Vor und Nachname (Druckbuchstaben): _____

Legi Nummer: _____

Unterschrift: _____

252-0024-00L
Parallele Programmierung
ETH/CS: HS 2014
Basisprüfung
Freitag, 23.01.15
120 Minuten

This exam contains 22 pages (including this cover page) and 6 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. Do not forget to sign the exam!

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	16	
2	16	
3	14	
4	4	
5	23	
6	27	
Total:	100	

1. **Threads**

   (a) (2 points) A program spends 2/10 of its execution time inside critical sections. What is the maximum speedup this program can achieve using 4 processors? Assume that super-linear (i.e., better than linear) speedup is not possible. Show your work.

   Solution:
   Time spend inside critical sections cannot be parallelized. We assume linear speedup for everything else.
   If sequential execution time is t, then parallel execution time using 4 processors is: $t_4 = t_{par}/4 + t_{seq} = .8t/4 + .2t = .4t$. Hence, the parallel speedup is $1/.4 = 2.5$.
   If rationale is correct, award 1 point.

   (b) (2 points) You may assume that the actual running time of a parallel program on a dedicated P-processor machine is $T_p = T_1/P + T_\infty$. You produce two chess programs, a simple one and a optimized one. The simple one has parameters $T_1 = 2048$ seconds and $T_\infty = 1$ second. When you run it on your 32 processor machine, the running time is 65s. You then produce an "optimized" version with $T_1' = 1024$ and $T_\infty = 8s$. Why is it optimized? When you run it on your 32-processor machine, the running time is 40s, as predicted by our formula. Which program will scale better to a 512-processor machine?"

   Solution: The program reduces parallel runtime by increasing sequential runtime. This achieves better performance for $P = 32$ but the sequential runtime becomes a bottleneck with higher processor counts.

(c) (2 points) Give one reason why it is hard to test multithreaded programs.

Solution: The main problem is nondeterminism from scheduling. For example, one run of the program might reveal a data race, but subsequent runs do not, making the bug hard to track down.

(d) (2 points) You wrote an embarrassingly parallel program that uses multiple threads. Predict the program's speedup when using 2, 8, and 16 threads when you run it on a machine with 8 cores. Justify your answer.

Solution: 2: 2, 8: 8, 16: 8.

(e) (2 points) Consider the following Java code snippet spawning two threads:

```
public class Main {
    public static void main(String[] args) {

        new Thread("t0") {
            public void run() {
                System.out.print(getName() + " before; ");
                new Thread("t2") {
                    public void run() {
                        System.out.print(getName() + "; ");
                    }
                }.start();
                System.out.print(getName() + " after; ");
            }
        }.start();

        new Thread("t1") {
            public void run() {
                System.out.print(getName() + "; ");
            }
        }.start();

        return;
    }
}
```

What are the possible outputs printed on the console when you run this main method? Assume no console buffering.

- √ `t0 before; t1; t0 after; t2;`
- √ `t1; t0 before; t0 after; t2;`
- □ `t1; t2; t0 after; t0 before;`
- √ `t0 before; t0 after; t2; t1;`

(f) (6 points) Explain what a reentrant lock is. Is the intrinsic lock used for Java's `synchronized` construct reentrant or non-reentrant? Give a code example that would not work correctly if the intrinsic lock was the opposite of your answer. Explain what the problem would be in this case.

