Linköpings universitet
IDA Department of Computer and Information Sciences
Prof. Dr. Christoph Kessler

TENTAMEN / *EXAM*

TDDD56

Multicore and GPU Programming

24 apr 2014, 14:00–18:00, TER2

Jour: Christoph Kessler (070-3666687, 013-282406), visiting ca. 16:00.
Ingemar Ragnemalm (070-6262628)

Hjälpmedel / *Admitted material:*

– Engelsk ordbok / *Dictionary from English to your native language*

General instructions

- This exam has 8 assignments and 6 pages, including this one.
  Read all assignments carefully and completely before you begin.
- It is recommended that you use a new sheet of paper for each assignment, because they will be corrected by different persons.
  Sort the pages by assignment, number them consecutively and mark each one on top with your exam ID and the course code.
- You may answer in either English or Swedish. **English is preferred** because not all correcting assistants understand Swedish.
- Write clearly. Unreadable text will be ignored.
- Be precise in your statements. Unprecise formulations may lead to a reduction of points.
- Motivate clearly all statements and reasoning.
- Explain calculations and solution procedures.
- The assignments are *not* ordered according to difficulty.
- The exam is designed for 40 points. You may thus plan about 5 minutes per point.
- Grading: U, 3, 4, 5. The preliminary threshold for passing is 20 points.
- An exam review session will be announced on the course homepage.

1. (5 p.) **Multicore Architecture Concepts**

   (a) Define and explain the following technical terms:

      i. Hardware multithreading
      ii. Sequential (memory) consistency
      iii. Heterogeneous multicore system

      (Remember that an example is not a definition. Be general and thorough.) (3p)

   (b) Recent many-core processor architectures (such as Intel SCC, Kalray MPPA, Tilera TILE64) are increasingly organized in a way that differs from standard multicore (dual-core, quadcore...) processor architectures. What is the main difference, what is the technical reason for it, and what does it imply for the programmer? (2p)

2. (2 p.) **Parallel Programming with Tasks**

   How does a work-stealing scheduler work?

3. (7 p.) **Non-blocking Synchronization**

(a) Name 2 problems of lock-based synchronization that are removed by non-blocking synchronization. (1p)

(b) An *ordered linked list* (here, for integers) uses list items of the following type:

```
struct elem {
  int value;
  struct elem *next;
}
```

Pointer variable `head` points to the first list element. Insertion of a new element with value v is done by searching for v and inserting a new list element `e` for v after the element where the search ended. As a simplification we assume that the list always contains at least one element, namely an artificial dummy element with value $-\infty$ as the first element in the list. We also assume for simplicity that *no remove operations* occur.

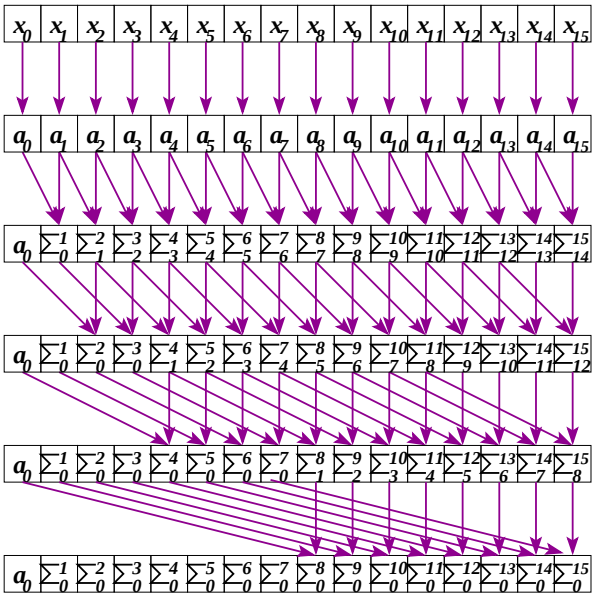
```
struct elem *e = (struct elem *)malloc(sizeof(struct elem));
struct elem *p = head;
// assume for simplicity that the list contains at least 1 element
// (the first element is a dummy element with value -infinity)
while (p->next!=NULL && p->next->value < v)
  p = p->next;
e->value = v;
// insert e into the list after p:
e->next = p->next;
p->next = e;
```

Assume that the list elements and the `head` pointer are stored in shared memory.

