Full Name:

Andrew Id:

15-418/618 Spring 2020
Exercise 2
SOLUTION

Assigned:	Fri., Jan. 31
Due:	Fri., Feb. 7, 11:00 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.

Problem 1: Instruction-Level Parallelism

The following set of problems concern instruction-level parallelism and the limitations to performance of loop code imposed by data dependencies and resource limitations (as expressed by throughput bounds).

Calculating the distance between two vectors is an important operation for many linear algebra packages and can be defined as:

$$\sqrt{\sum_{i=0}^{N-1} (A[i] - B[i])^2}.$$

Consider the following C code for calculating the squared distance between two vectors.

```
float distance(float A[], float B[], int N) {
        int i = 0;
        float total = 0;
        while (i < N) {                          // Performed by integer unit
                float a = A[i];                  // Load
                float b = B[i];                  // Load
                float diff = a - b;              // FP Add
                float squared = diff * diff;     // FP Multiply
                total = total + squared;         // FP Add
                ++i;                             // Performed by integer unit
        }
        return total;
}
```

Suppose for the following questions you have a machine with multiple execution units of the following types:

Load/Store: Performs load and store operations. Can perform its own address arithmetic.

Floating-point Adder: Performs floating-point addition and subtraction

Floating-point Multiplier: Performs floating-point multiplication

Integer: Performs integer operations, including addition and comparison.

The processor has the following combination of execution units:

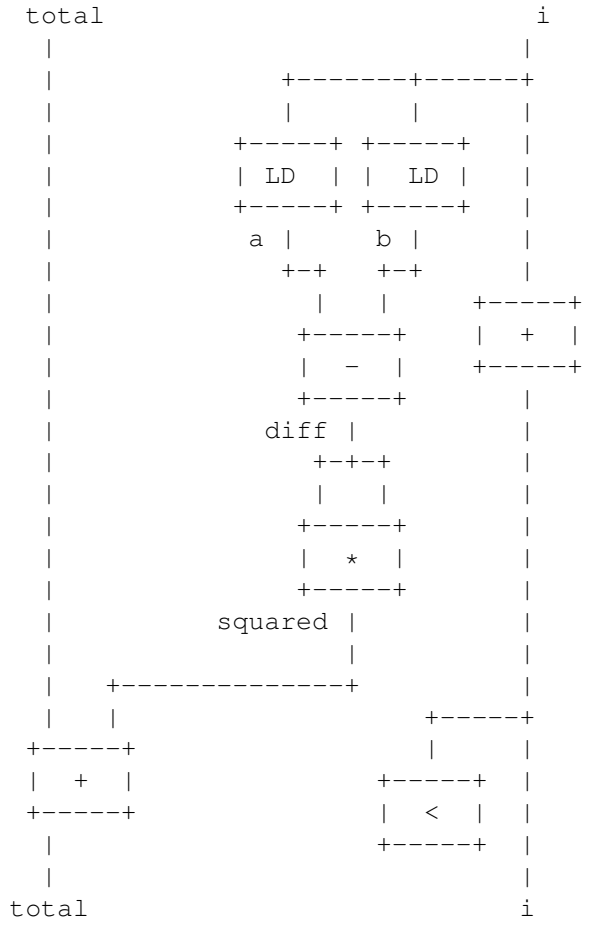
Unit	Count	Latency
Load/Store	1	4
Floating-point Add	2	1
Floating-point Multiply	1	2
Integer Add	2	1

Each of the multi-cycle units is *fully pipelined*, able to begin a new operation on each clock cycle.

You should also assume the following:

- The compiler produces optimized code. All local variables are held in registers.
- The machine described has a sophisticated CPU with support for out-of-order execution, pipelining, branch prediction, and speculation.
- N is very large ($> 10^6$).
- There are no cache misses.
- The only limits to average program performance are 1) the latencies due to data dependencies on loop-carried variables, and 2) the throughput limits of the functional units.

A. Draw the dataflow diagram for the operations in the loop of `distance`. Use the local variable names as labels.



B. Which local variables create loop dependencies?

Only `total` and `i`.

C. What latency bound does the loop of `distance` impose on the minimum average number of cycles for each element computed? Justify your answer.

Both the addition that updates `total` and the one that increments `i` create latency bounds of 1. Only these two updates require the results from previous iterations, and hence they are the only ones constraining the overall latency caused by data dependencies.

D. What throughput bound does the set of available execution units impose on the loop of `distance`? Explain.

The two load operations, sharing a single load/store unit, create a bound of two cycles. All other operations (FP Add, FP Multiply, integer) create bounds of one cycle.

