Full Name:

Andrew Id:

15-418/618 Spring 2019
Exercise 2

Assigned:	Mon., Feb. 4
Due:	Fri., Feb. 8, 11:59 pm

Overview

This exercise is designed to help you better understand the lecture material and be prepared for the style of questions you will get on the exams. The questions are designed to have simple answers. Any explanation you provide can be brief—at most 3 sentences. You should work on this on your own, since that's how things will be when you take an exam.

You will submit an electronic version of this assignment to Gradescope as a PDF file. For those of you familiar with the LaTeX text formatter, you can download the template and configuation files at:

http://www.cs.cmu.edu/~418/exercises/config-ex2.tex
http://www.cs.cmu.edu/~418/exercises/ex2.tex

Instructions for how to use this template are included as comments in the file. Otherwise, you can use this PDF document as your starting point. You can either: 1) electronically modify the PDF, or 2) print it out, write your answers by hand, and scan it. In any case, we expect your solution to follow the formatting of this document.

Barrier Synchronization

The following problems are inspired by the code shown in Slides 48 and 49 in Lecture 05. In the following, we show only key parts of the code. You can get the complete versions in the directory

http://www.cs.cmu.edu/~418/exercises/ex2-code

We suggest you download this code and study it. You can also compile and run it. (See the `README.txt` file.) Insert print statements into the code to trace the actions of the different threads.

The synchronization code in the slides show implement parts of a grid solver. To focus more directly on synchronization issues, we will adapt the code for the following, highly contrived, application.

Let B denote a *batch size* and θ, with $0.0 < \theta < 1.0$ denote a *threshold.* Suppose we perform a series of *phases*, where in each phase we compute B random numbers, each ranging between 0.0 and 1.0, and take their average a. How many phases will it take to reach a case where $a \leq \theta$, and what is the achieved value of a? We assume the "random" numbers are actually generated by a pseudo-random generator, and so, assuming we always start with the same seed, the result will be deterministic.

Our implementation has B threads running concurrently, using barrier synchronization to keep them operating on the same phase. The global variables are as follows:

```
//// Global variables ////
// Read-only
float target_average; // Convergence goal
int batch_size;       // Number in batch
// Only written by single thread
long pcount = 0;      // Number of phases required
// Read-write
float phase_sum;      // Sum of all values in current phase
```

A Three-Barrier Solution

The following is the thread procedure for a version using three barriers, similar to the code in Slide 48. The full program is in the file `rconverge1.c`. There are barriers between the three actions performed on shared variable `phase_sum` in each phase: setting it to zero, incrementing by the random value, and testing for convergence. The call `uniform()` returns a (pseudo-)random number between 0.0 and 1.0.

```
//// Thread procedure # 1 ////
void *thread_proc(void *ival) {
    long myid = (long) ival;
    bool myconverged = false;
    long count = 0;
    while (!myconverged) {
        count++;

        phase_sum = 0.0;                // Action 1: Set to zero

        barrier();   // Barrier #1

        float myval = uniform();
        atomic_add(&phase_sum,  myval); // Action 2: Increment

        barrier();   // Barrier #2

        myconverged =                   // Action 3: Test
            (phase_sum/batch_size) <= target_average;

        barrier();   // Barrier #3

    }
    if (myid == 0)
        pcount = count;
    return NULL;
}
```

Problem 1: Understanding the Three-Barrier Solution

For each of the three barriers, what can go wrong if you eliminate it? You can try this with the actual code. You may want to insert print statements to track what happens. Describe the behavior you observe, and explain why it is happening.

A. Barrier #1

B. Barrier #2

C. Barrier #3

Single-Barrier Solutions

As Slide 49 demonstrates, it is possible to transform the three-barrier solution into one that uses only one barrier, using a technique known as *software pipelining*. Whereas the three-barrier solution does one phase per iteration, stepping through the three actions for that phase, the pipelined version performs the actions for three different phases during each iteration. It does so by maintaining multiple versions of the shared, global variables and making sure that the different actions being performed within the iteration operate on different versions. The number of versions that must be maintained is referred to as the *pipeline depth*.

For our case, we only need to have multiple copies of the variable phase_sum:

```
float phase_sum[PIPEDEPTH];  // Sum of all values in current phase
```

The quantity PIPEDEPTH is a compile-time constant. In the version provided to you, it is set to 128.

First, we will explore a version of the thread procedure that is easier to understand. The thread procedure is shown below. The full code is in the file rconverge2.c. We see that it maintains three indices into the phase_sum, one for each of the basic operations. These indices are incremented by one for each phase, modulo the pipeline depth.

```
//// Thread procedure #2 ////
void *thread_proc(void *ival) {
    long myid = (long) ival;
    bool myconverged = false;
    int previndex = 0;
    int index = 1;
    long count = 0;
    phase_sum[1] = 0;
    barrier();       // Barrier #1

    while (!myconverged) {
        count++;

        int nextindex = (index+1) % PIPEDEPTH;
        phase_sum[nextindex] = 0.0;            // Action 1: Set to zero

        float myval = uniform();
        atomic_add(&phase_sum[index],  myval); // Action 2: Increment

        myconverged =                          // Action 3: Test
            (phase_sum[previndex]/batch_size) <= target_average;

        barrier();   // Barrier #2

        previndex = index;
        index = nextindex;
    }
    if (myid == 0)
        pcount = count - 1;
    return NULL;
}
```

Problem 2: Understanding the First Single-Barrier Solution

A. Barrier #1 is called only at the beginning of the procedure. Describe its purpose. What incorrect behavior can arise by eliminating it?

