TU WIEN Informatics Parallel Computing

Research Group for Parallel Computing
Faculty of Informatics
TU Wien

Advanced Multiprocessor Programming

Summer term 2023
Theory exercise 1 of 2

Issue date: 2023-03-27
Due date: 2023-04-24 (23:59)

Total points: 38 (+ 2 bonus)

1 Amdahl's Law

Ex 1.1 (2 points)

Suppose 98% of a program's execution time is perfectly parallelizable.

a) What is the overall speedup that can be achieved by running said program on a machine with 96 processing cores?

b) Provide a tight upper limit for the overall speedup that can be achieved for this program (on any machine).

Ex 1.2 (2 points)

The company you work for is about to upgrade all of their servers to slower but more energy efficient processors with much higher core counts. Suppose the sequential part of the main program running on these servers accounts for 20% of the program's computation time. In order to compensate for the slower processing speeds and to better utilize the higher core count, management asks you to optimize the sequential part of said program — make it run k times faster —, s.t. the revised program scales 3 times better with a given number of processing cores meaning the *relative (!)* speedup of the revised program is 3 times higher than that of the original program. What value of k should you require for a given number of processor cores n?

Ex 1.3 (2 points + 2 bonus points)

Let $p = 0.7$ denote the perfectly parallelizable fraction of a given program. Suppose there are two *mutually exclusive* optimizations to be done on this program, meaning if one optimization is done, the other is no longer available:

(I) Improve the execution time of the parallelizable part p by a factor of 7.

(II) Improve the execution time of the sequential part $1 - p$ by a factor of 3.

a) Provide the relative speedup $s(n)$ of the original program, as well as of the program after optimization (I) $s^{(I)}(n)$ and the one after (II) $s^{(II)}(n)$ for general processor core count n and for $n = 10$. Calculate the execution times $T_n^{(I)}$ and $T_n^{(II)}$ (of the two optimized programs) for $n = 10$ in terms of the original workload normalized to 1. (*Hint:* $T_1^{(I)} = 0.4$ *and* $T_1^{(II)} = 0.8$)

b) Find a tight bound n' for the processing core count s.t. for $n \geq n'$ optimization (II) results in *lower execution times (≠ speedup)* than optimization (I). Provide your starting inequality!

c) *[2 bonus points]* Explain why optimization (I) results in a lower relative speedup than even the original (unoptimized) program. What is the intuition and potential pitfall behind relative speedup?

Ex 1.4 (2 points)

You have a choice between buying one uniprocessor that executes ten billion instructions per second, or a twenty-processor multiprocessor where each processor executes one billion instructions per second. Explain how you would decide which to buy for a particular application.

2 Locks

Ex 2.1 (2 points)

Why do we need to define a doorway section and why cannot we define FCFS in a mutual exclusion algorithm based on the order in which the first instruction in the *lock*() method was executed? Argue your answer in a case-by-case manner based on the nature of the first instruction executed by *lock*(): a read or write, to separate or the same location.

Ex 2.2 (4 points)

Programmers at the Flaky Computer Corporation designed the protocol shown in Figure 1 to achieve n-thread mutual exclusion. For each question either sketch a proof or display an execution, where it fails.

a) Does this protocol satisfy mutual exclusion?

b) Is this protocol starvation-free?

c) Is this protocol deadlock-free?

```
1  class Flaky implements Lock {
2    private int turn;
3    private boolean busy = false;
4    public void lock() {
5      int me = ThreadID.get();
6      do {
7        do {
8          turn = me;
9        } while (busy);
10       busy = true;
11     } while (turn != me);
12   }
13   public void unlock() {
14     busy = false;
15   }
16 }
```

Figure 1: Flaky Lock

Ex 2.3 (10 points)

Another way to generalize the two-thread Peterson lock seen in Figure 2 is to arrange a number of 2-thread Peterson locks in a binary tree. Suppose n is a power of two. Each thread is assigned a leaf lock which it shares with one other thread. Each lock treats one thread as thread 0 and the other as thread 1.

In the tree-lock's acquire method, the thread acquires every two-thread Peterson lock from that thread's leaf to the root. The tree-lock's release method for the tree-lock unlocks each of the 2-thread Peterson locks that thread has acquired, *starting from the leaf up to the root.* At any time, a thread can be delayed for a finite duration. (In other words, threads can take naps, or even vacations, but they do not drop dead.) For each of the first three listed properties below, either sketch a formal induction proof that it holds, or describe a (possibly infinite) execution, where it is violated. In case of a violation: can you find a (simple) fix to make it work? If so, provide the necessary changes and sketch an induction proof that your fix really works.

a) Mutual exclusion

b) Deadlock freedom

c) Starvation freedom

d) Is there an upper bound on the number of times the tree-lock can be acquired and released between the time a thread starts acquiring the tree-lock and when it succeeds? If so, sketch a proof, otherwise construct an unbounded execution.

```
1  class Peterson implements Lock {
2    // thread-local index, 0 or 1
3    private boolean[] flag = new boolean[2];
4    private int victim;
5    public void lock() {
6      int i = ThreadID.get();
7      int j = 1 - i;
8      flag[i] = true;            // I'm interested
9      victim = i;                // you go first
10     while (flag[j] && victim == i) {}; // wait
11   }
12   public void unlock() {
13     int i = ThreadID.get();
14     flag[i] = false;           // I'm not interested
15   }
16 }
```

Figure 2: Peterson Lock

Ex 2.4 (4 points)

The $\mathfrak{L}$-exclusion problem is a variant of the starvation-free mutual exclusion problem. We make two changes: as many as $\mathfrak{L}$ threads may be in the critical section at the same time, and fewer than $\mathfrak{L}$ threads might fail (by halting) in the critical section.

