,j) = \min\{D(i,j), D(i,k) + D(k,j)\}$$

for a triple (i,j,k) of vertex indices, where D is a matrix containing the currently best known pairwise distances. Essentially, the relaxation step checks if the current best path from vertex i to j is shorter than by going through some intermediate point k. The Floyd-Warshall algorithm is given by the triple loop around this relaxation step as shown in the following pseudocode:

```
// initially, "A" is the weighted adjacency matrix of the graph
// (distances between unconnected vertices are initialized to +INF)
for (int k = 0; k < N; k++) {
    for (int i = 0; i < N; i++) {
        for (int j = 0; j < N; j++) {
            A[i][j] = min(A[i][j], A[i][k] + A[k][j]);
        }
    }
}
```

And that's it!

The next natural step is to construct a parallelized version of Floyd-Warshall. The simplicity of the algorithm, in addition to the power of `cilk_for`, make it a tempting target for parallelization. For the rest of this homework, assume that there are no negative-weight cycles.

Enter the `floyd/` subdirectory, and run `make floyd_w`. Executing `./floyd_w -p` just runs your parallelized code. To check correctness, run `telerun -cores 8 ./floyd_w -p -c`.

Write-up 8: Try replacing each of the `for`-loops separately in `floyd_parallel` with a `cilk_for`. For which of the loops does `telerun -cores 8 ./floyd_w -p -c` fail correctness?

Write-up 9: For which of the `for`-loops, parallelized with `cilk_for` and compiled with `CILKSAN=1`, does Cilksan report a race in the program? Use `./floyd_w -p -m 1 -n 10` when running your program for each loop.

You might notice that the inner loop does in fact have a race condition, but Cilksan does not seem to report it. It turns out that the Cilk compiler is coarsening that loop to the point where there are no parallel iterations (If you aren't convinced, try running it with Cilkscale and looking at the parallelism). To prevent the compiler from coarsening the loop, you can use the directive

`#pragma cilk grainsize 1`. Place this pragma right above the inner most `for`-loop when you make it a `cilk_for`. Retest correctness, and rerun with Cilksan to confirm that you now see a race reported.

You should find that the answers to the previous two write-ups are not the same. In fact, there are two loops that when parallelized create races, but in such a way that even if a race happens, the validity of the program is not compromised. These are referred to as ***benign races***. Parallelize both loops, and use that code for the remainder of this homework.

Write-up 10: Why does parallelizing the loops that cause the benign race in the `floyd_parallel()` (with two loops parallelized) still produce the correct result? Why can't we parallelize the outer-most loop? Assume that there are no negative-weight cycles.

Be careful when answering this question. In particular, explain why the races in `floyd_parallel()` are benign despite the fact that the races among iterations of the loop in the following code are not benign:

```
// Code to compute the minimum of array A
int m = A[0];
cilk_for(int i = 1; i < n; i++) {
    if (A[i] < m) {
        m = A[i];
    }
}
return m;
```

We now want to measure the scalability of `floyd_parallel()` with Cilkscale. To get a measurement of just `floyd_parallel()` apart from the rest of the code, Cilkscale provides `wsp_getworkspan()` to tell Cilkscale when to start and stop monitoring the program, and `wsp_dump()` to collect the results. Uncomment the calls to these functions around the `switch-case` statement in `test()` in `floyd_w.c` when measuring parallelism of `floyd_parallel()` with Cilkscale. Don't forget to use `#pragma cilk grainsize 1` on each loop since we want to make sure the compiler does not coarsen either loop.

Once you are ready, run your parallelized Floyd-Warshall with Cilkscale using

```
$ make -B CILKSCALE=1 && CILK_NWORKERS=1 ./floyd_w -p -m 700 -n 1
```

Record the running time of your program by running:

```
$ make clean && make && telerun --cores 8 time -v ./floyd_w -p -m 1000 -n 1
```

Write-up 11: Compare the reported parallelism by Cilkscale and running times of `floyd_parallel()` with and without the grainsize `#pragma`'s. Explain the differences you see.

On x86-64 machines, naturally aligned loads and stores are atomic up to 64 bits. For example, suppose that a write to a 64-bit location is concurrent with a read to the same location. If the accesses are atomic, the reader obtains either the value originally in the location or the value that the writer wrote. The value it reads is never composed of some bits of each (or an entirely different value altogether). For 128-bit values, however, it could be that the reader sees the high-order 64 bits of the original value and the low-order 64 bits of what the writer wrote.

Write-up 12 (optional): If we change the `typedef` of `DIST` to `__int128` in `floyd_parallel`, are the races still benign or could an incorrect value be produced? Assume that the graph contains no negative-weight cycles. Explain.

Now, assume that the graph contains a negative-weight cycle. Are the races still benign? Will we always obtain the same output as the serial program produces? Explain.

5 Turn-in

When you've written up answers to all of the above questions, turn in your write-up by uploading it to Gradescope, and commit and push your code to your Git repository.