Programing for Performance (ECE 459): Midterm solution

Xavier, Jon, and Hany; edits by Patrick

February 17, 2010

1 Question 1: Parallelization Patterns

1.1 Naming Parallelization patterns (10 points)

- Build system: **Multiple independent tasks, at a per-file granularity**.

  Given a project to build, the build system can compute sets of independent files and compile each set in parallel. The compilation of each independent set would proceed sequentially.

  After each set compiles, the build system might need to combine results from the sets. This requires communication: the compiler needs to communicate success/failure to the build system after each compilation, and the compiler output files need to go to the linker.

- Optical character recognition system: **Pipeline of tasks**.

  We are assuming that we have a bitmap and we want to convert it into characters. The first task is segmentation into lines and characters, while the second task is classification of each character. This can happen in parallel for the different characters.

1.2 Concrete examples (10 points)

- Single task, multiple threads (fork-join): **Computation of a mathematical function with independent sub-formulas**.

  Each sub-formula may be computed by a different thread. At the end of all sub-formula computations, it may be necessary to combine the sub-results sequentially. Example: $f(x) = h(x) + g(x)$.

- Producer-consumer: **Processing of stock-market data**.

  A server might generate raw financial data (quotes) for a particular security. The server would be the producer. Several clients (or consumers) may take the raw data and use them in different ways. For instance by generating averages, means, charts, etc.

2 Question 2: Speculation

2.1 Conditions under which it is safe to parallelize the calls (5 points)

- `longCalculation` and `secondLongCalculation` must not call each other.
- The implementation of `secondLongCalculation` must not depend on any values that may be set or modified by `longCalculation`.
- The return value of `longCalculation` must be deterministic.

2.2 A more sophisticated speculation (10 points)

```
void doWork(int x, int y) {
  int value1, result;
  static int exp_value; //expected return value of longCalculation

  #pragma start parallel region
  {
    #pragma perform parallel task1
    {
      value1 = longCalculation(x, y);
      if (value1 != exp_value) {
        #pragma kill task2
        result = secondLongCalculation(value1);
      }
    }

    #pragma perform parallel task2
    {
      result = secondLongCalculation(exp_value);
    }

    #pragma wait for parallel tasks to complete

    return result;
  }

}
```

2.3 Symbolic estimation of the running time of the speculation (5 points)

Given the following parameters:

- T: running time of the speculation
- T_1: running time of task 1
- T_2: running time of task 2
- T_c: running time of `longCalculation`
- T_d: running time of `secondLongCalculation`

- P_e: probability that the expected value calculation is correct.
- T_s: synchronization overhead

The symbolic formula representing the running time of the speculation is:

$$T = max(T_1, T_2) + T_s$$

with:

$$T_1 = T_c + (1 - P_e) \times T_d$$
$$T_2 = P_e \times T_d + (1 - P_e) \times T_c$$

3 Question 3: Interpreting Profiling Data (20 points)

(7 points) **Write down a formula** for `jpegr`'s runtime as a function of the number of processors.

The formula for `jpegr`'s runtime as given by **Amdahl's Law** is:

$$T_{jpegr} = 12.5 + \frac{87.5}{N} \quad (1)$$

where N is the number of processors.

(7 points) **Estimate** the number of processors you can profitably use if you are willing to accept a runtime within 10% of the perfect runtime.

We want the parallelized code to run in 1.25 s. This is 10% of the serial code time, or perfect runtime (since $\frac{P}{N} = 0$).

$$\begin{aligned} \frac{87.5}{N} &= 1.25 \\ N &= 70 \end{aligned}$$

Therefore, we need **70 processors** to get a runtime within 10% of the perfect runtime.

(6 points) **Calculate the throughput** that you can get if you have 100 processors available.

There are two interpretations for this question. The simplest is that we are only one instance of the program at a time; in which case the answer is **1 image every 13.375 seconds** using equation 1. Another is that you are able to run separate instances of the program to process as many images as you can in the 87.5 second time frame. We have to do the serial calculation first which takes 12.5 s, leaving 75 s where we can finish as many parallel jobs as possible. Below is a calculation for the number of tasks we can handle:

$$\begin{aligned} tasks\left(\frac{87.5}{100}\right) &= 75 \\ tasks &= 85.7 \\ &= 85 \end{aligned}$$

We have to round down since the number of tasks we can complete is discrete. Therefore we can process **85 images in 87.5 seconds** or a throughput of 0.97 images/second.