B. Why does global variable `pcount` get set to `count - 1` in this code, but to `count` in the earlier code?

C. How small can constant `PIPEDEPTH` be set and still get correct behavior? Explain why this is the case.

D. Suppose you swap the order of Barrier #2 with the code implementing Action 3. Explain why the program still works for the default version of `PIPEDEPTH`.

E. With this swapped version, how small can you set `PIPEDEPTH` and still get the correct answer? Explain.

Problem 3: Understanding the Second Single-Barrier Solution

Our final version of the code resembles that seen in Slide 49. The code is in the file `rconverge3.c`.

```
//// Thread procedure #3 ////
void *thread_proc(void *ival) {
    long myid = (long) ival;
    bool myconverged = false;
    int index = 1;
    long count = 0;
    phase_sum[1] = 0;
    barrier();       // Barrier #1

    while (!myconverged) {
        count++;

        int nextindex = (index+1) % PIPEDEPTH;
        phase_sum[nextindex] = 0.0;            // Action 1: Set to zero

        float myval = uniform();
        atomic_add(&phase_sum[index],  myval); // Action 2: Increment

        barrier();   // Barrier #2

        myconverged =                          // Action 3: Test
            (phase_sum[index]/batch_size) <= target_average;

        index = nextindex;
    }
    if (myid == 0)
        pcount = count;
    return NULL;
}
```

This version maintains only two indices, and the barrier synchronization has been put before Action 3. Let us explore this code:

A. Explain why the code does not require the third index `previndex`

B. In this version, the global variable `pcount` is set to the local value `count`, without decrementing it. Explain why this is the correct result.

C. How small can constant `PIPEDEPTH` be set and still get correct behavior? Explain why this is the case.

Cilk Scheduling

The following problems are based on the presentation of the Cilk programming environment from Lecture 06.

Suppose we are running a program that uses the Cilk mechanisms for fork-join parallelism. Assume the following:

- The system is running two threads
- Every execution of `cilk_spawn` requires 1 millisecond. The executing thread will push its continuation at the top of its queue and begin executing (after 1ms) the spawned function ("child first" scheduling).
- One thread can steal work from the queue of another. It always steals from the bottom of the queue. Stealing requires 2 milliseconds, and then the thread can begin performing whatever operation is indicated by the steal.
- Procedure `foo` requires 3 milliseconds to complete.
- All other operations require only a negligible amount of time.

As an example, consider the following code

```
void simple() {
    cilk_spawn foo(1);
    foo(3);
    cilk_synch;
}
```

If Thread 1 starts executing `simple`, we can trace its execution as follows:

Time	Thread 1	Thread 2
0	Spawn `foo(1)`; Push `foo(3)`	Idle
1	Execute `foo(1)`	Steal `foo(3)`
2	Executing	Stealing
3	Executing	Execute `foo(3)`
4	Idle	Executing
5	Idle	Executing

Problem 4: Divide and Conquer Parallelism

Consider the following function for executing multiple copies of function `foo` via a series of forks that repeatedly split the problem in half, following a divide-and-conquer control structure.

```
void rfor(int start, int last) {
    if (start == last)
        foo(start);
    else {
        int middle = (start + last)/2;
        cilk_spawn rfor(start, middle);
        rfor(middle+1, last);
    }
    cilk_synch;
}
```

A. Fill in the following table, showing how two threads would handle the execution of `rfor(0,3)`. (You may not require all of the rows in the table.)

Time	Thread 1	Thread 2
0		
1		
2		
3		
4		
5		
6		
7		
8		
9		
10		
11		
12		
13		
14		
15		

B. What do you get as the computed speedup for this execution?

Problem 5: Iterative Parallelism

Consider the following function for executing multiple copies of function `foo` by having a series of calls to function `ifor`, each spawning one execution of `foo`. (Although the function is written as a recursive procedure, you can see that the sequence of spawns is identical what would occur if they were done with a single loop, as shown in Lecture 06, starting with Slide 39.)

```
void ifor(int start, int last) {
    if (start == last)
        foo(start);
    else {
        cilk_spawn ifor(start, start);
        ifor(start+1, last);
    }
    cilk_synch;
}
```

A. Fill in the following table, showing how two threads would handle the execution of `ifor(0,3)`:

Time	Thread 1	Thread 2
0		
1		
2		
3		
4		
5		
6		
7		
8		
9		
10		
11		
12		
13		
14		
15		

B. What do you get as the computed speedup for this execution?

C. What insights do these to example give you regarding the best way to use Cilk in exploiting fork-join parallelism?