Examination 2021-08-26

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 compile and run C/C++ programs with MPI. 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 questions at 19:00, and your submission **must be uploaded by 19:30**. In case of trouble with Studium, you can mail your solutions to the teacher, but your mail also needs to be sent before 19:30.

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. **Amdahl's Law (3 + 3 = 6 pts total).**
   (a) You have a choice between buying one uniprocessor that executes five zillion instructions per second, or a ten-processor multiprocessor where each processor executes one zillion instructions per second. Using Amdahl's Law, explain how you would decide which to buy for a particular application. (A *zillion* is a fictitious, indefinitely large number.)
   (b) A friend of yours who works for a company developing parallel software claims that, despite Amdahl's law, in practice many parallel programs obtain excellent speedups. Can you think of reasons for this to hold or is your friend exaggerating?

2. **Concurrency Errors (6 pts total; 3 each).** Give an example of a linked list implementation (using C++/Pthreads pseudocode) and a *sequence of memory accesses* to the linked list in which the following pairs of operations can potentially result in problems:
   (a) Two `delete`s executed concurrenty.
   (b) An `insert` and a `delete` executed concurrenty.

   You do not need to submit fully working code. But please make sure that the chronological order of the interleaving in the sequence of memory accesses that shows the problem is clear.

3. **Data Races (6 pts total; 3 each).** Which of the following programs are data race free (i.e., they do not contain a data race)? Justify your answer by either showing a 'racy' execution or by giving a reason why there cannot be a data race.

   (a) Thread 1: `lock m; *x = 1; unlock m; *y = 1`
   Thread 2: `lock m; r = *x; unlock m; if (r = 1) then print *y`

   (b) Thread 1: `*y = 1; lock m; *x = 1; unlock m;`
   Thread 2: `lock m; r = *x; unlock m; if (r = 1) then print *y`

   (c) Thread 1: `*y = 1; lock m; *x = 1; unlock m;`
   Thread 2: `lock m; r = *x; unlock m; if (*x = 1) then print *y`

   where `m` is a monitor, `x` and `y` shared-memory locations and `r` is a thread-local variable. Assume that all memory locations are zero-initialised.

---

4. **Integration Using Threads (6 pts total; 3 each).** A program spawns T threads to calculate, numerically, the area under a function from 0 to 1: $\int_0^1 f(x)dx$. The precision is controlled by a parameter N that sets how many trapezes each thread computes.

   Example: for $T = 4$ threads, each thread computes an equally large section of the integral:

   - Thread 0 computes: $\int_0^{\frac{1}{4}} f(x)dx$.
   - Thread 1 computes: $\int_{\frac{1}{4}}^{\frac{2}{4}} f(x)dx$.
   - Thread 2 computes: $\int_{\frac{2}{4}}^{\frac{3}{4}} f(x)dx$.
   - Thread 3 computes: $\int_{\frac{3}{4}}^{1} f(x)dx$.

   Assume that the program has been implemented well: no major performance bugs exist.

   You are executing this program on a computer that gives this output when you run the `lscpu` command (some details omitted):

```
$ lscpu
Architecture:          x86_64
CPU(s):                4
Thread(s) per core:    1
Core(s) per socket:    4
Socket(s):             1
CPU MHz:               1600.000
CPU max MHz:           2267.0000
CPU min MHz:           1600.0000
L1d cache:             32K
L1i cache:             32K
L2 cache:              256K
L3 cache:              8192K
```

   (a) What scaling behaviour (not measuring setup-times) would you expect to see when increasing the number of threads?

   (b) What maximum speed up would you expect, and why?

5. **OpenMP (2 + 2 + 3 + 2 = 9 pts total).** Count sort is a simple serial sorting algorithm that can be implemented as follows:

```
void count_sort(int a[], int n) {
  int i, j, count;
  int* temp = malloc(n*sizeof(int));
  for (i = 0; i < n; i++) {
    count = 0;
    for (j = 0; j < n; j++)
      if (a[j] < a[i])
        count++;
      else if (a[j] == a[i] && j < i)
            count++;
    temp[count] = a[i];
  }
  memcpy(a, temp, n*sizeof(int));
  free(temp);
} /* count_sort */
```

The basic idea is that for each element `a[i]` in the array, we count the number of elements in the array that are less than `a[i]`. Then we insert `a[i]` into a temporary array using the subscript determined by the count. There's a slight problem with this approach when the array contains equal elements, since they could get assigned to the same slot in the temporary array. The code deals with this by incrementing the count for equal elements on the basis of the subscripts. If both `a[i] == a[j]` and `j < i`, then we count `a[j]` as being "less than" `a[i]`.

After the algorithm has completed, we overwrite the original array with the temporary array using the string library function `memcpy`.

(a) If we try to parallelize the `for i` loop (the outer loop), which variables should be private and which should be shared? Justify your answer.

(b) If we parallelize the `for i` loop using the scoping you specified in the previous part, are there any loop-carried dependences? Explain your answer.

(c) Can we parallelize the call to `memcpy`? Can we modify the code so that this part of the function will be parallelizable? Explain your answers.

(d) Suppose that you wrote a parallel C program implementing Count sort using OpenMP. What do you think would be the performance of your parallelization compared to the serial version of the same program and why?

---

6. **MPI Programming (7 pts).** Write an MPI program that implements multiplication of a vector by a scalar and dot product. The user should enter two vectors and a scalar, all of which are read in by process 0 and distributed among the processes. The results are calculated and collected onto process 0, which prints them. You can assume that n, the order of the vectors, is evenly divisible by `comm_sz`.

To get all points for this question, you need to submit a working program in a file, together with instructions on how to compile it and run it. For partial credit, submit either a program skeleton or code on paper.

Good luck !