Solution:

- A reentrant lock allows a thread to acquire a lock it already owns. (2pt)
- The intrinsic lock is reentrant (1pt)
- The following code would deadlock (3pt):

```
class Foo {
    ...
```

```
        public synchronized void f() {
            ...
        }

        public synchronized void g() {
            ...
            f();
            ...
        }
    }
```

2. **Deadlock**

Read the code below. The code creates a large number of tasks which execute in parallel at some time in the future.

```
// A "box" that can hold arbitrary Java objects
class Box {
  private Object value;
  public Box(Object x) { value = x; }
  public synchronized Object get() { return value; }
  public synchronized Object set(Object x) {
    Object y = value; value = x; return y;
  }
}

public static void main(String args[]) {
  Box b1 = new Box("hi"), b2 = new Box("bye");
  List<Runnable> tasks = new ArrayList<Runnable>();
  for (int i = 0; i < 1000; i++) {
    // 'createTask' returns a Runnable; the task body is
    // defined separately for each part of the question.
    if (i % 2 == 0) {
      tasks.add(createTask(b1, b2));
    } else {
      tasks.add(createTask(b2, b1));
    }
  }

  // Executes the tasks asynchronously and this call blocks until
  // all tasks have completed.
  ExecutorService exec = Executors.newCachedThreadPool();
  exec.invokeAll(tasks);
}
```

In this question you need answer whether the following code can deadlock for each different implementation of createTask(). Justify your answers (and use an example if applicable).

(a) (3 points)

```
// create a runnable to be executed as a task.
public Runnable createTask(final Box fb1, final Box fb2) {
  return new Runnable() {
    public void run() {
      // code that will be executed when the task starts:
      Object x1 = fb1.get();
      Object x2 = fb2.set(x1);
      fb1.set(x2);
    } };
}
```

Solution: No deadlock. Only one lock is ever held at once.

(b) (3 points)
```
// create a runnable to be executed as a task.
public Runnable createTask(final Box fb1, final Box fb2) {
  return new Runnable() {
    public void run() {
      // code that will be executed when the task starts:
      synchronized (fb1) {
        synchronized (fb2) {
          Object x1 = fb1.get();
          Object x2 = fb2.set(x1);
          fb1.set(x2);
        } } } };
}
```

Solution: Can deadlock. Sometimes `fb1` will be the first Box, and sometimes the second, so its possible for tasks to be waiting on the others' box locks, in a cycle.

(c) (10 points) Unlike Java, many languages do not include an intrinsic lock for their objects. In such a language we might want to save space by creating a smaller number of locks for a given number of objects.

Let's assume a lock implementation using Semaphores:

```
// a lock implementation using semaphores
class SemaLock {
  private final Semaphore sema;
  private final long semaId;

  SemaLock(long sid) {
    sema = new Semaphore(1);
    semaId = sid;
  }

  long getId()  { return this.semaId; }
  // Fortsetzung auf der naechsten Seite!

  // normally sem.acquire() throws an Exception, but we
  // (and you!) can ignore exceptions for this question.
  void lock()   { sema.acquire(); }
  void unlock() { sema.release(); }
}
```

And a variant of a box class, called `Box2`, that uses these locks:

```
class Box2 {
  private Object value;
  private final SemaLock lock;
```

```
public Box2(Object x, SemaLock l) {
    value = x;
    lock = l;
}

public Object get() {
    Object v;
    lock.lock();
    v = value;
    lock.unlock();
    return v;
}

public Object set(Object x) {
    lock.lock();
    Object y = value; value = x;
    lock.unlock();
    return y;
}
```

We create 10 locks and 1000 boxes as follows:

```
public static void main(String args[]) {
    final int nLocks   = 10;
    final int nObjects = 1000;
    // initialize locks
    List<SemaLock> locks = new ArrayList<SemaLock>();
    for (int i=0; i<nLocks; i++) {
        locks.add(new SemaLock(i));
    }
    // create boxes
    List<Box2> boxes = new ArrayList<Box2>();
    for (int i=0; i<nObjects; i++) {
        SemaLock boxLock = locks.get(i % nLocks);
        boxes.add(new Box2(new Integer(i), boxLock));
    }
    // Fortsetzung auf der naechsten Seite!
    // Now, use box objects from multiple threads
    ...
}
```

Is the synchronization practice described above safe compared to using one lock per object? Is there a tradeoff regarding the number of lock objects we use? Justify your answer.

