Examination 2020-12-21

Introduction to Parallel Programming (1DL530) – 5 hours

This is an examination **done at home**, so you can have your books, lecture slides, and notes open when you take it. You also need to have access to a PC or server where you can compiler and run C/C++ programs with Pthreads. Please refrain from searching the web for answers to these questions but, if you do, **you need to explicitly mention these sources**. Please avoid submitting hand-written text or low-quality pictures from mobile phones; instead **use an editor or word processor** for all your answers in text.

When you finish, you need to **upload your program(s) and a single PDF with your answers and with instructions on how to compile and run your program(s)** to your group in the Studium. Also, remember to **write your anonymous code in both your program and in the PDF with your answers**. You should stop answering qurstions at 13:00, and your submission **must be uploaded by 13:30**. In case of trouble with Studium, you can mail your solutions to the teacher, but your mail also needs to be sent before 13:30.

The exam contains **40** points in total and their distribution between sub-questions is clearly identifiable. 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**.
Whenever in *real doubt* for what a particular question might mean, **state your assumptions clearly**.

DON'T PANIC!

---

1. **Efficiency of Parallelism (8 pts total; 2 each).**

   - Suppose that the runtime of a serial program is given by $T_{serial} = n^2$, where n is the size of the program's input and the units of the runtime are in microseconds. Suppose that a parallelization of this program has runtime $T_{parallel} = n^2/p + log_2(p)$.
     (a) What would happen to the speedups and efficiencies of the program as p is increased and n is held fixed?
     (b) What would happen if p is held fixed and n is increased?
   - Suppose that $T_{parallel} = T_{serial}/p + T_{overhead}$. Also suppose that we fix p and increase the problem size.
     (c) Show that if $T_{overhead}$ grows more slowly than T_{serial}, the parallel efficiency will increase as we increase the problem size.
     (d) Show that if, on the other hand, $T_{overhead}$ grows faster than T_{serial}, the parallel efficiency will decrease as we increase the problem size.

---

2. **Safe Locking (5 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.

---

3. **Synchronization Using Pthreads (4 + 2 + 2 + 1 = 9 pts total).** Although producer-consumer synchronization is easy to implement with semaphores, it is also possible to implement it with mutexes. The basic idea is to have the producer and the consumer share a mutex. A flag variable that is initialized to false by the main thread indicates whether there is anything to consume. With two threads we would execute something like this:

```
while (1) {
  pthread_mutex_lock(&mutex);
  if (my_rank == consumer) {
    if (message_available) {
      print message;
      pthread_mutex_unlock(&mutex);
      break;
    }
  } else { /* my_rank == producer */
    create message;
    message_available = 1;
    pthread_mutex_unlock(&mutex);
    break;
  }
  pthread_mutex_unlock(&mutex);
}
```

So if the consumer gets into the loop first, it will see there is no message available and return to the call to `pthread_mutex_lock`. It will continue this process until the producer creates the message.

(a) Write a Pthreads program, in either C or C++, that implements this version of producer-consumer synchronization with two threads.

(b) Can you generalize this so that it works with $2k$ threads (where odd-ranked threads are consumers and even-ranked threads are producers)?

(c) Can you generalize this so that each thread is both a producer and a consumer? For example, suppose that thread q "sends" a message to thread $(q + 1)$ `mod` t and "receives" a message from thread $(q - 1 + t)$ `mod` t?

(d) Does this use busy-waiting?

---

4. **OpenMP (4 pts).** Consider the following loop that we want to parallelize using OpenMP.

```
a[0] = 0;
for (i = 1; i < n; i++)
  a[i] = a[i-1] + i;
```

There's clearly a loop-carried dependence, as the value of `a[i]` cannot be computed without the value of `a[i-1]`. Can you see a way to eliminate this dependence and parallelize the loop?

---

5. **More OpenMP (3 + 2 = 5 pts total).** Consider the following recursive function to compute the depth of a binary tree.

```
int getDepth() {
  int hL = 0; int hR = 0;
  if (this.left  != null) { hL = this.left.getDepth(); }
  if (this.right != null) { hR = this.right.getDepth(); }

  return 1 + max(hL, hR);
}
```

   (a) Rewrite this function by adding pseudo-OpenMP annotations.

   (b) Assume that `this.left` and `this.right` have an operation `.getSize()` that returns the number of values these subtrees contain in constant time (A tree without children has size 1, a tree with two children but no "grand children" has size 3, etc.). Use this operation to speed up your function from question 5a. Explain what problem this solves.

---

6. **Parallel Performance (2 + 3 = 5 pts totel).** Your friends are devastated: their program does not scale! Luckily, you are around. Your friends' program runs four threads, each thread executes the function `doWork`. Here is how it looks like:

```
float partialResults[4];

void doWork(int threadID) {
  for (int i = 0; i < 10000000; ++i)
    partialResults[threadID] += computeSegment(threadID, i);
}

int main(...) {
  ... // initialze partialResults, spawn worker threads, then join them
  float sum = 0.0;
  for (int i = 0; i < 4; ++i) { sum += partialResults[i]; }
  printf("sum = %f\n", sum);
}
```

The program has very short sequential parts, so your friends expected their program to scale linearly. However, the runtime barely improves for two threads, and it gets even worse for more threads than two. The function `computeSegment` is not our concern, it does only boring, but tedious mathematical computations.

   (a) What is the problem, in your opinion?

   (b) How would you fix the problem? Rewrite the relevant parts of the code.

7. **MPI Programming (4 pts).** In lecture 11, we examined various forms of MPI communication. Suppose `comm_sz` $= 8$ and the vector `x` $= (0, 1, 2, \ldots 15)$ has been distributed among the processes using a block distribution. Draw a diagram illustrating the steps in a butterfly implementation of allgather of `x`.

Good luck !