ECE498SL: Engineering Software Systems

Final Examination

11 May 2011

- You may use your brain and assorted writing implements.
- You may not use books, notes, calculators, other people's brains, etc.
- You may have two 8.5×11-inch pages (double-sided) of handwritten notes.
- This exam has 8 pages, including this one.

" ' … by Professor Timothy Dwight: "The happiest person is the person who thinks the most interesting thoughts." ' Promptly one starts recalling such Happiness Boys as Nietzsche, Socrates, de Maupassant, Jean-Jacques Rousseau, William Blake, and Poe."

—from Dorothy Parker's review of the book *Happiness*

Most of the problems require a short response. If you choose to write a long response, say more than about 25 words, I'll probably stop reading after the short part at the front. Avoid pronouns: it makes it hard to know what it means.

Name: _____

A: As you probably know, most desktop software companies compile their code for specific ISAs (such as x86) before selling it to the end consumer. In contrast, graphics processing algorithms (such as shaders) are typically compiled from an intermediate form into the GPU's ISA just before they are executed.

Answer the following two questions in the context of software development for the current and future generations of parallel desktop CPU architectures.

A.1 (5 points): Explain one advantage of delaying the final stage of compilation.

A.2 (5 points): Explain one disadvantage of delaying the final stage of compilation.

B. As you may recall, the bulk-synchronous paradigm (BSP, where P has many possible meanings) is an approach in which all threads in a program periodically stop and wait for one another (typically using barriers).

B.1 (5 points): Explain one benefit of using a BSP approach when writing large parallel programs.

B.2 (5 points): Explain one drawback of using a BSP approach. *Hint: one such drawback is likely to be more important on multi-core chips than on clusters.*

C (5 points): As part of Homework #2, you were asked to make use of a function that printed the contents of an STL (Standard Template Library) queue by popping all of the elements off and then pushing them back on. The lack of an interface for inspecting intermediate elements of the queue made this approach necessary.

Consider an extension to the STL queue abstraction that enables use of a queue with multiple threads. The **`push`** and **`pop`** operations (as well as others) are synchronized with respect to themselves and to one another through mechanisms private to the parallel queue abstraction.

Now the question: can you write a function that prints the contents of a parallel queue? Explain the significance of the inheritance anomaly to your answer.

D (5 points): *Parallel speedup* is the ratio of parallel execution time to sequential execution time, and *parallel efficiency* is the ratio of parallel speedup to the number of processors. Under what circumstances might observe parallel efficiency above 1?

E (5 points): One way in which we can divide models of parallelism is by whether or not arbitrary threads can access data asynchronously with respect to one another. Message-passing models arose on distributed memory machines, in which processors cannot directly address all of the memory, but must instead request access to remote memory from other processors (using messages).

When re-optimized for execution on a shared memory machine, message-passing runtimes could make use of the hardware support for coherent sharing of memory. However, most of them do not. Explain this choice in terms of the debugging advantages, and particularly with respect to the difficulty of data race detection in buggy code using a model that allows asynchronous access (such as a single address space) compared with one allowing only synchronous accesses (such as multiple address spaces).

F: Consider the template-based, non-blocking implementation of a linked list on the following page. The code uses compare-and-swap (assembly code not shown) to perform insertions (the **`insert`** method) and removals (the **`popFirst`** method) from the front of the list atomically with respect to other insertions and removals. For simplicity, Assume that the compiler does not reorder nor eliminate memory operations and that the ISA supports sequential consistency.

F.1 (5 points): Explain why the **`Epoch`** template class is necessary.

F.2 (5 points): Explain the meaning of "non-blocking" and how the algorithms used to insert and remove elements from the list provide this (non-blocking) behavior.

F.3 (5 points): Although the implementation is technically non-blocking, the design is flawed. Consider the following scenario: a programmer tries to use a linked list (as implemented here) to track tasks. One thread produces tasks and inserts them into the list, while others remove tasks and perform the work. Explain how this system might fail to operate as desired despite the non-blocking nature of the implementation.