An implementation must satisfy the following conditions:

- **$\mathfrak{L}$-exclusion:** at any time at most $\mathfrak{L}$ threads are in the critical section.
- **$\mathfrak{L}$-starvation-freedom:** as long as fewer than $\mathfrak{L}$ threads are in the critical section, some thread that wants to enter the critical section will eventually succeed (even if some threads in the critical section have halted).

Modify the n-process Filter mutual exclusion algorithm from Figure 3 to turn it into an $\mathfrak{L}$-exclusion algorithm. Provide the whole source code!

```
1  class Filter implements Lock {
2    int[] level;
3    int[] victim;
4    public Filter(int n) {
5      level  = new int[n];
6      victim = new int[n]; // use 1..n-1
7      for (int i = 0; i < n; i++) {
8        level[i] = 0;
9      }
10   }
11   public void lock() {
12     int me = ThreadID.get();
13     for (int i = 1; i < n; i++) { // attempt level i
14       level[me] = i;
15       victim[i] = me;
16       // spin while conflicts exist
17       while ((∃k != me) (level[k] >= i && victim[i] == me)) {};
18     }
19   }
20   public void unlock() {
21     int me = ThreadID.get();
22     level[me] = 0;
23   }
24 }
```

Figure 3: Filter Lock

Ex 2.5 (2 points)

In practice, almost all lock acquisitions are uncontended, so the most practical measure of a lock's performance is the number of steps needed for a thread to acquire a lock when no other thread is concurrently trying to acquire the lock.

Scientists at Cantaloupe-Melon University have devised the following 'wrapper' for an arbitrary lock, shown in Figure 4. They claim that if the base *Lock* class provides mutual exclusion and is starvation-free, so does the *FastPath* lock, but it can be acquired in a constant number of steps in the absence of contention. Sketch an argument why they are right, or give a counterexample.

```
1  class FastPath implements Lock {
2    private static ThreadLocal<Integer> myIndex;
3    private Lock lock;
4    private int x, y = -1;
5    public void lock() {
6      int i = myIndex.get();
7      x = i;                    // I'm here
8      while (y != -1) {}        // is the lock free?
9      y = i;                    // me again?
10     if (x != i)               // Am I still here?
11       lock.lock();            // slow path
12   }
13   public void unlock() {
14     y = -1;
15     lock.unlock();
16   }
17 }
```

Figure 4: FastPath Lock

3 Register construction

Ex 3.1 (2 points)

Consider the safe Boolean MRSW construction shown in Figure 5.
True or false: if we replace the safe Boolean SRSW register array with an array of atomic SRSW registers, then the construction yields an atomic Boolean MRSW register. Justify your answer by sketching a proof or providing a counterexample.

```
1  public class SafeBooleanMRSWRegister implements Register<Boolean> {
2    boolean[] s_table; // array of safe SRSW registers
3    public SafeBooleanMRSWRegister(int capacity) {
4      s_table = new boolean[capacity];
5    }
6    public Boolean read() {
7      return s_table[ThreadID.get()];
8    }
9    public void write(Boolean x) {
10     for (int i = 0; i < s_table.length; i++)
11       s_table[i] = x;
12   }
13 }
```

Figure 5: Safe Boolean MRSW construction

Ex 3.2 (2 points)

Consider the atomic MRSW register construction shown in Figure 6.
True or false: if we replace the atomic SRSW registers with regular SRSW registers, then the construction still yields an atomic MRSW register. Justify your answer by sketching a proof or providing a counterexample.

```
1  public class AtomicMRSWRegister<T> implements Register<T> {
2    ThreadLocal<Long> lastStamp;
3    private StampedValue<T>[][] a_table; // each entry is SRSW atomic
4    public AtomicMRSWRegister(T init, int readers) {
5      lastStamp = new ThreadLocal<Long>() {
6         protected Long initialValue() { return 0; };
7      };
8      a_table = (StampedValue<T>[][]) new StampedValue[readers][readers];
9      StampedValue<T> value = new StampedValue<T>(init);
10     for (int i = 0; i < readers; i++) {
11       for (int j = 0; j < readers; j++) {
12         a_table[i][j] = value;
13       }
14     }
15   }
16   public T read() {
17     int me = ThreadID.get();
18     StampedValue<T> value = a_table[me][me];
19     for (int i = 0; i < a_table.length; i++) {
20       value = StampedValue.max(value, a_table[i][me]);
21     }
22     for (int i = 0; i < a_table.length; i++) {
23       if (i == me) continue;
24       a_table[me][i] = value;
25     }
26     return value;
27   }
28   public void write(T v) {
29     long stamp = lastStamp.get() + 1;
30     lastStamp.set(stamp);
31     StampedValue<T> value = new StampedValue<T>(stamp, v);
32     for (int i = 0; i < a_table.length; i++) {
33       a_table[i][i] = value;
34     }
35   }
36 }
```

Figure 6: Atomic MRSW construction

Ex 3.3 (4 points)

Does Peterson's two-thread mutual exclusion algorithm from Figure 2 still work if the shared atomic `flag` registers are replaced by safe registers? Argue by either providing a proof sketch or counterexample.