Vor und Nachname (Druckbuchstaben): _____

Legi Nummer: _____

Unterschrift: _____

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

Diese Prüfung enthält 20 Seiten (inklusive diesem Deckblatt) und 6 Aufgaben. Überprüfen Sie dass keine Seiten fehlen. Füllen Sie alle oben verlangten Informationen aus. Schreiben Sie die Legi Nummer oben auf jede einzelne Seite, für den Fall das Seiten verlorengehen oder abgetrennt werden. Vergessen Sie nicht, die Prüfung zu unterschreiben!

Als Hilfsmittel sind nur 4 Seiten (2 A4-Blätter) handgeschriebene Notizen erlaubt. Es sind keine zusätzliche Hilfsmittel (Unterlagen, Bücher, Notizen, Taschenrechner) für diese Prüfung zugelassen.

Nehmen Sie sich am Anfang 5 Minuten Zeit, um alle Aufgaben durchzulesen. Während dieser Zeit ist es nicht erlaubt, Prüfungsfragen zu beantworten.

Es gelten die folgenden Regeln:

- **Lösungen müssen lesbar sein**. Verwenden Sie für Ihre Lösungen den verfügbaren Platz. Lösungen mit unklarer Reihenfolge oder anderweitig unverständlicher Präsentation können zu Punktabzügen führen.
- **Lösungen ohne Lösungsweg erhalten nicht die volle Punktzahl**. Eine korrekte Antwort, ohne Lösungsweg, Erklärungen, oder Algebraische Umformungen erhält keine Punkte; inkorrekte Antworten, mit teilweise richtigen Formeln, Berechnungen und Umformungen können Teilpunkte erhalten.
- Falls mehr Platz benötigt wird, schreiben Sie auf die leeren Seiten der Prüfung oder fordern Sie bei den Assistenten extra Blätter an. Versehen Sie die Aufgabe mit einem klaren Hinweis, falls das der Fall ist. Als Richtlinie: **Die Aufgaben sollten sich alle in dem vorgegebenen Platz beantworten lassen.**
- Die Aufgaben können auf **Englisch oder Deutsch** beantwortet werden. Benutzen Sie keinen roten Stift!

Problem	Points	Score
1	16	
2	16	
3	14	
4	4	
5	23	
6	27	
Total:	100	

1. **Threads**

   (a) (2 points) Ein Programm verbringt 2/10 seiner Laufzeit in einer Critical Section. Was ist der maximale Speedup der erreicht werden kann mit 4 Prozessoren? Nehmen Sie an, dass super-linear (besser als linearer) Speedup nicht möglich ist. Zeigen Sie den Lösungsweg auf.

   .....

   .....

   .....

   .....

   .....

   .....

   (b) (2 points) Gegeben sei die Laufzeit eines Programm auf einem Rechner mit P CPUs als $T_p = T_1/P + T_\infty$. Sie haben zwei Schachprogramme, ein simples und eine optimierte Variante. Das simple Schachprogramm ist charakterisiert durch Parameter $T_1 = 2048$ Sekunden und $T_\infty = 1$ Sekunde. Wenn Sie das Programm auf einem Rechner mit 32 Prozessoren ausführen, beträgt die Laufzeit 65s. Für ihr optimiertes Schachprogramm ist $T_1' = 1024$ und $T_\infty' = 8$. Warum kann man das Programm als optimiert bezeichnen? Wenn Sie das Programm auf dem 32-core Rechner ausführen, beträgt die Laufzeit 40s. Welches Programm skaliert besser auf einem Rechner mit 512 Prozessoren?

   .....

   .....

   .....

   .....

   .....

   .....

(c) (2 points) Warum ist ein Programm mit mehreren Threads schwieriger zum testen als eines welches nur einen einzigen Thread benutzt?

(d) (2 points) Sie haben ein hochgradig parallelisierbares Programm geschrieben das mehrere Threads benutzt. Prognostizieren Sie den Speedup des Programms wenn 2, 8 und 16 Threads auf einer 8 Prozessor Maschine benutzt werden.

