Full Name:

Andrew Id:

15-418/618 Spring 2020
Exercise 5

Assigned:	Mon., March 30
Due:	Monday., April 6, 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, configuation, and figure files at:

http://www.cs.cmu.edu/~418/exercises/config-ex5.tex
http://www.cs.cmu.edu/~418/exercises/ex5.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: Locking and Transactional Memory

Consider the following very simple banking system, where all account values are whole dollars.

```
// Account balances in whole dollars
int balance[NACCT];
```

The only supported banking operations are to transfer money between accounts and to check that the sum of all balances equals or exceeds some threshold value.

```
// Transfer money between accounts.
// Do not allow from_acct to become negative
boolean transfer(int to_acct, int from_acct, int amount);

// Check that sum of all accounts equals or exceeds some threshold
boolean check_sum(int threshold);
```

If a transfer operation would cause an account to become overdrawn, function `transfer` should return false without changing any balances. (You can assume that `amount` is not negative.)

Function `check_sum` must be implemented in such a way that it gets a value for the sum that would be a valid snapshot of the state of the balances for some sequentially consistent ordering of the transactions.

1A: A Banking System with Locks

Suppose you are provided a mutual exclusion lock for each account:

```
int lock[NACCT];
```

The library functions for acquiring and releasing a lock have the following prototypes:

```
void acquire(int *lockp);
void release(int *lockp);
```

Fill in code for these two operations using locking. You may not add any additional locks. Make sure your code cannot deadlock. Try to minimize the amount of time that locks are held.

1. Transfer

```
boolean transfer(int to_acct, int from_acct, int amount) {
    boolean ok = true;
    acquire(lock + from_acct);
    if (balance[from_acct] < amount)
    {
      ok = false;
    }
    else
    {
      balance[from_acct] -= amount;
    }
    release(lock + from_acct);

    if (ok)
    {
      acquire(lock + to_acct);
      balance[to_acct] += amount;
      release(lock + to_acct);
    }

    return ok;
}
```

2. Check Sum

```
boolean check_sum(int threshold) {
    int sum = 0;
    for (Int i = 0; i < NACCT; i++)
    {
      acquire(lock + i);
      sum += balance[i];
      release(lock + i);
    }

    return sum >= threshold;
}
```

1B: A Banking System with Transactional Memory

Suppose we have a transactional memory system supporting the following operations:

```
// Begin transaction
void xbegin();

// Abort transaction
void xabort();

// (Attempt to) commit transaction.
// Returns true if successful, false if failed
boolean xend();
```

Rewrite the two functions using transactional memory primitives rather than locks.

1. Transfer

```
boolean transfer(int to_acct, int from_acct, int amount) {
    boolean ok;
    while (true)
    {
      xbegin();
      ok = balance[from_acct] >= amount;
      if (ok)
      {
        balance[from_acct] -= amount;
        balance[to_acct] += amount;
        if (xend())
        {
          break;
        }
      }
      else
      {
        xabort();
      }
    }

   return ok;
}
```

2. Check Sum

```
boolean check_sum(int threshold) {
    int sum = 0;
    while (true)
    {
      sum = 0;
      xbegin();
      for (int i = 0; i < NACCT; i++)
```

```
      {
        sum += balance[i];
      }
      if (xend())
      {
        break;
      }
    }

    return sum >= threshold;
}
```

3. If the system performs millions of transactions per second over thousands of bank accounts, what problem to you forsee for the implementation based on transactional memory? Assume the transactional memory is implemented using lazy versioning and optimistic conflict detection.

Chances are that `check_sum` would keep failing, because ongoing transfers would keep causing writes to the array `balance`.
Lazy versioning means that it won't write to memory directly, namely it won't create too much memory write operations. Optimistic conflict detection means that it only detects conflict on **`xend`**`()`. `check_sum` need a lot of time to operate on `balance`, while `transfer` only need to operate two elements. So when `check_sum` tries to commit, it often will find that some of elements is been modified, and hence failed.

1C: Improving transactional memory performance

Suppose that you can be guaranteed that there will be only one outstanding call to `check_sum` at a time, and that it will be relatively infrequent (around once per hour.) Devise a simple scheme, without any locks, where the system can temporarily force all transfers to wait while a sum is being computed, thereby minimizing the chance of the sum computation failing. You may make use of the following additional global variable:

```
boolean summing = false;
```

1. Transfer

```
boolean transfer(int to_acct, int from_acct, int amount) {
    boolean ok;
    while (true)
    {
      xbegin();
      ok = balance[from_acct] >= amount && !summing;
      if (ok)
      {
        balance[from_acct] -= amount;
        balance[to_acct] += amount;
        if (xend())
        {
          break;
        }
      }
      else
      {
        xabort();
      }
    }

   return ok;
}
```

2. Check Sum

```
boolean check_sum(int threshold) {
    int sum = 0;
    summing = true;
    while (true)
    {
      sum = 0;
      xbegin();
      for (int i = 0; i < NACCT; i++)
      {
        sum += balance[i];
      }
```

```
      if (xend())
      {
        break;
      }
    }
    summing = false;

    return sum >= threshold;
}
```

Problem 2: Processor Scaling

This problem explores the two scaling models presented on slides 5–9 in the lecture on Heterogeneous Parallelism:

http://www.cs.cmu.edu/~418/lectures/21_heterogeneity.pdf.