Solution: It is a safe practice because mutual exclusion is guaranteed (2pt). The tradeoff is scalability (more locks → more parallelism) against space for the lock objects (2pt).

Finally, add a `swap(Box2 b)` method in the `Box2` class that *atomically* exchanges the

objects of the two boxes (the instance and the argument) and is thread-safe.

```
... // inside Box2 class definition
 private unsafeSwap(Box2 b) { // unsafe!
     Object o = this.value;
     this.value = b.value;
     b.value = o;
 }
 void swap(Box2 b) {
 ..........................................................................

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

 }
```

Solution:

```
private unsafeSwap(Box2 b) {
```

```
    Object o = this.value;
    this.value = b.value;
    b.value = o;
}
void swap(Box2 b) {
    if (this.lock.getId() < b.lock.getId()) {
        this.lock.lock();
        b.lock.lock();
    } else if (this.lock.getId() > b.lock.getId()) {
        b.lock.lock();
        this.lock.lock();
    } else {
        // semaphores are not reintrant!
        this.lock.lock();
    }

    unsafeSwap();

    ... unlock in the same order;
}

4pt if did not consider the == case
```

3. **Barriers**

(a) (4 points) Explain in words what a barrier is and in which situations you could use it (give an example for this).

> **Solution:** Barrier: rendezvous for arbitrary number of threads. Alternative definition: A barrier is a way of forcing asynchronous threads to act almost as if they were synchronous. Examples: game of life, displaying the pixels in an image.
> 2p for a correct definition. 2p for a good example.

(b) (5 points) The following code uses N threads to compute the entries in array A of size N. Each thread is responsible for updating a single entry in the array. Thread 0 is supposed to print the contents of the array after every iteration, to show the new values. Extend the code below by using a barrier to make sure the values printed are all from the same iteration.
You can use the `CyclicBarrier` and it's `await` method from `java.util.concurrent`, ignore any exception handling code for this exercise).

```
public class Main {

    public static int computeNextElement() {
        // computes the next element and returns it
    }

    public static void main(String[] args) {
        int N = 10;
        int[] A = new int[N];

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

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

        for (int i=0; i<N; i++) {
            final int threadId = i;

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

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

            new Thread(new Runnable() {

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

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

                @Override
                public void run() {

                    ..........................................
```

```
                    .....................................

                    for (int j = 0; j < 5; j++) {
                        A[threadId] = computeNextElement();

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

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

                        if (threadId == 0) {
                            for (int k = 0; k < N; k++) {
                                System.out.print(A[k] + " ");
                            }
                            System.out.println();
                        }

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

                        .....................................
                    }

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

                    .....................................
                }
            }).start();
        }
    }
}
```

Solution:

```
    final CyclicBarrier barrier = new CyclicBarrier(N);

    for (int i=0; i<N; i++) {
        final int threadId = i;

        new Thread(new Runnable() {

            @Override
            public void run() {

                for (int j = 0; j < 5; j++) {
                    A[threadId] = computeNextElement();

                    try {
```

```
                    barrier.await();
                } catch (InterruptedException |
                   BrokenBarrierException e) {
                    e.printStackTrace();
                }

                if (threadId == 0) {
                    for (int k = 0; k < N; k++) {
                        System.out.print(A[k] + " ");
                    }
                    System.out.println();
                }

                try {
                    barrier.await();
                } catch (InterruptedException |
                   BrokenBarrierException e) {
                    e.printStackTrace();
                }
            }
        }
    }).start();
}
```

(c) (5 points) The FaultyBarrier class is problematic if it is used instead of CyclicBarrier in the previous exercise. Describe the problem and give a scenario that triggers the problem.

```
class FaultyBarrier {
  final int size;
  AtomicInteger count;

  public FaultyBarrier(int n) {
    this.size = n;
    this.count = new AtomicInteger(0);
  }

  public void await() {
    int position = count.incrementAndGet();
    if (position == this.size) {
      count.set(0);
    } else {
      while (count.get() != 0) { /* spin-wait */ }
    }
  }
}
```

Solution: Problem: Not reusable: deadlock / race during re-initialization. (2pt)
Two threads: T1 sees it isn't last and spins; T2 is last so it resets the counter and starts a new round. T1 still spinning on previous round nor does T2 progress.) (3pt)
Solution: use two barriers and alternate or have odd-even phases. (Also called a sense-reversing barrier).