(e) (2 points) Gegeben ist der folgende Java-Code, der zwei Threads benutzt:

```
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;
    }
}
```

Was sind mögliche Outputs, die auf der Konsole ausgegeben werden, wenn die main Methode ausgeführt wird? (Es kann angenommen werden, dass die print Funktion die Argumente auf dem Bildschirm ohne Verzögerung ausgibt. Anders gesagt, wenn die Funktion zurückkehrt ist der String auf dem Bildschirm ausgegeben worden.)

- ☐ `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) Erklären Sie was ein reentrant Lock ist. Ist der intrinsic Lock, welcher für das Java `synchronized` Keyword verwendet wird reentrant oder non-reentrant? Geben Sie ein Codebeispiel, das nicht korrekt funktionieren würde, wenn der intrinsic Lock das Gegenteil ihrer Antwort wäre.

. . . . . .

. . . . . .

. . . . . .

. . . . . .

. . . . . .

. . . . . .

. . . . . .

. . . . . .

. . . . . .

. . . . . .

2. **Deadlock**

Das nachfolgende Java Programm erstellt eine grosse Anzahl von Tasks (Runnables) welche anschliessend parallel ausgeführt werden.

```
// 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);
}
```

Entscheiden Sie für die nachfolgenden Implementationen von createTask() ob diese zu Deadlocks führen können. Begründen Sie ihre Antworten (gegebenenfalls anhand eines Beispiels).

(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);
    } };
}
```

.....

.....

.....

.....

(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);
        } } } };
}
```

.....

.....

.....

.....

(c) (10 points) Viele Sprachen verfügen nicht über einen intrinsic Lock für jedes erzeugte Objekt wie Java. In so einer Sprache kann man Speicherplatz sparen, indem man für alle Objekte eine kleinere Anzahl von Locks erzeugt.

Wir benutzen den folgenden Lock basierend auf einer Semaphore:

```
// 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(); }
}
```

Und die folgende Variante einer Box Class (genannt `Box2`), die den Lock verwendet:

```
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;
  }
```

Wir erzeugen 10 Locks und 1000 Boxen wie folgt:

```
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
    ...
}
```

Ist die Art der Synchronisierung wie oben beschrieben korrekt? Beschreiben sie den Tradeoff der sich ergibt aus der Anzahl von Lock-Objekten die verwendet werden.

Fügen Sie eine Methode `swap(Box2 b)` der Klasse `Box2` hinzu, welche *atomar* die Objekte beider Boxen vertauscht.

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

. . . . . .

. . . . . .

. . . . . .

. . . . . .

. . . . . .

```
}
```

3. **Barriers**

(a) (4 points) Beschreiben Sie in Worten was eine Barrier ist und anhand eines Beispiels, wann eine Barrier benutzt werden kann.

.....

.....

.....

.....

.....

.....

(b) (5 points) Der folgende Code erzeugt N Threads um die Einträge eines Array A der Grösse N zu berechnen. Jeder Thread ist verantwortlich für das Update von exakt einem Arrayeintrag. Thread 0 soll den Inhalt des Arrays nach jeder Iteration ausgeben um die neuen Werte anzuzeigen. Erweitern Sie den unten stehenden Code, um sicher zu sein, dass immer die Werte von A aus der gleichen Iteration ausgegeben werden.

Sie können z.B. die `CyclicBarrier` mit der `await` Methode aus `java.util.concurrent` verwenden, Exception handling kann ignoriert werden).

```
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();
            }
        }
    }
```

(c) (5 points) Die `FaultyBarrier` Klasse ist problematisch, wenn diese in der vorherigen Teilaufgabe anstelle der `CyclicBarrier` verwendet werden würde. Beschreiben Sie das Problem und ein Beispielszenario bei dem das Problem auftritt.

```
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 */ }
    }
  }
}
```

.....

