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

TENTAMEN / *EXAM*

TDDD56

Multicore and GPU Programming

16 jan 2014, 14:00–18:00, T2, U1, U3, (T1)

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) What is the *power wall* in (single-core) processor architecture, and why does it lead to the development of multicore architectures? Give a quantitative argument. (2p)

   (b) Define and explain the following technical terms:

      i. SIMD (vector) instructions

      ii. Moore's Law

      iii. Cache coherence

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

2. (7.5 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:
p->next = e;
e->next = p->next;
```

   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)

(d) A core component of task-based parallel programming environments is a dynamic (user-level) task scheduler that maps tasks to worker threads at runtime. Why could such a dynamic task scheduler possibly benefit from using a non-blocking shared data structure (instead of a lock-protected one)? (Short answer) (0.5p)

3. (6.5 p.) **Design and Analysis of Parallel Algorithms**

Finding the roots of a parallel forest

A *parallel binary forest* is a large shared array F of N elements of the type

```
struct treeelem {
  struct treeelem *leftchild, *rightchild; // given as input
  struct treeelem *root; // to be computed
  struct treeelem *parent; // auxiliary pointer, not given
  // ... further entries
} F[N];
```

As known from sequential computing, a forest contains (usually, several) trees. Each node has at most one parent node; the nodes without parent are called the *roots* of the forest. The forest nodes are stored in arbitrary order in the array `F`. The child pointers are given as input; for leaf nodes the child pointers are NULL. Each node belongs to one tree, which is uniquely determined by (the address of) its root node. The task is to calculate, for each node, a pointer `root` to the root of the tree that it belongs to (see Figure 1, shown for node v).

(a) Develop a parallel algorithm for finding the roots of the N forest nodes in parallel time $O(\log N)$ on a CREW (Concurrent Read, Exclusive Write) PRAM with N processors.
(Hint: Work in two steps: 1. Use the auxiliary pointers `parent` and show how to calculate them quickly in parallel for all elements as a first step, so that for each non-root node its `parent` pointer now points to its direct parent node in the forest, and for each root node its `parent` pointer should be NULL. — 2. Now develop an efficient method to calculate the `root` pointers with the help of the `parent` pointers. You might use a technique known from the lecture.)
Show and explain the resulting pseudocode.
Show why its parallel time complexity is $O(\log N)$.
Explain why your algorithm matches the CREW constraint for the shared memory accesses, i.e., where concurrent read is used in the algorithm and why concurrent write access does not occur for any memory location. (5p)

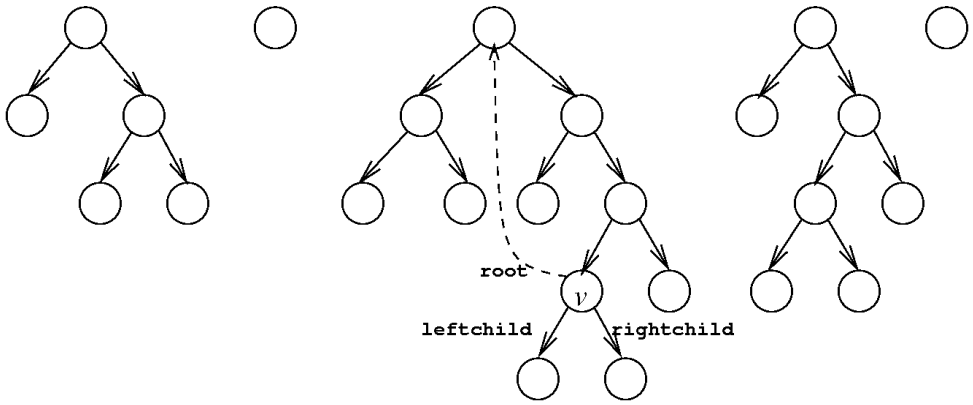

Figure 1: Forest example. The given pointers to the children are shown (solid arrows). The pointer `root` is to be computed for each node; it is shown here for node v only (dashed arrow).

(b) Analyze the algorithm for its asymptotic parallel work and parallel cost (each as a formula in N, using $\Theta()$ notation). Explain. (1.5p)

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

4. **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)

5. **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)

6. **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)

7. (3 p.) **Optimization and Parallelization**

   (a) Define and explain the following term: (1p)

      i. Loop-carried data dependence

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

8. (3 p.) **Parallel algorithmic design patterns and High-level parallel programming**

   Explain the main idea of parallel and accelerator programming using *algorithmic skeletons*. Make sure to explain what skeletons are and how they are used. Give two main advantages and two main drawbacks of skeleton-based parallel programming. (3p)

Good luck!