These are based on the paper "Amdahl's Law in the Multicore Era," by Mark D. Hill and Michael R. Marty of the University of Wisconsin, published in *IEEE Computer*, July, 2008.

In both models, we express computing performance and resources (e.g., chip area) relative to a baseline processor. In scaling to a processor with r resource units (scaled such that the baseline processor has $r = 1$), we obtain a processor with single-threaded performance $perf(r)$ (scaled such $perf(1) = 1$.)

The slides show graphs for the case of $perf(r) = \sqrt{r}$. This function is plausible—it captures the idea that increasing processing resources will improve performance, but not in a linear way. Increasing to large values of r yields diminishing returns in terms of single-threaded processor performance. We will generalize this model to one with $perf(r) = r^{\alpha}$, where α is value with $0.0 \leq \alpha \leq 1.0$. (For example, the slides are based on $\alpha = 0.5$.)

For scaling the system design to use n resource units, two options are considered:

Homogeneous: We create a new processor design that uses r resource units, and populate the system with $p = n/r$ identical processors. In this case, p must be an integer, but r need not be. For example, to get $p = 5$ when $n = 16$, we would have $r = 16/5 = 3.2$.

Heterogeneous: We create a design that uses r resource units (where r is an integer), and then create a system with one of these more powerful processors, plus $n-r$ baseline processors, giving $p = n-r+1$ total processors.

As with the conventional presentation of Amdahl's Law, we assume that the problem to be solved has some fraction f that can use arbitrary levels of parallelism, while the remaining fraction $1 - f$ must be executed sequentially.

With performance normalized to a single baseline processor, Amdahl's Law states that the speedup over the baseline is given by the equation

$$S = \frac{1}{T_{\text{seq}} + T_{\text{par}}} \tag{1}$$

where T_{seq} is the time required to execute the sequential portion of the code and T_{par} is the time required to execute the parallel portion of the code, with both of these using all available resources.

2A: Homogeneous Model Speedup

Slide 6 gives the following equation for the speedup in the homogeneous model:

$$S_{\text{ho}} = \frac{1}{\frac{1-f}{perf(r)} + \frac{f}{perf(r) \cdot \frac{n}{r}}} \tag{2}$$

Explain how Equation 2 follows the form of Equation 1.

1. What resources are used and what is the duration of T_{seq}?

   The one big processor with performance $perf(r)$ is used, and $T_{\text{seq}} = \frac{1-f}{perf(r)}$

2. What resources are used and what is the duration of T_{par}?

   Those $\frac{n}{r}$ baseline processors with performance $perf(r)$ are used, and $T_{\text{par}} = \frac{f}{perf(r) \cdot \frac{n}{r}}$

2B: Heterogeneous Model Speedup

Slide 8 gives the following equation for the speedup in the heterogenous model:

$$S_{\text{he}} = \frac{1}{\frac{1-f}{perf(r)} + \frac{f}{perf(r)+n-r}} \tag{3}$$

Explain how Equation 3 follows the form of Equation 1.

1. What resources are used and what is the duration of T_{seq}?

   The one bit processor with performance $perf(r)$ is used, and $T_{\text{seq}} = \frac{1-f}{perf(r)}$

2. What resources are used and what is the duration of T_{par}?

   The one processor with performance $perf(r)$ and $n - r$ processors with performance $perf(1)$ is used, and $T_{\text{par}} = \frac{f}{perf(r)+n-r}$.

2C: Optimizing Parameter r for the Homogeneous Model

For either of these models, we can find a value r^* that achieves maximum speedup as a function of f, n, and α. Keeping n fixed at 256 but varying parameters α and f, fill in the table below giving the values of r^* and $p = n/r^*$ for the homogeneous model, and the resulting speedups.

Hint: The number of processors $p = n/r$ must be an integer ranging from 1 to n. You can simply compute 2 for all possible values and select the one with maximum speedup.

