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

TENTAMEN / *EXAM*

TDDD56

Multicore and GPU Programming

11 jan 2019, 14:00–18:00, U2, T2

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

Hjälpmedel / *Admitted material:*

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

General instructions

- This exam has 8 assignments and 5 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 may 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.
- We expect to have the exam corrected by ca. *25 january*. An exam review session around end of january will be announced on the course homepage.

1. (6 p.) **Multicore Architecture Concepts**

   (a) Define and explain the following technical terms:

      i. MIMD parallelism
      ii. Hardware multithreading
      iii. Last-level cache
      iv. False sharing
      v. Heterogeneous multicore system

      (Remember that an example is not a definition. Be general and thorough.) (5p)

   (b) Why does the MSI coherence protocol, as described in the lecture, guarantee sequential (memory) consistency? (1p)

2. (9 p.) **Design and Analysis of Parallel Algorithms**

   (a) Define the following two properties:

      - Critical path length
      - Parallel work

      of a parallel algorithm, and explain (commented formula) how they are related to its parallel execution time with p processors through Brent's Theorem. (2p)

      (Hint: Make sure to properly introduce all symbols used and explain their meaning.)

   (b) **Exclusive Parallel Prefix Sums and Skeleton Programming** (3p)

      In the lectures we derived several parallel algorithms for the prefix-sums problem. We distinguish between inclusive and exclusive prefix sums.

      The *inclusive prefix-sums* problem for an array x of N elements consists in computing an array y of N elements with

$$y_i = \sum_{j=0}^{i} x_j \text{ for } i = 0, ..., N-1.$$

      The *exclusive prefix-sums* problem for an array x of N elements consists in computing an array y of N elements with

$$y_i = \sum_{j=0}^{i-1} x_j \text{ for } i = 0, ..., N-1.$$

      (i) Describe a parallel (EREW or CREW PRAM) algorithm for computing *inclusive* prefix-sums in parallel time $O(\log N)$ with N processors, and explain (calculation) why it has logarithmic time complexity. (1.5p)

      (ii) Exclusive parallel prefix sums can be computed easily from inclusive parallel prefix sums—How?

      Express the resulting parallel computation of *exclusive* prefix-sums as pseudocode using suitable (SkePU-like) *skeletons* and suitable data structures. (1.5p)

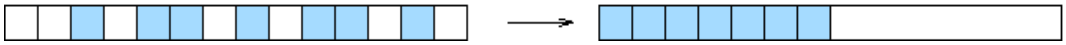
(c) **Parallel Memory Compaction in Garbage Collection** (4p)

Given is a shared heap for linked list items, organized as shared array A of N list items:

```
struct listitem {
   struct element {...} data;
   struct listitem *next;
   int reachable; // auxiliary field for garbage collection, given
   int newindex;  // auxiliary field for garbage collection
} A[N];           // shared parallel array of N list items
```

Some programming languages offer automatic memory management for the heap, i.e., list items need not be freed explicitly. Instead, the language's runtime system (e.g., the Java Virtual Machine) now and then runs a garbage collection pass that scans all heap elements and frees those that are no longer reachable from live pointer variables residing on the stack (such as local pointer variables) or from global pointer variables.

A garbage collection pass starts with marking all elements (by setting `reachable` to 1 or 0) whether they are reachable from live pointer variables or not. Assume for simplicity that this phase has already been done, i.e., `reachable` is 1 for reachable list items, and 0 for the others.



Develop a parallel (PRAM) algorithm for N processors that in time $O(\log N)$ compacts the heap, i.e., moves all k reachable elements "to the left" in A, i.e., they are then stored consecutively at their new index in $A[0:k-1]$, see the figure. (For now, don't worry about fixing the `next` pointers).

(*Hint:* Initialize, for each list item $A[i]$, its `newindex` field to a suitable value (which one? It should depend on whether $A[i]$ is reachable or not). Then apply a certain fundamental parallel operation from the lecture (which one?) over the `newindex` values to determine, for each list item $A[i]$, the overall number of reachable list items in $A[0:i-1]$, which, in the case of a reachable $A[i]$, will be just the new index position for $A[i]$ that we are looking for. Now only one step is missing...)

