A Breadth-First Course in Multicore and Manycore Programming

Suzanne Rivoire

Sonoma State University

March 12, 2010

Parallelism is everywhere

- *Multicore:* Scaling processor performance by increasing the number of cores/chip
- *Distributed/cloud/grid computing:* Applications/ computations that scale to large numbers of machines

“The free lunch is over.” [Sutter, 2005] – performance now requires harnessing parallelism

When to introduce parallelism?

- In the OS course
  - Traditional approach
  - Has been used to teach new multicore programming models [Rossbach, PPoPP ’10]

- Throughout undergraduate curriculum [Ernst, ITiCSE ’08]

- *Upper-level undergraduate elective*

How to introduce parallelism?

- Lots of parallel programming APIs/models, with new ones emerging all the time
- Typical parallel programming elective is a *graduate* course focusing on a particular (trendy, new) model
- For undergraduates, we tried *breadth-first*
  - Avoid committing to a particular model
  - Emphasize commonalities and underlying algorithms

Outline

- Breadth-first course overview
  - Goals
  - Organization
  - Structure
- Course content
- Evaluation

Course information

- *Title:* CS 385: Multicore and Manycore Programming [elective]
- *University:* Sonoma State University
- *Semester:* Spring 2009
- *Prereqs*: CS2, introductory computer organization
- *Enrollment:* 18 students, all undergraduate

By the end of this course, you will…

- Think parallel! Find task- and data-parallel decompositions
- Analyze the performance of your code and the barriers to scalability
- Understand developments in parallel hardware and software
- Be better programmers in general

Course organization

Weeks	Subject
1-2	Crash course in parallel decomposition, computer architecture, and performance analysis
3-6	OpenMP
6-9	Intel TBB
10-14	nVidia CUDA
14-16	Readings on other programming models

Why these models?

- Accessible to C/C++ programmers
- Well supported, mature (enough) infrastructure
- CPU- and GPU-based
- Different levels of abstraction

Course activities and assessments

- Lecture/discussion: 2 hours/week
- Lab activities: 2 hours/week supervised + some independent work
- Projects: Optimizing matrix multiplication in each model + 1 writing project
- Quizzes: 4 quizzes (one for each programming model + the paper-reading)
- Comprehensive final exam

Outline

- Breadth-first course overview
- Course content
  - Initial overview
  - OpenMP, TBB, CUDA details
  - Reading papers on other models
- Evaluation

Module 1: Overview of basics

- Lecture topics
  - Overview of multicore challenges (View from Berkeley)
  - Parallel decomposition; task parallelism; data parallelism
  - Performance analysis: speedup, scalability
  - Memory hierarchy, cache coherence, synchronization

Module 1: Overview of basics

- Sample activities/assignments
  - Parallelize this recipe!
  - Practice mapping computations to threads by “parallelizing” two embarrassingly data-parallel algorithms

Module 2: OpenMP

- OpenMP background
  - Simple API for shared-memory programming
  - Established and widely supported (1998-)
  - Support for data and (some) task parallelism
- Assignments
  - Parallelize and tune code from Module 1
  - Implement a data-parallel algorithm with significant dependencies

OpenMP sample code

```
#pragma omp parallel for
for (i = 0; i < N; i++)
  a[i] = b[i] + 1;
```

Module 3: TBB

- TBB background
  - Introduced by Intel in 2006
  - C++ template library (very STL-like)
  - High-level; hides implementation details
- Assignments
  - Port previous assignments to TBB
  - Use TBB's concurrent container classes

TBB sample code

```
class some_class {
   …
   void operator()(const blocked_range
   &range) const {
   for (int i = range.begin();
           i!= range.end(); i++)
      A[i] = B[i] + 1;
   }
};
parallel_for(blocked_range(0,N),
   some_class(A, B), auto_partitioner());
```

Module 4: CUDA

- CUDA background
  - Introduced by nVidia in 2007 for general-purpose GPU programming
  - Requires programmer to manage movement of data between CPU and GPU
  - Requires programmer to map computations to threads and thread blocks on GPU

Sample CUDA code

```
kernel<<< gridDim, blockDim, 0 >>>(A, B);

__global__ void kernel(float* A, float* B) {
   unsigned int tid = blockIdx.x*blockDim.x +
   threadIdx.x;
   A[tid] = B[tid]+1;
}
```

Module 5: Other models

- Papers read
  - *GPGPU:* Owens et al, *Proc. IEEE* 5/2008.
  - *MapReduce:* Dean et al, OSDI 2004.
  - *Transactional memory:* Adl-Tabatabai et al, *Queue* 12/06.
- Method: just-in-time teaching + discussion
  - Students submit writeups shortly before class
  - Their answers drive the day's discussion [Davis, SIGCSE '09]

Sample just-in-time assignment

- *Reading guide:* roadmap of paper
  - "Read the section on programming with transactions. Understand the programming examples and the graph. This section describes the guarantees that a TM system makes to the programmer, and the benefits to correctness and performance."

Sample writeup questions

- High-level comprehension:
  - Explain in your own words why versioning is needed and the difference between eager and lazy versioning.
- Low-level:
  - Explain Figure 2: what does it show, and why does that result occur?

