Vor und Nachname (Druckbuchstaben): _____

Legi Nummer: _____

Unterschrift: _____

252-0024-00L
Parallele Programmierung
ETH/CS: FS 2014
Basisprüfung
Samstag, 09.08.14
120 Minuten

This exam contains 21 pages (including this cover page) and 8 problems. Check to see if any pages are missing. Enter all requested information on the top of this page, and put your Legi number on the top of every page, in case the pages become separated.

The only written aids you are allowed are 4 sides (2 A4 pages) of hand-written notes. You may *not* use additional notes, your books, or any calculator on this exam.

Read ahead, take five minutes to read through the questions.

The following rules apply:

- **Organize your work**, in a reasonably neat and coherent way, in the space provided. Work scattered all over the page without a clear ordering will receive very little credit.
- **Mysterious or unsupported answers will not receive full credit on problems where we ask you to show your working**. A correct answer, unsupported by calculations, explanation, or algebraic work will receive no credit; an incorrect answer supported by substantially correct calculations and explanations might still receive partial credit.
- If you need more space, use the back of the pages or the blank pages; clearly indicate when you have done this. **As a guideline, you should be able to answer the questions within the provided space.**
- Provide your answers either in English or German. Do not use a red pen!

Problem	Points	Score
1	17	
2	8	
3	10	
4	8	
5	10	
6	14	
7	22	
8	17	
Total:	106	

1. **Short Questions**

   (a) (4 points) A single-threaded program you are developing is not fast enough. We assume that the program consists of a sequential part (that takes 60% of the execution time) and a parallel part (that takes the rest 40% of execution time). You have two options to make it faster:

   1. Optimize the sequential part so that its time is cut down to half
   2. parallelize the parallel part (assume that it scales perfectly)

   Both approaches require the same effort. Under what conditions would you choose the second option? Justify your answer.

   .....

   .....

   .....

   .....

   .....

   .....

   .....

   .....

   .....

   .....

   (b) (2 points) Consider you have to build an application that handles all incoming requests in a web server. For processing requests you have a choice between threads and tasks. Which one would you use and why?

   .....

   .....

   .....

   .....

   .....

   .....

(c) (2 points) Name two approaches to reduce lock contention and briefly explain the effect each of them have.

(d) (3 points) Explain in words what a barrier is and describe its interface. eschreiben Sie in Ihren eigenen Worten was eine *Barriere (eng.: barrier)* ist und wann Sie dieses Konstrukt benutzen würden. (Verwenden Sie ggf. ein Beispiel).

(e) (3 points) In Java there are two types of thread-safe collections: synchronized (Collections.synchronizedMap) and concurrent (e.g. ConcurrentHashMap). Explain the difference and give a benefit of concurrent over synchronized.

(f) (3 points) Briefly compare and contrast shared vs. isolated mutability.

.....

.....

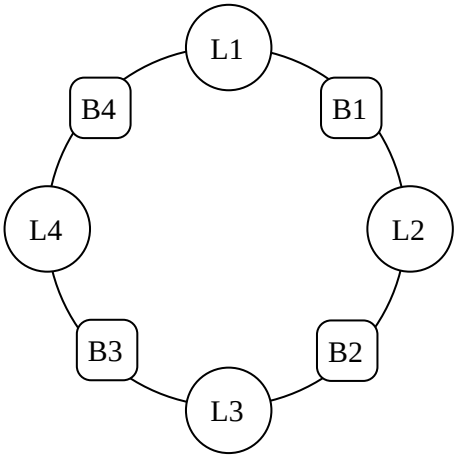
.....

.....

.....

2. (8 points) **Controlling Lights**

Assume you have 4 Lights (`L`) and 4 Buttons (`B`) arranged in a circle as in the figure below. The lights are represented by the `Light` class, whose fields are protected by the intrinsic lock (see for example `setColor()`).