You may find the following spreadsheet a useful starting point:

http://www.cs.cmu.edu/~418/exercises/ex5-model.xlsx

This spreadsheet generates the graphs shown on slide 9 for the case of $n = 256$. You can adapt this spreadsheet to determine the values of r^*. (The spreadsheet only considers only cases where p is a power of two. You must consider other integer values.)

α	f	r^*	$p = n/r^*$	S_{ho}
0.5	0.700	128.0	2	17.41
0.5	0.970	8.0	32	46.90
0.5	0.995	1.29	199	113.42
0.7	0.700	256.0	1	48.50
0.7	0.970	18.29	14	77.02
0.7	0.995	3.01	85	129.51

2D: Optimizing Parameter r for the Heterogeneous Model

Fill in the table below giving the values of r^* and $p = n - r^* + 1$ for the Heterogeneous model, and the resulting speedups.

α	f	r^*	$p = n - r^* + 1$	S_{he}
0.5	0.700	170	87	33.25
0.5	0.970	72	185	116.62
0.5	0.995	28	229	191.94
0.7	0.700	163	94	71.75
0.7	0.970	65	192	160.18
0.7	0.995	27	230	214.59

Problem 3: Cache Performance of Multi-Threaded Program (MODIFIED)

This problem relates to the image processing task described in Slides 40–46 of Lecture 22:

http://www.cs.cmu.edu/~418/lectures/22_dsl.pdf.

You have been hired by a movie production company to write programs used for rendering images. Each frame is $h \times w$ pixels, with a pixel represented with data type `float`. Your first task is to write filtering programs that apply $k \times k$ filters to an image. Code to do this is shown for on Slide 42 for the case of $k = 3$. The filters you must implement can be divided into two phases: one horizontal filtering stage and one vertical stage, as shown in Slide 43.

Consider the code of Slide 43. In this version, you allocate a temporary buffer large enough to hold all $w \times (h + k - 1)$ pixels of the result generated by the first phase and used by the second. The total work performed is proportional to $2k \cdot w \cdot h$. We call this the *baseline* code.

In the ideal case, the cache would be large enough to generate the temporary buffer and keep it resident in cache while it is being consumed in the second phase. Unfortunately, that is not feasible. You have to work within the following constraints:

- The images are 4K $\times$ 4K, i.e., $w = h = 4096$.
- The machine has 16 cores sharing a single 2 MB (2^{21}-byte) cache.

You therefore consider the code of Slide 46, where you choose a *chunk* size c, and size the temporary buffer to hold $c + k - 1$ rows. The code then processes the image in multiple *passes*. Each pass has two phases, with the first filling the temporary buffer based on $c + k - 1$ rows of the input, and the second generating c rows of the output. By selecting a small enough chunk size, the temporary buffer will be stored in the cache during phase 1 and remain in the cache as it is being used in phase 2.

There are many images to process, and so you plan to use the multiple cores to run multiple threads, each operating on separate images. However, you want to select a chunk size c such that all of the threads will keep their temporary buffers resident in cache.

A. Consider the single-threaded case. Your colleague states that, as long as the chunk size is small enough that you can fit $c + k$ rows in the cache, then you can keep the temporary buffer in the cache. His reasoning is as follows:

- Phase 1: Only one row of the input is needed at a time, and the temporary buffer requires only $c + k - 1$ rows.
- Phase 2: k rows of the temporary buffer are needed at a time to build up the output buffer of c rows.

Assuming the cache uses an LRU replacement policy, and ignoring conflict misses, explain why this estimate may be too optimistic.

B. What do you propose as a safe rule for the chunk size? (Some advice: it's better to keep this simple and conservative. If you have to run a cache simulator to answer this question, you're overdoing it.)

C. How many rows can safely fit in the cache? Remember that the input rows must be padded with an extra $k - 1$ pixels, and you want to avoid false sharing. Assume a cache block size of 64 bytes. You can assume $k \leq 11$.

D. When the caching works out, you find that the total amount of work is a good predictor of the program execution time. Give a formula, in terms of c and k, of how much work the chunked version must do, relative to the baseline version.

Baseline:

Chunked:

Ratio:

E. Consider the single-threaded case, and consider $k \in \{3, 5, 7\}$. For each of these cases, what would be the maximum chunk size? Give an estimate of the execution time, relative to the baseline running on a machine with a very large cache.

$k = 3$**:**

$k = 5$**:**

$k = 7$**:**

F. For values of $k \in \{3, 5, 7\}$, determine the number of threads that would lead to optimum performance, considering that increasing the number of threads requires decreasing the chunk size. What would be the chunk size? Give an estimate of the execution time, relative to the baseline running on a single-threaded machine with a very large cache.

$k = 3$**:**

$k = 5$**:**

$k = 7$**:**