i. Show that, without proper synchronization, two concurrent insertions may lead to an incorrect result. (Give a simple scenario, start with a one-element list).
In general, what is the condition (on concurrently inserted values and list contents) for such a conflict to occur?
Suggest a simple mutex-lock-based solution to make insertion thread-safe. (1.5p)

ii. Use appropriate CAS operations (i.e., no mutex locks) to provide a non-blocking *insert* operation (pseudocode). Explain your code.
Explain how possible conflicts between concurrent *insert* operations are recognized and handled properly by your implementation.
In particular, argue why even in the case of conflicts at least one of the conflicting operations will succeed. (3p)

(c) Do you know another kind of hardware atomic operation that can be used as an alternative to CAS but does not suffer from the ABA problem? How does it work (from the programmer's point of view)? (1.5p)



EREW parallel prefix sums, pseudocode:

```
float x : array[0..N − 1]; // input array
float a : array[0..N − 1]; // output array
int dist;
forall i : [0..N − 1] in parallel do
    // using N PRAM processors
    a[i] ← x[i]; // copy x to a
od
dist ← 1;
while dist < N do
    forall i : [0..N − 1] in parallel do
        // using N PRAM processors
        if i ≥dist then
            a[i] ← a[i−dist ] + a[i];
    od
    dist ← dist * 2;
od
```

Figure 1: EREW parallel prefix algorithm, with an illustration for $N = 16$ elements and processors.

4. (8 p.) **Design and Analysis of Parallel Algorithms**

   (a) Formulate Brent's Theorem (explained formula), give its interpretation, derive it (calculation), and describe its implications for parallel algorithm design and analysis. (3p)

   (b) The *parallel prefix sums* problem is defined as follows:
   Given a set S (e.g., the integers) and a sequence of N elements $x_0, \ldots, x_{N-1} \in S$, compute the sequence a of *prefix sums* defined by

$$a_i = \sum_{j=0}^{i} x_j \text{ for } 0 \leq i < N$$

   The massively parallel algorithm in Figure 1 has been proposed for the EREW (exclusive read, exclusive write) PRAM model; it uses one processor (hardware thread) per element in the input sequence x.
   (i) Explain why the algorithm is an EREW algorithm. (1p)
   (ii) Analyze the parallel execution time, parallel work and parallel cost (as formulas in N, using big-O notation) of this algorithm for a problem size N using N processors.
   (A solid derivation of the formulas is expected, just guessing the right answer gives no points.) (3p)
   (iii) Is this algorithm *work-optimal*? Justify your answer (formal argument). (1p)

[In case of questions about the following 3 assignments, ask I. Ragnemalm in the first hand.]

5. **GPU algorithms**

   Describe, in code or sufficient detail, how matrix multiplication of large matrices can be implemented on the GPU. (GPU kernel code only.) Emphasize the most vital considerations for good performance. (5p)

6. **GPU computing**

   (a) Describe three sorting algorithms in terms of their suitability for GPU implementation. Computational complexity should be considered. (3p)

   (b) The GPU design in centered around a number of features vital for its primary use, graphics. List three such features, as significant as possible, which are also important for GPU computing and assess their importance. (2p)

7. **GPU Quickies**

   (a) Where does GPU computing fit in Flynn's taxonomy? What name(s) does the architecture type have according to Flynn's taxonomy? (1p)

   (b) What does a Streaming Multiprocessor correspond to in CUDA and OpenCL, respectively? (1p)

   (c) What kind of algorithms benefit from using constant memory? (1p)

   (d) Some operations can be implemented either as scatter or gather operations. Which is most suitable for parallel implementation (on GPUs in particular)? Why? (1p)

   (e) In his guest lecture, Dr Garrido presented an optimized FFT implementation. Describe one trick or consideration beyond a trivial parallelization and the everyday rules of GPU implementation that can cause additional speedups, preferably from that presentation. (1p)

8. (3 p.) **Optimization and Parallelization**

   (a) Consider the following sequential loop:

```
void folr( float *x, float *a, float *b, int N )
{
  int i;
  x[0] = b[0];
  for (i=1; i<N; i++)
     x[i] = b[i] + a[i] * x[i-1];
}
```

   Assume that the arrays `x`, `a` and `b` do not overlap. Could the iterations of this loop be run in parallel? Justify your answer *(formal argument). (If you need to make additional assumptions, state them carefully.)* (1p)

   (b) What is the main motivation and the general principle (no details) of *dynamic* (i.e., runtime) parallelization of sequential loops? (1p)

   Concretely, for what kind of loops could it be suitable? (0.5p)

   What is its main drawback compared to static (i.e., compile-time) parallelization? (0.5p)

Good luck!