F.4 (5 points): Suggest a simple way to fix the implementation so as to prevent the failure discussed in **Part F.3**.

```
template<class T> class Epoch {
        T*      ptr;
        int32_t epoch;
    public:
        Epoch () : ptr (NULL), epoch (0) {}
        T* getPtr () { return ptr; }

        // try to compare-and-swap pointer value and bump up epoch
        // number atomically; returns true on success
        bool CAS (const Epoch<T>& oldEpoch, T* newPtr);
};

template<class T> class LinkedList {

        class ListElt {      // a dynamically-allocated list element
                ListElt* next;
                T        data;
            public:
                ListElt (const T& obj) : next (NULL), data (obj) {}
            friend class LinkedList<T>;
        };

        Epoch<ListElt> head; // head of list

    public:

        bool insert (const T& obj) {    // returns true on success
            ListElt*       elt = new ListElt (obj);
            Epoch<ListElt> oldEpoch;
            do {
                oldEpoch  = head;
                elt->next = oldEpoch.getPtr ();
            } while (!head.CAS (oldEpoch, elt));
            return true;
        }

        bool popFirst (T* objPtr) {    // returns true on success
            Epoch<ListElt> oldEpoch;
            ListElt*       elt;
            ListElt*       nextVal;
            do {
                oldEpoch  = head;
                elt       = oldEpoch.getPtr ();
                if (NULL != elt) {
                    *objPtr   = elt->data;
                    nextVal   = elt->next;
                } else {
                    nextVal   = NULL;
                }
            } while (!head.CAS (oldEpoch, nextVal));
            if (NULL != elt) {
                delete elt;
                return true;
            }
            return false;
        }
};
```

G: After hearing about your experience parallelizing connected components in Lab #3, your new supervisor decides to give you the job of parallelizing another graph-oriented code. As shown below, the program starts with a large array of graphs, each of which is marked as being of a certain type (the number of types is small relative to the number of graphs). You are to find a way to parallelize the **`merge_all_graphs`** function, which calculates, for each graph type, a summary of all graphs of that type in the array.

```
struct typed_graph_t {
    int32_t type; // from 0 to NUM_TYPES - 1
    Graph*  g;
};

// these variables are initialized before the code below executes
static typed_graph_t   graphs[NUM_GRAPHS];
static graph_summary_t summary[NUM_TYPES];

void merge_all_graphs ()
{
    int32_t i;

    for (i = 0; NUM_GRAPHS > i; i++) {
        graph_merge (&summary[graphs[i].type], graphs[i].g);
    }
}
```

You know the following about the **`graph_merge`** function:

- It takes a long time to run on a graph (say O(10 seconds)).
- It neither reads nor writes anything except its arguments (a summary and a graph) and private variables.

G.1 (5 points): Without making other assumptions about the properties of **`graph_merge`**, how can you parallelize the function?

G.2 (5 points): What is the best parallel speedup that you can hope to achieve in this way, assuming infinite processing resources?

G.3 (5 points): Load imbalance will have a direct impact on parallel speedup after you have parallelized the code. Explain this relationship and why you cannot address it without more information about **`graph_merge`**.

H (5 points): An parallel data abstraction is *serializable* if any set of operations that can be executed on an instance produces the same return values and internal state for the instance as some serial ordering of the operations. The operations executed by any single thread must also be the same in the serial ordering as in the thread's execution order. Thus, if only a single thread uses an instance of the parallel abstraction, the semantics are identical to what one expects with a sequential version of the abstraction.

Explain how *linearizability* differs from serializability, preferably with an abstract example of behavior that is serializable but not linearizable.

I (5 points): Explain why a ticket/bakery lock might be considered to be more fair than a lock based on test-and-set.

J (5 points): What was the most interesting thing that you found in parallelizing image segmentation for Lab #3? *Hint: these should be free points!*

K (5 points): One of David Bailey's ways to fool the masses is to…

> *"Scale up the problem size with the number of processors, but omit any mention of this fact."*

Explain why such an approach might mislead readers in terms of the parallel speedup achieved by the results being reported.

L: A friend of yours asks your advice on parallelizing a medical imaging application. Given a set of 3D image data—about ten images consisting of 200^3 points—the application identifies "features" using a fairly complex analysis method that examines a 2D neighborhood (X and Y dimensions) of 20^2 points around any given point. Starting from a sequential version, the friend must decide between parallelizing across images and parallelizing across smaller chunks of work, such as 2D image planes or even individual points in the image.

Your friend understands the benefit of handling each image separately: the input data set is an image, and the application copies can simply execute independently, requiring effectively no work on the part of your friend.

L.1 (5 points): Assuming that your friend only has access to four dual-core processors, explain one advantage of parallelizing over 2D image planes relative to parallelizing over full images.

L.2 (5 points): Explain one advantage of parallelizing over individual points in the images relative to parallelizing over 2D image planes.