Programming for Performance (ECE459): Midterm

February 17, 2010

This open-book midterm has 5 questions, worth 20 points each. Choose any 4 of these questions, for a total of 80 points. If you answer 5 questions, we'll choose which 4 to mark. Answer the questions in your answer book. You may consult any printed material (books, notes, etc).

Question 1: Parallelization Patterns (20 points)

(5 points each) For each of the following situations, **name an appropriate parallelization pattern and the granularity at which you would apply it, explain the necessary communication, and explain why your pattern is appropriate**. (About 5 lines each.)

- build system, e.g. parallel make;
- optical character recognition system.

(5 points each) **Give a concrete example** where you would use the following parallelization patterns. **Explain** the granularity at which you'd apply the pattern.

- single task, multiple threads;
- producer-consumer (no rendering frames, please).

Question 2: Speculation (20 points)

We saw this code:

```
void doWork(int x, int y) {
  int value = longCalculation(x, y);
  return secondLongCalculation(value);
}
```

The notes included one way to parallelize this code using speculation. (5 points) **State the conditions on `longCalculation` and `secondLongCalculation` under which it is safe to parallelize these calls.** To clarify, are there any things that the implementations of `longCalculation` and `secondLongCalculation` should not do, if you're going to parallelize

them? (5 points) **Write down a more sophisticated speculation for this method.** What I have in mind here is terminating `secondLongCalculation` early. State any additional assumptions that the calculations must respect for your new speculation to be safe. You may use reasonable threading primitives. (10 points) **Estimate the running time of your speculation.** Write down a symbolic formula which summarizes the running time, including any probabilities that you need.

Question 3: Interpreting Profiling Data (20 points)

I ran `oprofile` on `jpegr` on my computer to generate the following data.

```
CPU: Core 2, speed 2800.03 MHz (estimated)
Counted CPU_CLK_UNHALTED events (Clock cycles when not halted) with a
  unit mask of 0x00 (Unhalted core cycles) count 100000
samples  %        image name               symbol name
105468   87.3564  jpegr                    op_resize_work
15166    12.5616  jpegr                    do_rot_90
24        0.0199  jpegr                    read_jpeg
23        0.0191  jpegr                    jpeg_read
1        8.3e-04  jpegr                    do_transform
1        8.3e-04  jpegr                    ece459_scale_thumbnail
1        8.3e-04  jpegr                    jcopy_markers_execute
1        8.3e-04  jpegr                    jpeg_transform_fp
```

Let's assume that `op_resize_work` and `do_rot_90` are independent (i.e. neither one calls the other). Let's also assume that we can perfectly parallelize `op_resize_work` but not `do_rot_90`. You can round the numbers to 87.5 and 12.5 to make the arithmetic easier. **Write down a formula** for `jpegr`'s runtime as a function of the number of processors. **Estimate** the number of processors you can profitably use if you are willing to accept a runtime within 10% of the perfect runtime.

Now, let's change the assumptions a bit. Let's assume that the non-parallelizable part of the program takes 12.5s, and that you can run as many inputs as you want in 87.5 seconds, as long as you have enough processors. **Calculate the throughput** that you can get if you have 100 processors available.

Question 4: Race Conditions (20 points)

Here's a C linked list implementation.

```
struct node {
  int elem; int size;
  struct node * next, * prev;
};
```

```
/* list has a header */
/* assume that add (not shown) updates size appropriately */
struct node * findAndDelete (struct node * list, int target) {
  struct node * n = list->next, * np = NULL;
  int i = 0;
  for (i = 0; i < size; i++) {
    if (n->elem == target) {
      np->next = n->next;
      n->next = NULL;
      return n;
    }
    np = n; n = n->next;
  }
}

void clear (struct node * list) {
  list.next = list.prev = list;
  list.size = 0;
}
```

I mentioned in passing that race conditions can lead to infinite loops.

(5 points) **Show me a simple race condition that causes incorrect output**: draw a linked list which is shared between two threads, and write down an interleaving of statements in `findAndDelete` which causes one of the threads to get stuck in an infinite loop, along with enough information for me to understand what the statements are doing.

(5 points) **Propose a fix.** I'm not looking for compilable syntax; you can verbally describe any constructs you use. Try to be as specific as possible. Your solution should be free of race conditions when multiple threads access the list in parallel.

(10 points) **Find an infinite loop caused by a race condition.** This loop should only include a finite number of calls to the functions I've listed above. Include the same information as the first part above.

Question 5: Dependences and parallelization (20 points)

Consider this JPEG manipulating code from `transupp.c` (part of `libjpeg`).

```
1 LOCAL(void)
2 transpose_critical_parameters (j_compress_ptr dstinfo)
3 {
4   int tblno, i, j, ci, itemp;
```

```
5    jpeg_component_info *compptr;
6    JQUANT_TBL *qtblptr;
7    JDIMENSION jtemp;
8    UINT16 qtemp;
9
10   /* Transpose image dimensions */
11   jtemp = dstinfo->image_width;
12   dstinfo->image_width = dstinfo->image_height;
13   dstinfo->image_height = jtemp;
14   itemp = dstinfo->min_DCT_h_scaled_size;
15   dstinfo->min_DCT_h_scaled_size = dstinfo->min_DCT_v_scaled_size;
16   dstinfo->min_DCT_v_scaled_size = itemp;
17
18   /* Transpose sampling factors */
19   for (ci = 0; ci < dstinfo->num_components; ci++) {
20     compptr = dstinfo->comp_info + ci;
21     itemp = compptr->h_samp_factor;
22     compptr->h_samp_factor = compptr->v_samp_factor;
23     compptr->v_samp_factor = itemp;
24   }
25
26   /* Transpose quantization tables */
27   for (tblno = 0; tblno < NUM_QUANT_TBLS; tblno++) {
28     qtblptr = dstinfo->quant_tbl_ptrs[tblno];
29     if (qtblptr != NULL) {
30       for (i = 0; i < DCTSIZE; i++) {
31         for (j = 0; j < i; j++) {
32           qtemp = qtblptr->quantval[i*DCTSIZE+j];
33           qtblptr->quantval[i*DCTSIZE+j] =
34             qtblptr->quantval[j*DCTSIZE+i];
35           qtblptr->quantval[j*DCTSIZE+i] = qtemp;
36         }
37       }
38     }
39   }
40 }
```

(10 points) **List** 5 dependencies (loop-carried or memory-carried) in the above code. Make sure that at least one of the dependencies are not RAW dependencies. For each statement with dependencies—whether in a loop or not—you should list the statements which it depends on. Free example: line 13 is a RAW memory-carried dependence on line 11.

(10 points) **For each loop in the above code, write down whether it could potentially be automatically parallelized, and explain why.** Assume that you don't want to carry out any transformations in the structure of the code, but that you will add qualifiers and copy variables.