4. **Count threes**

Assume an array of `N` integers, and the following code that uses an accumulator-based algorithm to count the number of `3` in this array:

```
public int countThrees(int numbers[]) {
    int count = 0;
    for (int i=0; i<numbers.length; i++) {
        if (numbers[i] == 3)
            count++;
    }
    return count;
}
```

(a) (2 points) Using this example, describe why accumulator-based algorithms cannot be efficiently parallelized.

> **Solution:** shared state, synchronization will add to much overhead.
> Data dependency between each iteration of the loop (on 'count' variable) which prevents pipelining or parallelisation.

(b) (2 points) Propose an alternative algorithm for implementing the same functionality that can be parallelized. Justify your answer.

> **Solution:** divide-and-conquer, data parallel.
> Reduction tree =¿ count is like a sum over ints from 0, 1. With Java 8 Streams: numbers.parallelStream().filter(x =¿ (x == 3)).sum() (Note: this assumes that numbers is a Collection not array.)

5. **Concurrent Message Passing**

(a) (5 points) We consider a one-dimensional finite difference problem, in which we have a vector X of size N and must compute X^T, where

$$0 < i < N-1, 0 \leq t < T : X_i^{t+1} = X_{i-1}^t + 2 * X_i^t + X_{i+1}^t$$

Briefly explain the design for a parallel algorithm that uses an actor framework to compute the finite differences of X. You should use one actor per vector element. Your explanation must also describe the messages you exchange between actors (i.e., how would you write a onReceive method for this using the Akka framework).

Solution: Every actor is connected to its neighbors. At step t, each task i other than task 0 and task N-1:

- Sends its data on its left and right channels, (1pt)
- Receives and from its left and right channels, and (1pt)
- Uses these values to compute X^{t+1}. (1pt)

(b) (10 points) Now we assume a similar problem where we want to compute the pairwise interaction Y_i for each element X_i in vector X. The interaction is defined by function F as follows:

$$Y_i = \sum_{j=0}^{N-1} F(X_i, X_j), i \neq j$$

In the code below, we try to model each vector element X_i as an actor that computes Y_i and sends its value X_i to all other actors as a message. Complete the actor class as described to compute Y.

```
class VectorElementActor extends UntypedActor {

    // You can use the following values and functions in
    // your code assume they are correctly initialized
    // and defined.
    // Size of vector X
    private static int N;
    // Our value in the vector (X_i)
    private int value;
    // Our index in vector (i)
    private int index;
    // Storage of Y_i
    private int y;
    // Returns ActorRef for corresponding vector element X_j
    private ActorRef getActor(int j);
    // Function F to compute pairwise interaction
    private int F(int xi, int xj);

    VectorElementActor() {}
```

```
    @Override
    public void preStart() {

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

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

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

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

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

        ..................................................
    }

    @Override
    public void onReceive(Object msg) {

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

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

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

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

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

Solution:

```
@Override
public void preStart() {
    for (int j=0; j<N; j++) { (2pt)
        if (j != index) { (2pt)
            getActor(j).tell(new Integer(value),
               getSelf()); (2pt)
        }
    }
}

@Override
public void onReceive(Object msg) {
    if (msg == (Integer)msg) { (2pt)
```