Show and explain the pseudocode, and analyze its parallel *time*, *work* and *cost*, and the parallel *speedup* over a straightforward sequential compaction algorithm (asymptotic notation in all cases; state which PRAM model variant you assume). (4p)

Bonus question (+1p, provided that the main question was solved properly):

Explain how to extend this parallel algorithm to also fix the `next` pointers properly when doing the compaction.

(Hint: If a list item is reachable, then its list successor (`next`) element is also reachable.)

3. (3 p.) **Parallel Programming with Threads and Tasks**

   (a) What does *thread pinning* mean? (1p)

   (b) What is the purpose of back-off strategies in mutex-lock implementations? (1p)

   (c) How does a work-stealing task scheduler work? (1p)

4. (4 p.) **Non-blocking Synchronization**

   (a) The operation $y = \mathit{fetch\_and\_add}(p, a)$ atomically adds to the memory location pointed to by p the integer value a and returns the (integer) value y that the location had immediately before the update.

   You are given a multicore processor that has no atomic *fetch_and_add* instruction but that provides an atomic *compare_and_swap* (CAS) instruction instead. Write a software implementation of atomic *fetch_and_add* using CAS. Explain your solution. (2p)

   (b) Explain the operations Load-Linked (LL) and Store-Conditional (SC), and explain how the behavior of LL+SC differs from that of CAS if used for atomic updating of a shared memory location in the context of lock-free shared data structures. If you had the choice between LL+SC and CAS, which one would you prefer, and why? (2p)

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

5. **GPU Algorithms and Coding** (5p)

   (a) Describe, with figures and pseudo code, how to apply an optimized convolution filter on a color image with a GPU, using a convolution kernel like the one below or similar, possibly larger. Detailed code is not demanded, but vital considerations for performance and correct results should be included. You should not expect the kernel to be separable. (3p)

1	4	6	4	1
4	16	24	16	4
6	24	36	24	6
4	16	24	16	4
1	4	6	4	1

/256

   (b) A *histogram* is an array `h` that records for each possible (integer) value the number of its occurrences in a large set of integers, e.g., an array `a`. It can be computed like this:

```
for all elements i in a[] do
    h[a[i]] += 1
```

Write code (close to actual code) for running this algorithm with good performance on a GPU (e.g. using CUDA or OpenCL). (2p)

6. **GPU Architecture concepts** (5p)

   (a) In order to get the best performance from shared memory, what should the access pattern be? Clarify with a figure, including how to improve a bad access pattern. (2p)

   (b) Some image filters are separable. Describe how and why this works and give an example of a separable filter. This has potential to improve performance. Why? (2p)

   (c) A typical case of separable filter is to split a filter into one horizontal and one vertical filter, often of the same size. However, the two parts may each run with significantly different performance, one much faster than the other. Suggest a likely reason why this could happen. (1p)

7. **GPU Quickies** (5p)

   (a) What do you need to do to get coalesced memory access? (1p)

   (b) GPUs have evolved around the needs of graphics applications. Give an example feature, apart from shaders and multiple threads, that was added for the needs of graphics which is valuable for GPU computing. (1p)

   (c) Texture access provides two unique features that we otherwise do not have. Name one, and describe with a brief sentence. (1p)

   (d) In GPU Computing using the graphics pipeline, in what stage are the computations usually carried out? (1p)

   (e) Compare OpenCL and Compute Shaders in terms of portability. You should know at least one strong point of each. (1p)

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

   (a) Consider the following loop nest:

```
for i = 1, ..., M
   for j = 1, ..., N-1
      A[i][j] = x*A[i-1][j-1] + y*A[i-1][j] + z*A[i-1][j+1];
```

   Would it be correct to apply *loop interchange* to this loop nest? Justify your answer (dependence-based argument). (1p)

   (b) Give a sufficient condition (dependence based argument) for that a loop's iterations can be executed in parallel. (1p)

   (c) Why is it, in general, so hard for C/C++ compilers to statically analyze a given sequential legacy program and parallelize it automatically? (1p)

Good luck!