Create a function `tryPressButton()` that presses the button only when **both** neighbor lights are GREEN. Note that there are hints in the comments of the code below. Other threads can operate on the `Light` objects (e.g., by calling `setColor()`). Explain your synchronization scheme using a few sentences.

```
enum Color { // Different colors
    GREEN,
    RED,
    ...
}

public class Light {
    // for simplicity, fields are public
    public int id;      // light number (1, 2, 3, 4);
    public Color color; // current color;

    // class is protected by the intrinsic lock
    public void synchronized setColor(Color c) {this.color  = c;}

    ...
}

// You can use the following functions:
// get the i light. Assume this function is thread-safe
public static Light getLight(int i)  { ... }
// get the i button. Assume this function is thread-safe
public static Button getButton(int i) { ... }

// You need to complete this function:
public void tryPressButton(i) {

    // Get the ith Button
    // You can use b.press() to press it
    Button b = getButton(i);

    // get the ith Light
    Light li = getLight(i);
    ..........................................................
    ..........................................................
    ..........................................................
    ..........................................................
    ..........................................................
    ..........................................................
    ..........................................................
    ..........................................................
    ..........................................................
    ..........................................................
    ..........................................................
    ..........................................................
    ..........................................................
    ..........................................................
    ..........................................................
    ..........................................................
```

```
        . . . . . .
        . . . . . .
        . . . . . .
        . . . . . .
        . . . . . .
        . . . . . .
        . . . . . .
        . . . . . .
        . . . . . .
        . . . . . .
        . . . . . .
        . . . . . .
        . . . . . .
        . . . . . .
        . . . . . .
}
```

Explanation of synchronization scheme:

. . . . . .

. . . . . .

. . . . . .

3. **Iterative algorithm**

(a) (7 points) Assume you have an initial array A_0 of size N, and an algorithm that takes R repetitions, and in each produces a new version of the array based on the old version. Each element in the new array depends **only** in the old array and the location of the element (`fn` function in the pseudocode example below).

Pseudocode:

```
for (r=0; r<R; r++) {   // r: repetitions
  for (i=0; i<N; i++) { // i: array element
    A_{r+1}[i] = fn(A_r,i) // compute ith element
  }
}
```

This algorithm can be parallelized as follows:

```
for (r=0; r<R; r++) {
  create T threads
  give each thread a part of the output array to compute
  wait untill all threads are finished
}
```

Where each thread works as follows:

```
// perform the computation for the given part of the array
for (/* thread's part of the array */) {
  A_{r+1}[i] = fn(A_r,i)
}
```

Spawning the threads at each step, however, has overhead. Describe the modifications needed to provide a **correct** version of the above code, where the threads are spawned only in the beginning. You can use the skeleton below.

Algorithm:

```
allocate all arrays
.....................................................
.....................................................
create T threads
.....................................................
.....................................................
give each thread a part of the output array to compute
.....................................................
.....................................................
wait until all threads finish
.....................................................
.....................................................
```

Each thread:

. . . . . .
. . . . . .
. . . . . .
. . . . . .
. . . . . .
. . . . . .

(b) (3 points) Assume that f(A_r, x) takes time k for $x \in 1, 2, \ldots, n$ etc. And $f(A_r, x = 0)$ takes time $20 \times k$. Would that cause a problem? Justify your answer and briefly describe a solution to the problem if one exists.

. . . . . .

. . . . . .

. . . . . .

. . . . . .

. . . . . .

. . . . . .

. . . . . .

. . . . . .

. . . . . .

. . . . . .

4. **Data and pipeline parallelism** Assume you have functions `f()`, `g()`, `h()` that take a single `Integer` argument, and return an `Integer`. Also assume that they **only** access their argument and no other data when executed.

   For example:

```
public Integer f(Integer x) {
        // only access x, and no other data
}
```

   You want to compute `f(g(h(x)))` for each element `x` on a large array.

   (a) (2 points) Describe in a few sentences how can you build a parallel version of this computation using the data parallel model.

   (b) (2 points) Describe in a few sentences how can you build a parallel version of this computation using a pipeline.

(c) (2 points) What is the maximum speedup you can get when using the pipeline version and why? (only consider the stages where the pipeline is full)

(d) (2 points) Can you combine the pipeline and data parallel versions? Give an example.

5. (10 points) **Matching Parentheses**

In this task we are concerned with the problem of matching parentheses. We would like to count the number of *unmatched* open and closing parentheses in a string. For convenience, the string is represented by an array where 0 is an open parenthesis and 1 is a closing parenthesis. Some examples and the expected output are shown below.