```
            y += F(value, (Integer)msg); (2pt)
        }
        else unhandled(msg);
    }
```

(c) (2 points) How many messages do you need to send in your implementation? How many message channels does your actor system need to set-up (a channel is a bi-directional communication line between two actors)?

Solution: Messages: $N * (N - 1)$ (1pt)
Channels: $N * (N - 1)/2$ (1pt)

(d) (2 points) If you can assume F is symmetric, can you change your implementation to use less messages? How many?

Solution: Only half of the actors need to send a message to the other half $F(X_i, X_j) = F(X_j, X_i)$, amount of channels stays the same (1pt)
Messages: $N * (N - 1)/2$ (1pt)

(e) (4 points) Now assume you want to reduce the amount of channels and only use N channels. Describe how you would you change your algorithm. How many messages do you need to exchange with your new algorithm? Note that the channel endpoints are fixed to only two actors (you can not change the endpoint of a channel to send to multiple actors using the same channel). You can no longer assume that F is symmetric for this question.

Solution: Build a ring of actors, each actor is connected to successor and predecessor. Every actor forwards messages through the ring. (3pt) Messages: $N*(N-1)$ (1pt)

6. **OpenCL: Sobel Filter** The Sobel operator is used widely in image processing. It performs a 2-D spatial gradient measurement on an image and so emphasizes regions of high spatial frequency. This regions typically correspond to edges. With an input (Figure 1a) it produces an output image that only highlights the edges of the input (Figure 1b).

(a) Original image.

(b) Output image of the sobel algorithm.

Figure 1: Input and Output for the Sobel Algorithm.

The Sobel operator consists of a pair of 3×3 convolution kernels, G_x and G_y, given below. G_y is simply G_x rotated by 90°. These kernels are designed to respond maximally to edges running vertically and horizontally relative to the pixel grid, one kernel for each of the two perpendicular orientations. The kernels compute, for the center pixel of an image region X, how big the change in brightness is in that region (and hence if the center pixel lies on an edge). The pixel value in the output image (gradient magnitude) Y is given by:

$$Y = \frac{\sqrt{\left(\sum_{i=0}^{2}\sum_{j=0}^{2} X_{(i,j)} \cdot G_{x(i,j)}\right)^2 + \left(\sum_{i=0}^{2}\sum_{j=0}^{2} X_{(i,j)} \cdot G_{y(i,j)}\right)^2}}{2}$$

Figure 2 shows how Sobel is applied on a single 3×3 block inside the image. The new pixel value for the center of the 3×3 block corresponds to the calculation for Y. Also note that we can ignore pixels that lie on the border of the image as computing the values for the border would require us to have the pixel values beyond the borders of the image.

The filters G_x and G_y for Sobel are defined as follows:

$$G_x = \begin{bmatrix} -1 & 0 & 1 \\ -2 & 0 & 2 \\ -1 & 0 & 1 \end{bmatrix}, G_y = \begin{bmatrix} -1 & -2 & -1 \\ 0 & 0 & 0 \\ 1 & 2 & 1 \end{bmatrix}$$

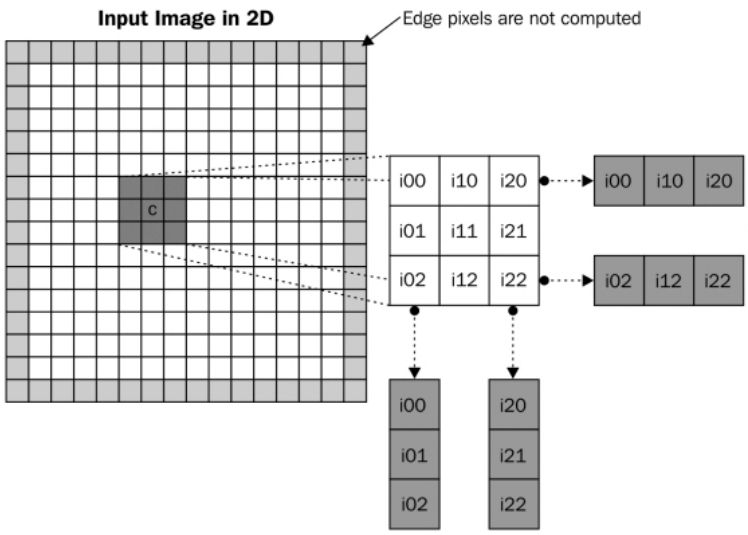

Figure 2: Application of the Sobel filter for a single pixel (C) inside an image. The columns i00 – i02 and i20 – i22 are multiplied with G_x and the rows i00 – i20 and i02 – i22 are multiplied with G_y to compute a new value Y for the center pixel C.

(a) (3 points) Sobels edge detection mechanism can be implemented very efficiently on GPUs. Explain why? Hint: think about the steps involved in the computation of Y

Solution:

- Can compute each pixel in parallel, no dependencies, no synchronization
- Simple operations, can be execute by GPU
- Need only access to neighboring pixels

(b) (4 points) Given image region X as a 3×3 matrix below, apply the Sobel filter on X and calculate Y.

$$X = \begin{bmatrix} 1 & 2 & 1 \\ 3 & 1 & 3 \\ 2 & 2 & 2 \end{bmatrix}$$

Solution:

$$X * G_x = 0$$

(1.5pt)

$$X * G_y = 2$$

(1.5pt)

$$Y = \frac{\sqrt{0 + 2^2}}{2} = 1$$

(1pt)

(c) (20 points) Implement a OpenCL kernel for a Sobels algorithm. The OpenCL kernel receives as input a black-white image represented by an array of integer values between 0 and 255, with 0 being completely black and 255 representing white. Our OpenCL program is set-up to call the kernel for every pixel (x, y) in the input image. The resulting output image should be stored in the output Array. You can use the hypot(x,y) OpenCL function to compute $\sqrt{x^2 + y^2}$.