.....

.....

.....

.....

.....

.....

.....

4. **Count threes**

Gegeben sei ein Array mit `N` Integers und der folgende Akkumulator-Algorithmus, um die Anzahl der Elemente mit dem Wert `3` in dem Array zu zählen.

```
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) Anhand dieses Beispiels, beschreiben Sie warum Akkumulator-Algorithmen nicht effizient parallelisiert werden können.

.....

.....

.....

.....

(b) (2 points) Beschreiben Sie einen alternativen Algorithmus, welcher die gleiche Funktionalität gewährleistet, aber parallelisierbar ist. Begründen Sie ihre Antwort.

.....

.....

.....

.....

5. **Concurrent Message Passing**

(a) (5 points) Wir betrachten ein eindimensionales Finite Differenzen Problem für einen Vektor X mit Grösse N und berechnen X^T

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

Beschreiben Sie das Design für einen parallelen Algorithms mit einem Actor-Framework um X^t zu berechnen. Verwenden Sie einen Actor für jedes Vektor-Element. Ihre Beschreibung muss beinhalten, welche Nachrichten zwischen den Actors ausgetauscht werden (wie würden Sie die onReceive Methode in Akka implementieren)?

(b) (10 points) Wir betrachten ein ähnliches Problem, bei dem wir die paarweise Interaktion Y_i für jedes Element X_i in Vector X wie folgt berechnen:

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

In dem unten stehenden Code modellieren wir jedes Vektor-Element X_i als einen Actor, der Y_i berechnet und seinen Wert X_i allen anderen Actors als Message schickt. Komplettieren Sie die Klasse `VectorElementActor` wie beschrieben, um Y zu berechnen. Sie können die `tell` Methode der `ActorRef` benutzen Klasse um dem Actor eine Message zu schicken.

```
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) {

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

(c) (2 points) Wieviele Nachrichten versenden Sie in ihrer Implementierung? Wieviele Message-Channels muss das Actor-System erstellen (ein Channel ist ein bidirektionaler Kanal zwischen zwei Actors)?

.....

.....

(d) (2 points) Nehmen Sie an, dass F symmetrisch ist. Können Sie ihre Implementierung ändern um weniger Nachrichten zu verschicken? Wieviele?

.....

.....

(e) (4 points) Wir wollen die Anzahl der Message-Channels reduzieren und nur noch N Channels verwenden. Beschreiben Sie wie der Algorithmus abgeändert werden kann. Wie viele Messages müssen Sie mit dem neuen Algorithmus senden? Beachten Sie, dass Channel-Endpoints immer fest für genau zwei Actors sind (man kann den Endpoint eines Channels nicht ändern um eine Message an mehrere Actors mit dem gleichen Channel zu schicken). Sie können nicht länger annehmen, dass F symmetrisch ist.

.....

.....

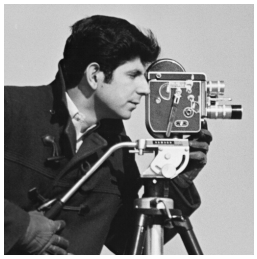
.....

.....

.....

6. **OpenCL: Sobel Filter** Der Sobel-Operator ist eine bekannte Technik aus der Bildbearbeitung. Der Operator berechnet einen räumlichen 2D Gradienten um Regionen mit hohen Frequenzunterschieden hervorzuheben. Dies sind typischerweise Ränder oder Kanten in einem Bild.

   Mit einem Input-Bild (Figure 1a) wird ein Output-Bild (Figure 1b) erzeugt, das nur noch die Kanten des Bildes anzeigt.



(a) Original image.

(b) Output Bild des Sobel Algorithmus.

Abbildung 1: Input and Output des Sobel Algorithmus.