Project: Tutorial on one model

- Explain when the model is important/useful
- Guide the reader through a simple example with some performance tuning/analysis

Outline

- Breadth-first course overview
- Course content
- Evaluation
  - Learning objectives?
  - Course structure and assignments?
  - Future changes

Evaluation instruments

- End-of-semester survey and evaluations
- Reflections in student project reports (throughout semester)
- Choice of model for final project

Good things

- Lab assignments
  - Rated as most helpful component of course for 3 of the 4 learning outcomes
- CUDA
  - By far the most popular model
  - GPU “cool” factor? Bigger speedups? Low-level control?
  - Students preferred programming models in order from low- to high-level
- Discussions of papers

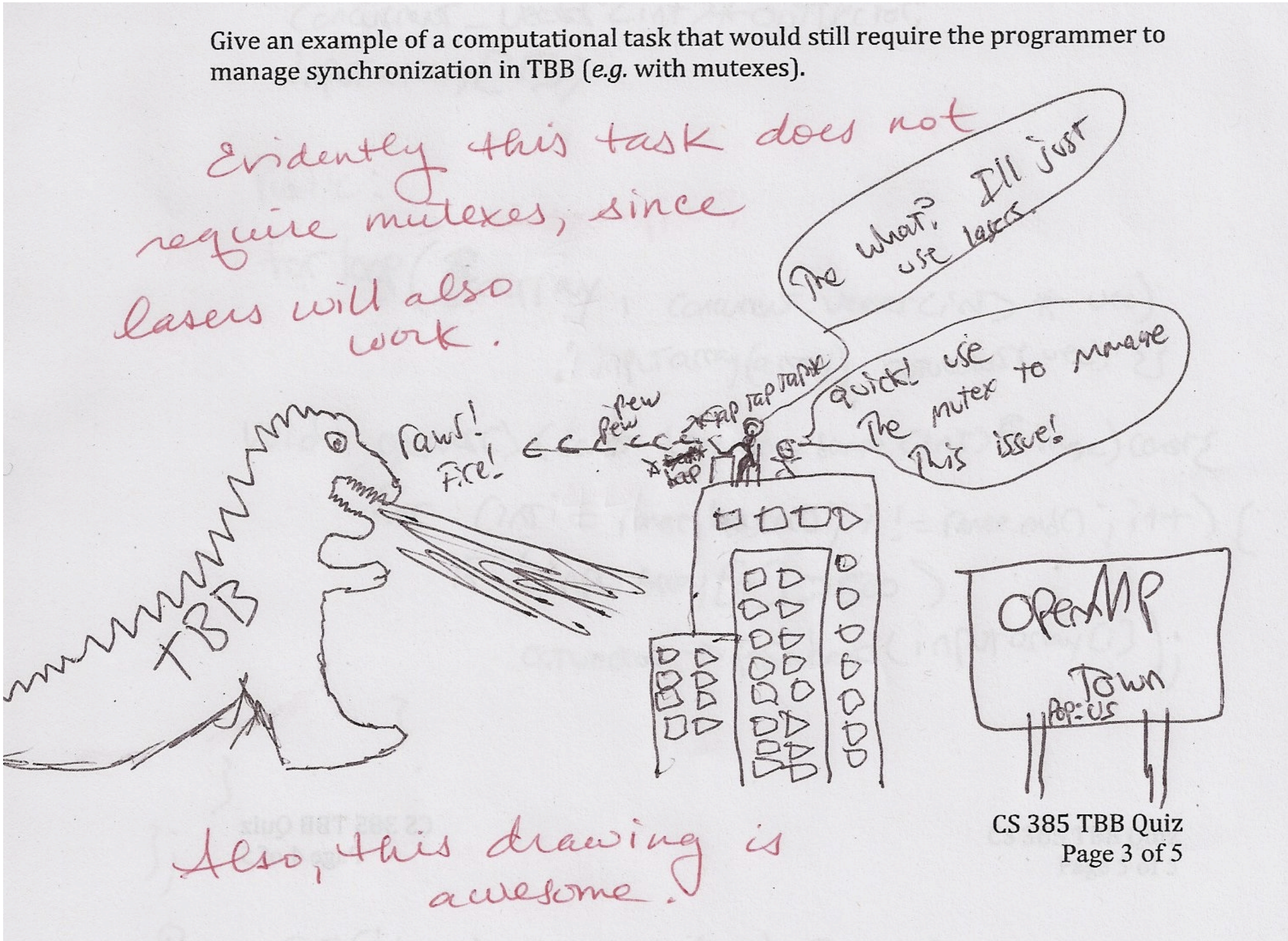
Bad things

- Projects
  - Limited shared hardware => flawed speedup results and a lot of frustration
  - Matrix multiplication: too staid a problem?
- TBB?
  - Least popular model…hard to understand performance, too much bureaucracy
  - Worthwhile challenge to students?
  - Valuable comparison point?

Open Questions

- Breadth-first?
  - Class was evenly split between liking the course as-is and wanting slightly more depth
  - No one wanted pure depth-first (studying only one model)
- Choice of programming models?
  - New models, new infrastructure emerging
  - Would keep GPU model (CUDA? OpenCL?) but the rest is up for debate

http://rivoire.cs.sonoma.edu/cs385/



Give an example of a computational task that would still require the programmer to manage synchronization in TBB (*e.g.* with mutexes).

Evidently this task does not require mutexes, since lasers will also work.

Also, this drawing is awesome.