E. If you could add one additional execution unit to the mix above, what would it be? How would that change your program performance?

If there were one more load/store unit, the throughput bound would drop to one cycle, matching the latency bound.

Barrier Synchronization

The following set of problems are inspired by the code shown in Slides 48 and 49 in Lecture 05. In the following, we show only key parts of our version 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 results 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 2: 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

The code terminates early, with an incorrect number of iterations. Without this barrier, some threads may still be setting `phase_sum` to zero while others are incrementing it, resulting in a sum that is too low, and hence meets the termination standard prematurely.

B. Barrier #2

The code hangs up. Some threads start performing the termination test before others have added their contributions to `phase_sum`. As a result, these threads may think the termination condition has been satisfied and exit their loops. The remaining threads then get hung up at a barrier synchronization, because the terminated threads never reach the barrier.

C. Barrier #3

The code hangs up. Some threads fail the termination test and continue onto the next phase, setting `phase_sum` to zero. Meanwhile, some of the slower threads see this small value as an indication that the termination condition has been met. The slower threads exit the loop, and the remaining threads get hung up at a barrier.

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*. The idea is to maintain multiple versions of the shared, global variables, and make sure that the different actions being performed within the loop operate on different versions. The number of versions required 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 3: 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?

This barrier makes sure that `phase_sum[1]` is set to zero before the phase computation begins. This must take place before the threads start incrementing this copy of the sum.

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

The termination condition is based on the previous version of the sum, corresponding to phase `count - 1`.

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

It can be set to three. At any given time, the program references just three versions of the sum. Setting it lower causes an overlap between the ones being referenced by Actions 1 and 3.

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`.

Yes the program still works. The different threads may start operating on different phases, with some doing the convergence test and others setting `phase_sum` to zero. But, these will reference different versions of the sum.

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

The pipeline depth must be at least four, to handle three different sets of indices operating across two phases.

Problem 4: Understanding the Second Single-Barrier Solution

Our final version of the code resembles that seen in Slide 49:

```
//// 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`

The code does the convergence test on the version of the sum indexed by variable `index`. It can safely do so, because the preceding barrier ensures that the incrementing has been completed.

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.

The convergence test is performed on the most recent version of the sum, corresponding to phase number `count`.

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

The minimum value is three. Although there are only two indices for each thread, these can be offset from one another.

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 specified in the record.
- The 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 5: 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, analogous to the control structure of quicksort:

```
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	Spawn `rfor(0,1)`; push `rfor(2,3)`	Idle
1	Spawn `rfor(0,0)`; push `rfor(1,1)`	Steal `rfor(2,3)`
2	Execute `foo(0)`	Stealing
3	Executing	Spawn `rfor(2,2)`; push `rfor(3,3)`
4	Executing	Execute `foo(2)`
5	Execute `foo(1)`	Executing
6	Executing	Executing
7	Executing	Execute `foo(3)`
8	Idle	Executing
9	Idle	Executing
10		
11		
12		
13		
14		
15		

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

Serial execution would be $4 \cdot 3 = 12$ cycles. Parallel execution $= 10$ cycles. Speedup = $12/10 = 1.2$.

Problem 6: Iterative Parallelism

Consider the following function for executing multiple copies of function `foo` by having the main thread spawning off series of threads, each executing one instance of `foo`. (Although the function is written as a recursive procedure, you will see that the sequence of spawns is identical what would occur if they were done as a single loop, as in shown in Lecture 06, starting with Slides 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	Spawn `ifor(0,0)`; push `ifor(1,3)`	Idle
1	Execute `foo(0)`	Steal `ifor(1,3)`
2	Executing	Stealing
3	Executing	Spawn `ifor(1,1)`; push `ifor(2,3)`
4	Steal `ifor(2,3)`	Execute `foo(1)`
5	Stealing	Executing
6	Spawn `ifor(2,2)`; push `ifor(3,3)`	Executing
7	Execute `foo(2)`	Steal `ifor(3,3)`
8	Executing	Stealing
9	Executing	Execute `foo(3)`
10	Idle	Executing
11	Idle	Executing
12		
13		
14		
15		

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

Serial execution would be $4 \cdot 3 = 12$ cycles. Parallel execution $= 12$ cycles. Speedup = $12/12 = 1.0$.

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

In these examples, both spawning and stealing have a high cost, relative to the actual computation being performed. This exacerbates inefficiencies caused by wasted activity, in this case, the stealing back and forth of the main block of work. Divide and conquer parallelism provides a cleaner approach to, getting both threads fully occupied quickly, with minimal inefficiency.