Examination 2023-10-20

Introduction to Parallel Programming (5 hours)

Please make sure to write your exam code on each page you hand in. When you are finished, please put the pages withyour answers together in an order that corresponds to the order of the questions, and mark the questions you have answered on the exam front sheet. You can take with you this exam if you want.
This is a **closed book** examination but you are allowed to have with you an **A4 sheet of prepared hand-written notes**. The exam contains **40** points in total and their distribution between sub-questions is clearly shown. To get a final grade of **5** you must score at least **34**. To get a grade of **4** you must score at least **28** and to pass the exam you must score at least **19**. Note that you will get **credit only for answers that are correct (or a very serious attempt) and clear**. All answers should preferably be **in English**; if you are uncomfortable with English you can of course use Swedish. Whenever in *real doubt* for what a particular question might mean, **state your assumptions clearly**.

DON'T PANIC!

1. **General Questions ($3 \times 2 = 6$ pts total).** Give brief answers to the following questions:

   (a) What is Amdahl's law and what consequences does it have for parallel programming?

   (b) What are the main implementation ingredients (i.e., what fields are typically needed) in the implementation of read-write locks?

   (c) Why/how does backoff improve the performance of locking compared to TAS and TATAS locks?

2. **Parallelization (6 pts).** You have created a program that finds all files in a given directory of your hard drive, and scans each one for the sequence of bits `101010`. If a file contains this sequence, it ouputs its name to standard output. However, when running the program you quickly realize that you have far too many files for running on just one thread.

   Describe a method of parallelizing this program. Make sure to answer the following questions:

   (a) What types of synchronization would you employ?

   (b) How would you balance the load?

   (c) Assuming a very large and complex directory structure, what speedup would you expect to achieve with your solution if you double the number of concurrently running threads?

   Explain your method and your answers!

3. **Threads and Locks ($3 \times 2 = 6$ pts total).** A *stack* is a data structure that resembles a pile of plates; when hungry, you take (`pop`) the top plate from the pile and eat, and when you are finished eating you wash and dry the plate and then you put (`push`) the cleaned plate on top of the remaining plates again.

   The following C++ code implements a stack, but instead of plates we use integers, and the `pop` operation returns `-1` if the stack is empty.

```
1      struct Node {
2        int value;
3        Node *next;
4        Node(int v, Node *n) : value(v), next(n) {}
5      };
6      class Stack {
7       private:
8        Node *head = nullptr;
9       public:
10       void push(int v) {
11         head = new Node(v, head);
12       }
13       int pop() {
14         if (head == nullptr) {
15           return -1;
16         }
17         Node *n = head;
18         head = head->next;
19         int v = n->value;
20         delete n;
21         return v;
22       }
23     };
```

   This implementation works well for single-threaded applications, but usually crashes with segmentation faults as soon as more than one thread is using it.

   (a) Describe an execution in which two or more threads using the stack results in a crash.

   (b) In order to alleviate this problem, you decide to protect the stack by adding *coarse grained* mutex locks. After which line would you *declare* the lock, and after which lines would you *lock* and *unlock* it?

   (c) After finalizing your now thread-safe stack, you decide to benchmark your program. You create a benchmark where each thread performs as many (random) operations it can in 10 seconds, where the majority are `push` operations. Assuming N operations per second on one thread, how many do you expect for four threads? Explain your reasoning!

4. **OpenMP (4 pts).** A student that was passing by the lecture hall of the OpenMP lecture heard that with a simple `#pragma omp parallel for` he can parallelize a serial code in just a minute! He applied the "advice" in the `for` loops of his program, he run it on a (large) multicore but did not notice any time difference. Explain to him possible reasons for the behaviour he is experiencing (the more, the merrier).

5. **OpenMP Programming (2 × 3 = 6 pts total).** Consider the following program to sort an array. The `merge_sort` function takes pointers to the beginning and end of the array and sorts it. The function `merge` takes two sorted sub-arrays (from `begin` to `mid` and from `mid` to `end`) and merges them so that all elements from `begin` to `end` are sorted.

```
void merge_sort(int* begin, int* end) {
  if (begin < end) {
    int* mid = begin + (end - begin) / 2;
    merge_sort(begin, mid);
    merge_sort(mid, end);
    merge(begin, mid, end);
  }
}
```

(a) Parallelize this function by adding OpenMP annotations *only*.

(b) Do you see any problems in your solution that might affect performance? If not, explain why. If yes, propose possible solution(s) and explain the problems that these solve.

6. **Queue Locks (2 × 3 = 6 pts total).** The code shown below is an alternative implementation of CLHLock in which a thread reuses its own node instead of its predecessor node.

```
public class BadCLHLock implements Lock {
  // most recent lock holder
  AtomicReference<Qnode> tail = new AtomicReference<QNode>(new QNode());
  // thread-local variable
  ThreadLocal<Qnode> myNode = new ThreadLocal<QNode> {
    protected QNode initialValue() {
      return new QNode();
    }
  };
  public void lock () {
    Qnode qnode = myNode.get();
    qnode.locked = true; // I'm not done
    // Make me the new tail, and find my predecessor
    Qnode pred = tail.getAndSet(qnode);
    // Spin while the predecessor holds lock
    while (pred.locked) {}
  }
  public void unlock() {
    // reuse my node next time
    myNode.get().locked = false;
  }
  static class Qnode { // Queue node inner class
    public boolean locked = false;
  }
}
```

(a) How can this implementation go wrong? Explain.

(b) How does the MCS lock avoid the problem even though it reuses thread-local nodes?

7. **MPI Programming (2 × 3 = 6 pts total).** In lecture 11, we examined *MPI Collective Communication* and saw different ways that communication can be depicted using trees. We also discussed the functions `MPI_Scatter` and `MPI_Gather`. Suppose `comm_sz` = 8 and `n` = 16.

   (a) Draw a diagram that shows how `MPI_Scatter` can be implemented using tree-structured communication with `comm_sz` processes when process 0 needs to distribute an array containing `n` elements.

   (b) Draw a diagram that shows how `MPI_Gather` can be implemented using tree-structured communication when an `n`-element array that has been distributed among `comm_sz` processes needs to be gathered onto process 0.

Good luck !