Examination 2022-10-21

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 containing your solutions together in an order that corresponds to the order of the questions. 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 might 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 (6 pts total; 2 pts each).** Give brief answers to the following questions:
   (a) What are read-write locks and what are their main advantages compared to mutex locks?
   (b) What do the terms blocking and non-blocking synchronization mean? How do these two synchronization mechanisms compare against each other?
   (c) Why in some OpenMP programs is it necessary to use the `omp_get_num_threads()` call inside an `omp parallel` section instead of using the `NUM_THREADS` constant that has been set globally or the number of threads we specify in the corresponding `#pragma`?

---

2. **Amdahl's Law ($2 + 2 = 4$ pts total).** Use Amdahl's Law to resolve the following questions:
   (a) Suppose a computer program has a method M that cannot be parallelized, and that this method accounts for 40% of the program's execution time. What is the limit for the overall speedup that can be achieved by running the program on an n-processor multiprocessor machine?
   (b) Suppose the method M can be sped up four-fold. What fraction of the overall execution time must M account for in order to double the overall speedup of the program?

---

3. **Safety and Liveness (4 pts total).** For each of the following, state whether it is a safety or liveness property. Identify the "bad" or "good" thing of interest.
   (a) Two things are certain: death and taxes.
   (b) Customers are served in the order they arrive.

(c) The cost of living never decreases.

(d) If an interrupt occurs, then a message is printed within one second.

You will get 1 point for each correct answer and lose 1 point for each wrong answer, so don't just guess. But, of course, your overall grade in this question cannot be negative.

---

4. **Safe Locking (4 pts).** When using a linked list data structure to e.g., represent a *set*, the operations `insert` and `delete` conceptually consist of two distinct "phases." In the first phase, both operations search the list for either the position of the new node or the position of the node to be deleted. If the outcome of the first phase so indicates, in the second phase a new node is inserted or an existing node is deleted. In fact, it is quite common for linked list programs to split each of these operations into two function calls. For both operations, the first phase involves only read-access to the list; only the second phase modifies the list. Would it be safe to lock the list using a read-lock for the first phase? And then to lock the list using a write-lock for the second phase? Explain your answer.

---

5. **Queue Locks ($2 \times 3 = 6$ 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?

---

6. **OpenMP Programming (6 pts total).** A programmer has parallelized a `foo` function using OpenMP pragmas as follows:

```
int foo(int n, int *a, int *b, int *c, int *d) {
  double tmp, total = 0;
  #pragma omp parallel
  {
    #pragma omp sections
    {
      #pragma omp section
      {
        #pragma omp for
        for (int i = 0; i < n; i++) {
          a[b[i]] += b[i];
          total += b[i];
        }
      }
      #pragma omp section
      {
        #pragma omp for
        for (int i = 0; i < n; i++) {
          tmp = c[i];
          c[i] = d[i];
          d[i] = tmp;
          total += c[i];
        }
        #pragma omp for
        for (int i = 0; i < n; i++) {
          total += d[i];
        }
      }
    }
  }
  return total;
}
```

If you think there are any issues (i.e., errors, omissions in the pragmas, or pragmas which are missing) with this code, explain why they are issues and suggest a (good) way to fix them.

---

7. **MPI Programming (3 + 3 = 6 pts total).** A group of students have written a program that uses `MPI_Allreduce` to compute a sum. They expect that each process will send its value to each other process, such that they each add their sums up, for a total of 49 messages. They run the program with 8 processes, but when they measure the number of messages sent, they find that only 14 messages are sent.

   (a) Explain how a clever implementation of `MPI_Allreduce` can reach this result. Draw a diagram denoting the messages sent.

Another group of students runs the same program on a different set of machines, with far greater latency; it takes an entire second for a message to travel from the sender to the receiver.

The performance is poor, of course, but then they remember another way `MPI_Allgather` could send data; one that consists of more messages —but fewer steps— than the one in (a).

(b) Draw a diagram denoting the messages sent. Does this method perform better than the method in (a) for this configuration of machines? Explain your reasoning.

---

8. **More OpenMP Programming (2 + 2 = 4 pts total).** Consider the following program:

```
#define DIM 1000

void randomize_array(char * arr) {
  // Puts random values in array
}

int main (int argc, char ** argv) {
  int* x = malloc(DIM * DIM);
  int* y = malloc(DIM * DIM);
  int* z = malloc(DIM * DIM);

  randomize_array(x);
  randomize_array(y);

  #pragma omp parallel for collapse(2)
  for (int i = 0; i < DIM; i++)
    for (int j = 0; j < DIM; j++) {
      z[i][j] = 0;
    #pragma omp parallel for
    for (int k = 0; k < DIM; k++) {
      result = x[i][k] * y[k][j];
      z[i][j] += result;
    }
  }
  // Print output
  return 0;
}
```

(a) This OpenMP program has some correctness issues. Can you find them?

(b) Provide a version of the `main()` function with these issues fixed. Feel free to rewrite its code slightly if you think this is needed or it simplifies things.

Good luck !