String	**Array**	**Result**
(	[0]	open = 1, closed = 0
)(	[1,0]	open = 1, closed = 1
()(	[0,1,0]	open = 1, closed = 0
()()	[0,1,0,1]	open = 0, closed = 0
((())	[0,0,0,0,1,1]	open = 2, closed = 0
))()()(	[1,1,0,1,0,1,0]	open = 1, closed = 2

Sketch how you would implement a method `matchParenParallel(int[] array)` that takes the entire array and computes its result *in parallel*. You can assume there is already a method for matching parentheses sequentially (as shown below) which you are free to use as a subroutine if you wish.

```
public static Pair<Integer, Integer> matchParentheses(
    int[] array, int start, int end) {
  int open = 0, closed = 0;
  for (int i = start ; i < end; i++) {
    // (Details of how this is implemented don't matter.)
  }
  return Pair.of(open, closed);
}
```

You can provide pseudocode, text, and/or drawings in your explanation. If you use any frameworks (e.g. raw threads, Java standard library, MapReduce) make sure it is clear from your explanation how these would be used.

6. **Data parallelism** Java 8 introduced a new framework to transform a collection of elements. As a reminder of some of what the API offers:

```
interface Stream<T> {
  /* Returns the count of elements in this stream. */
  long count();

  /* Returns only elements that match the given predicate. */
  Stream<T> filter(Predicate<? super T> predicate);

  /* Transforms a stream by applying the given function to every
     element. */
  <R> Stream<R> map(Function<? super T,? extends R> mapper);
  ...
}
```

(a) (2 points) Explain one pro and one con of the design of this API. (As a comparison point you can consider normal for-loops and threads.)

.....

.....

.....

.....

.....

.....

(b) (4 points) A bank has faulty ATM machines which sometimes writes a transaction multiple times to the log and so they wrote a routine to drop an element when it is equal to its neighbour. For this they used the new Streams API and their solution is fine when run sequentially, but doesn't work when parallelized. Explain the problem in 2-3 sentences.

```
List<String> accounts = Arrays.asList(new String[]{ "C3",
   "B2", "B2", "A1", "C3", "D4", "D4", "D4" });
// Expected output: ["C3", "B2", "A1", "C3", "D4"]

String last = null;
List<String> result =
  accounts.parallelStream()
          .filter(x -> {
              if (x.equals(last)) {
                return false; // drop
              } else {
                last = x;
                return true; // keep
              }
          })
          .collect(); // create a collection from the string
```

.....

.....

.....

.....

.....

(c) (8 points) Lets simplify the previous problem; you just need to detect that there are duplicates, without removing them. Using the fork-join framework write a function which does the following: given a list/array, return true if any pairs of elements are equal, otherwise return false. (Pseudo-code is fine and you can use the `spawn` and `sync` keywords as in the lecture.)

7. **Implementing Locks**

(a) (4 points) When waiting in a synchronization primitive, there is the possibility to spin or suspend execution. Explain these two options and the trade-off on performance.

(b) (8 points) You are building a new machine which only provides hardware support for a single thread synchronization primitive: *compare-and-swap* (CAS). As a reminder, this is an atomic operation and its signature and semantics are described below.

[NOTE: Java use different naming ... add a note that compare-and-swap and compare-and-set are one and the same]

```
class AtomicBoolean {
  /**
   * Atomically sets the value to the given updated value
   * only if both the current value and the expected values
   * are equal.
   *
   * @param expect the expected value
   * @param update the new value
   * @return True if successful; False return indicates that
   * the actual value was not equal to the expected value.
   */
  boolean compareAndSet(boolean expect, boolean update);

  /* Returns the current value. */
  public final boolean get();

  /* Unconditionally sets to the given value. */
  public final void set(boolean newValue);

  ...
}
```

Your task is to implement the code for a simple spinlock using only this atomic operation and nothing else. The lock you implement does *not* need to be reentrant and you can assume the client code uses the lock correctly (e.g. no double release or release by a thread not holding the lock).

```
class SpinLock {
  // Define any new members you need here.
  private final AtomicBoolean locked  = new AtomicBoolean();
  ..................................................
  ..................................................

  public void acquire() {
    ..............................................................
    ..............................................................
    ..............................................................
    ..............................................................
  }

  public void release() {
    ..............................................................
    ..............................................................
```

```
      .....................................................
      .....................................................
  }
}
```