Der Sobel Operator arbeitet mit zwei 3×3 Convolution-Kernel, G_x und G_y, definiert weiter unten. Diese Kernel sind so konstruiert um maximal auf vertikale oder horizontale Kanten in einem Bild (ein 2D Array aus Pixelwerten) zu reagieren, ein Kernel für jede der beiden (horizontale und vertikale) Achsen. Die Kernel berechnen für das mittlere Pixel einer Bildregion X, wie gross die Änderung bezüglich der Helligkeit ist in dieser Region (und dadurch ob das Pixel auf einer Kante liegt oder nicht). Der Pixel-Wert in dem resultierenden Output-Bild (Gradient Magnitude) Y is gegeben als:

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

Abbildung 2 zeigt, wie Sobel angewendet wird auf einen einzelnen 3×3 Block in einem Input-Bild. Es wird der neue Pixelwert für das Output-Bild in der Mitte des 3×3 Block berechnet. Der neue Pixelwert basiert auf der Berechnung für Y. Beachten Sie, dass Pixel an den äusseren Rändern des Bildes nicht berücksichtigt werden, weil wir für deren Berechnung auch Pixelwerte ausserhalb des Bildes berücksichtigen müssten.

Die Filter G_x und G_y sind wie folgt definiert:

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

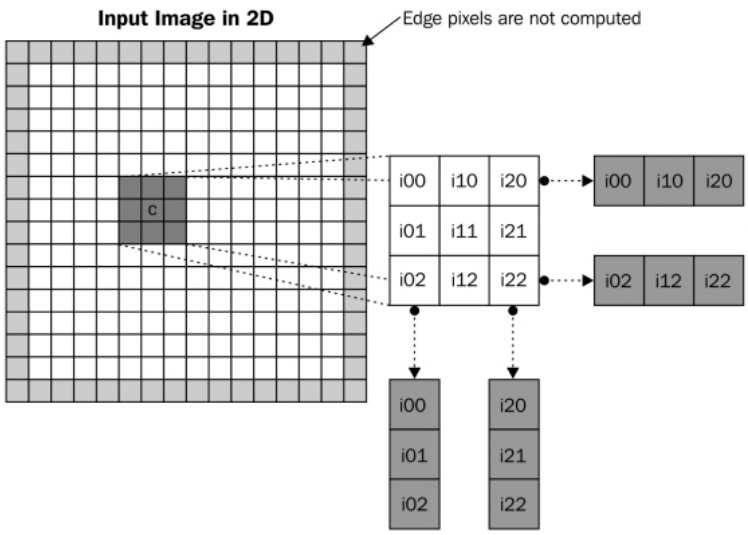


Abbildung 2: Anwendung von Sobel für ein einzelnes Pixel (C) in einem Bild. Die Kolumne i00 – i02 und i20 – i22 werden multipliziert mit G_x. Und die Zeile i00 – i20 und i02 – i22 werden multipliziert mit G_y um den neuen Pixelwert (Y) für das Pixel C zu erhalten.

(a) (3 points) Der Sobel Algorithmus kann sehr effizient auf GPUs ausgeführt werden. Warum? (Hinweis: Welche Schritte beinhaltet die Berechnung von Y?)

.....

.....

.....

.....

.....

(b) (4 points) Gegeben sei die Bildregion X als 3×3 Matrix unten. Berechnen Sie Y mit Sobels Formel.

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

.....

.....

.....

.....

.....

(c) (20 points) Implementieren Sie einen OpenCL-Kernel für Sobels Algorithmus. Der OpenCL Kernel erhält als Input ein schwarz-weiss Bild, repräsentiert durch ein Array von Integerwerten zwischen 0 und 255 (0 für schwarz und 255 repräsentiert weiss). Unser OpenCL Programm ist so konfiguriert, dass der Kernel für jedes Pixel (x, y) des Input-Bildes aufgerufen wird. Das resultierende Output-Bild soll in dem **output** Array abgespeichert werden. (Sie können die OpenCL Funktion hypot(x,y) benutzen um $\sqrt{x^2 + y^2}$ zu berechnen.)

```
__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
```