```
__kernel void SobelDetector(__global uchar4[][] input,
                            __global uchar4[][] output) {
  uint x = get_global_id(0); // Pixel coordinate X
  uint y = get_global_id(1); // Pixel coordinate Y

  uint width = get_global_size(0); // Total width of image
  uint height = get_global_size(1); // Total height of image
```

Solution:

```
__kernel void SobelDetector(__global uchar4[][] input,
                            __global uchar4[][] output) {
  uint x = get_global_id(0);
  uint y = get_global_id(1);

  uint width = get_global_size(0);
  uint height = get_global_size(1);

  if( x >= 1 && x < (width-1) && y >= 1 && y < height - 1)
     (3pt)
  {
    float4 i00 = convert_float4(input[(x - 1)][(y - 1)]);
    float4 i10 = convert_float4(input[x][y - 1]);
    float4 i20 = convert_float4(input[x + 1][y - 1]);
    float4 i01 = convert_float4(input[(x - 1][y]);
    float4 i11 = convert_float4(input[x][y]);
    float4 i21 = convert_float4(input[x + 1][y]);
    float4 i02 = convert_float4(input[x - 1][y + 1]);
    float4 i12 = convert_float4(input[x][y + 1]);
    float4 i22 = convert_float4(input[x + 1][y + 1]);
    (9pt)

    // To understand why the masks are applied this way,
       look
    // at the mask for Gy and Gx which are respectively
       equal
    // to the matrices:
    // { {-1, 0, 1}, { {-1,-2,-1},
    //   {-2, 0, 2},   { 0, 0, 0},
    //   {-1, 0, 1}}   { 1, 2, 1}}
    float4 Gx = -i00 - 2*i10 - i20 + i02 - 2*i12 + i22;
```

```
    float4 Gy = -i00 + i20 - 2*i01 + 2*i21 - i02 + i22;
    (5pt)

    // The math operation here is applied to each element
       of
    // the unsigned char vector and the final result is
       applied
    // back to the output image
    output[x][y] = convert_uchar4(hypot(Gx,
       Gy)/(float4)(2));
    (3pt)
  }
}
```

- Either 2D array or pointer access in code is fine
- Not necessary to do proper casting to float and convert back for points