4 Question 4: Race conditions (20 points)

(15 points) **Show me a simple race condition that causes incorrect output**

Let's ignore the prev data for the list and just consider next, since that's all findAndDelete changes. Consider the list: 1 (2), 2 (3), 3 (4), 4 (NULL) where the brackets indicate the value of the next node. We have 2 function calls to findAndDelete in separate threads, one has 2 has the target, the other has 3 and both threads are in the if statement.

Thread 1: `np->value` = 1, `n->value` = 2

Thread 2: `np->value` = 2, `n->value` = 3

Thread 1 executes `np->next` = `n->next` giving us: 1 (3), 2 (3), 3 (4), 4 (NULL).

Thread 2 executes `np->next` = `n->next` giving us: 1 (3), 2 (4), 3 (4), 4 (NULL).

Then both threads execute `n->next` = `NULL`, in this case the order doesn't matter, giving us 1 (3), 2 (NULL), 3 (NULL), 4 (NULL). If we iterate through the list we'll get 1, 3 as the result instead of the expected 1, 4.

(5 points) **Propose a fix.**

The simplest is just to have a mutex around the body of findAndDelete, so only one thread can access and change the linked list at a time.

(5 bonus points) **Find an infinite loop caused by a race condition.**

There is no way to cause an infinite loop since the code only loops `list->size` times whenever the findAndDelete function is called.

5 Question 5: Dependences and Parallelization (20 points)

5.1 Five dependencies

All dependencies in the code are memory-carried. There are no loop-carried dependencies. A list of some of the existing dependencies:

1. Line 12 is a memory-carried WAR dependence on line 11.
2. Line 13 is a memory-carried WAR dependence on line 12.
3. Line 15 is a memory-carried WAR dependence on line 14.
4. Line 16 is a memory-carried WAR dependence on line 15.
5. Line 16 is a memory-carried RAW dependence on line 14.
6. Line 19 has a memory-carried RAW dependence (Initialization of ci & conditional check on ci)
7. Line 19 has a memory-carried RAW dependence (Increment of ci & conditional check on ci)
8. Line 20 is a memory-carried RAW dependence on line 19.
9. Line 21 is a memory-carried RAW dependence on line 20.
10. Line 22 is a memory-carried WAR dependence on line 21.
11. Line 23 is a memory-carried WAR dependence on line 22.

12. Line 23 is a memory-carried RAW dependence on line 21.
13. Line 27 has a memory-carried RAW dependence (Initialization of tblno & conditional check on tblno).
14. Line 27 has a memory-carried RAW dependence (Increment of tblno & conditional check on tblno).
15. Line 30 has a memory-carried RAW dependence (Initialization of i & conditional check on i).
16. Line 30 has a memory-carried RAW dependence (Increment of i & conditional check on i).
17. Line 31 has a memory-carried RAW dependence (Initialization of j & conditional check on j).
18. Line 31 has a memory-carried RAW dependence (Increment of j & conditional check on j).
19. Line 28 is a memory-carried RAW dependence on line 27.
20. Line 29 is a memory-carried RAW dependence on line 28.
21. Line 33 is a memory-carried WAR dependence on line 32.
22. Line 35 is a memory-carried WAR dependence on line 33.
23. Line 35 is a memory-carried RAW dependence on line 32.

5.2 Automatic Parallelization

1. Loop at line 19
   (a) Potential candidate for parallelization
   (b) Loop bound "dstinfo–>num_components" needs to be loop invariant. Assuming that it does not alias anything else, it should be qualified using restrict.
   (c) Pointer "compptr' should be qualified using the restrict qualifier.
   (d) Iterations has no dependencies as the loop accesses a linear array and swaps fields.
2. Loop at line 27
   (a) Potential candidate for parallelization
   (b) Constant loop bound (following C conventions for constants)
   (c) "qtblptr" should be restrict qualified
   (d) Linear array being accessed.
3. Loop at line 30
   (a) Potential candidate for parallelization
   (b) Constant loop bound (following C conventions for constants)
4. Loop at line 31
   (a) Potential candidate for parallelization
   (b) Loop bound is not changing, it is loop invariant.
   (c) Unlikely to be profitable while parallelizing the outer loops.
   (d) We need to restrict qualify or copy the pointer to "qtblptr–>quantval"
   (e) Although $i * DCTSIZE + j$ and $j * DCTSIZE + i$ are being used, the same array index is never repeated twice during execution of the loops. We need to ensure that indices never interfere.
   Ex. $DCTSIZE = 10$
   i=0, inner loop does not execute
   i=1, swaps (10,01)
   i=2, swaps (20,02) and (21,12)
   i=3, swaps (30,03), (31,13) and (32,23)