(c) (10 points) After benchmarking your implementation, you realize that spinning is inefficient when you have to wait for a long time. The hardware designers tell you they can add two more primitives to make the lock more efficient:

- `atomicSleep` is an atomic operation which corresponds to the following pseudo-code:

```
// Note: SuspendLock is discussed below
void atomicSleep(SuspendLock l, AtomicBoolean val,
   boolean x) {
  atomic { // atomically:
    val.set(x); // set value to x
    suspend(l); // stop (suspend) the thread.
                // the thread is now sleeping on lock l.
  }
}
```

- `wakeUp` is non-atomic with the following pseudo-code:

```
void wakeUp(SuspendLock l) {
  wakeup_thread(l); // wake up a thread sleeping on lock l
}
```

Your second task is to reimplement the lock more efficiently using these operations. You may recognize this matches the basic design of a mutex. (The same assumptions hold as in the previous part.)

```
class SuspendLock {
  // Define any new members you need here.
  ..........................................................
  ..........................................................
  ..........................................................
  ..........................................................
  ..........................................................
  ..........................................................

  public void acquire() {
    ..........................................................
    ..........................................................
    ..........................................................
    ..........................................................
    ..........................................................
    ..........................................................
    ..........................................................
    ..........................................................
    ..........................................................
    ..........................................................
  }

  public void release() {
    ..........................................................
    ..........................................................
    ..........................................................
    ..........................................................
    ..........................................................
    ..........................................................
  }
}
```

8. **OpenCL**

(a) (2 points) Recall Flynn's taxonomy; under what class would you place (i) sequential CPUs and (ii) GPUs?

.....

.....

.....

(b) (3 points) Compare CPUs and GPUs in terms of throughput and latency. Justify these performance characteristics in terms of architectural differences.

.....

.....

.....

.....

(c) (5 points) GPUs have several different types of memory as shown in the figure below. Explain what each of these are by discussing access rights, where these are physically located and the impact on access latency.

Note: In your answer you should mention: private, local, global, constant and host memory.

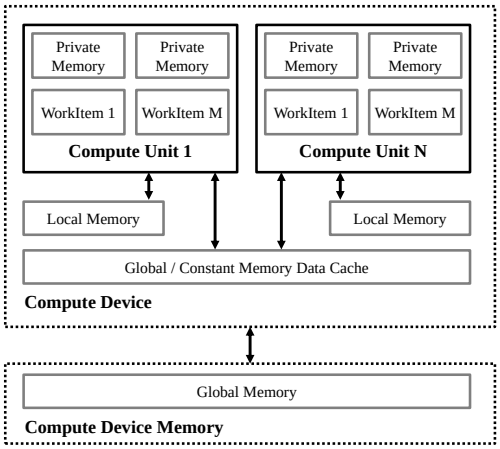


Figure 1: GPU Memory Hierarchy

(d) (3 points) Assume you are writing an algorithm that requires a tree reduction. The input array is too large to fit within a single work-group and the final result needs to written to global memory. How would you make sure the tree reduction is thread safe. Hint: you can assume the reduction operation is associative and commutative.

(e) (4 points) Below is a simple OpenCL kernel that uses a 2D grid of work-groups. It reads from *global* memory and fills a chunk of local memory. Finally, the work-item computes the final result by adding the values read from two *local* memory addresses.

Give an explanation for why the memory fence is needed in this code.

```
__kernel void fill_tiles(__global float* a,
                         __global float* b,
                         __global float* c)
{
  // find our coordinates in the grid
  int row = get_global_id(1);
  int col = get_global_id(0);

  // allocate local memory shared among the workgroup
  __local float aTile[TILE_DIM_Y][TILE_DIM_X];
  __local float bTile[TILE_DIM_Y][TILE_DIM_X];

  // define the coordinates of this workitem thread
  // in the 2D tile
  int y = get_local_id(1);
  int x = get_local_id(0);

  aTile[y][x] = a[row*N + col];
  bTile[y][x] = b[row*N + col];
  barrier(CLK_LOCAL_MEM_FENCE);

  // Note the change in tile location in bTile!
  c[row*N + col] = aTile[x][y] + bTile[y][x];
}
```

.....

.....

.....

.....

.....