Some resources for teaching concurrency *

Ganesh Gopalakrishnan, Yu Yang, Sarvani Vakkalanka, Anh Vo, Sriram Ananthakrishnan, Grzegorz SzuBzda, Geof Sawaya, Jason Williams, Subodh Sharma, Michael DeLisi, Simone Atzeni

School of Computing, Univ. of Utah, Salt Lake City, UT 84112, USA

Abstract. We provide a few handy teaching resources for concurrency. We present an overview of a class (in progress) on multi-core computing with emphasis on concurrency. We detail some exercises designed to teach memory consistency models, and allied topics. We then detail exercises for teaching threading, with tool support offered by dynamic verification tools Inspect (Utah) and CHESS (MSR). Last, we detail exercises for teaching message passing using the dynamic verification tool ISP (Utah). We conclude with a summary of pedagogical material being assembled, including many exercises from a popular textbook on MPI solved using ISP. Today’s textbooks, while good at covering conceptual topics and performance issues, do not integrate the use of formal verification tools – something we have begun developing pedagogical material for. In summary, teaching concurrency – especially to undergraduates and especially to prepare them for the upcoming multi-core revolution – is greatly facilitated by the use of dynamic verification tools, and by taking a hands-on approach before delving into the theory.

1 Introduction

There is growing interest in teaching concurrency, given that multi-core systems are in the forefront. The first author is at present teaching such a class [22] with help offered by the co-authors. This paper reports our experience to date in presenting various topics. We provide experience reports of using three tools, two of which were created in our group. The tools used were: Inspect [44–46] and CHESS [32, 33] (for threading), ISP [24, 27–31].

Designing any course is a balancing act. One balances the size and background of the audience against the time one is willing to commit to class preparation, and the tools one has access/control over. This paper presents our syllabus as well as our reasons for preferring a syllabus. In this paper, we also include material drawn from independent projects that were conducted as offshoots of our class. When these projects finish (in a few weeks), we will have additional case studies to offer. Our key insights so far are now summarized.

Use of Dynamic Verification Tools: It is quite advantageous to teach concurrency using formal dynamic verification tools [39] such as Inspect, CHESS, and ISP. Students who do not have a formal methods background are drawn by the debugger-like user interface provided by these tools. Using this approach,

* Supported in part by Microsoft, NSF CNS-0509379, CCF-0811429, and SRC
one can demonstrate the possibility of bug omissions using traditional debuggers, and the success of finding these bugs through formal dynamic verifiers that provide coverage guarantees (e.g., [23]). Students are also fascinated by the interleaving reductions offered by dynamic partial order reduction (DPOR) [38], which makes tools such as Inspect efficient, and applicable to thread programs of a few thousand lines. As far as our verifier ISP goes, it includes an MPI-specific version of DPOR called POE [27]. This algorithm is crucial to the success of ISP, and students are able to appreciate the need for such interleaving reductions.

Note: In our effort so far, we have not explored the use of the Java Pathfinder [34] tool. We would like to pursue this in future.

**Use of Model-based Verification Tools:** In previous courses, we have employed SPIN [41] to introduce concurrency. A similar tool to use for teaching message passing based on the Message Passing Interface (MPI) [47] API would be [42]. For our purposes, we preferred dynamic verification tools, as they allow direct execution of user-written codes. The ability to ask a student to natively execute a concurrent program and then push-button model-check the very same concurrent program has been a huge plus. Students readily accept the notion of formal dynamic verification as testing with accompanying formal guarantees.

Currently, there are many efforts in furthering dynamic verification (e.g. [43, 35]). However, we believe that many more such efforts are badly needed. Given that (i) debugging concurrent software is very expensive in practice [40], (ii) we must convince real designers to begin using verification tools on their source codes, and (iii) concurrent software of the future will most likely be written using languages that call external libraries and APIs, a whole range of dynamic formal verification techniques must be explored. Last but not least, while we did not include static analysis tools in our classes, any static analysis that produces little or no false alarms would also be welcome additions to the concurrency tool suite, as also would be static analysis techniques that help make dynamic analysis more efficient.

**Choice of Textbooks/papers:** While there are many excellent textbooks that can help teach concurrency with particular relevance to future multi-core computing (e.g., [1, 2]), they do not discuss how to formally verify the very source codes they present. This gap must eventually be closed. Our approach was to use [1] in two ways: (i) as a source of many conceptual topics, and (ii) as a source of examples for our ongoing independent projects, such as non-blocking algorithms. As said earlier, in the final version of this paper, we will have updates on these projects.

We now detail the choice of a syllabus (Section 2), followed by a discussion of how we taught memory models (Section 3), threading (Section 4), and message passing (Section 5). We discuss work in progress in applying one of our dynamic verification tools to do all the exercises from a popular textbook on message passing (Section 5.1). When finished, this exercise may offer another valuable teaching resource in a domain where we do not know of any other comparable resources. Section 6 has our concluding remarks where we briefly describe some
of the student independent projects in progress (updates to be provided in our final version).

2 A Syllabus for Teaching Multi-core Computing

Our syllabus was refined all along, and now looks as follows:

– Provide an overview of current topics and discussions occurring around multi-core computing.
– Provide hands-on experience on parallelization and potential speed-up.
– Provide exposure to the basic concurrency paradigms of threading and message passing.
– Provide a detailed introduction to the topic of memory models.
– Provide more detailed exposure to the use of dynamic formal verification tools for threading and message passing.
– Provide an overview of writing state transition semantics using labeled transition systems. Ask them to hand-calculate the outcome of running a short MPI program using a simple operational semantic definition.
– Help students firm-up independent project selections.
– Use [1] and cover many topics, including: correctness criteria, various locking protocols, issues such as ABA, memory models, foundations of atomic objects, linearizability, the concept of consensus numbers, etc.

By design, this syllabus reflects this authors’ background, as well as focus on correctness issues. It helped us cover a variety of topics not found in a single textbook. We now detail what was covered under each of the above topics.

Overview of current topics: Sutter [6], and Sutter and Larus [7] provide high level motivations for multi-core computing. A good overview of the need to be addressing heterogeneous multi-cores is provided in [5]. This paper is accompanied by a spreadsheet that allows students to vary the assumptions about core sizes and the portion of code that is serial, and obtain speed-up curves. The prevailing consensus is that future multi-core systems will employ heterogeneous cores for reasons supported by this analysis.

The paper by Dennis [3] provides a historical perspective on concurrency, and the virtues of employing a functional/declarative notation. Similar points are also made by Blelloch [4] who has developed a successful pedagogical approach based on functional programming. Cantrill et al. take a somewhat sober look at the multi-core hoopla, claiming that the advances in parallel software design over the decades of the 70s and 80s created the need for multi-core CPUs of today. We finished off this section with an in-depth look at [9] that provides a comprehensive overview of work, span, and parallel speed-up.

Hands-on Experience: We then provided hands-on experience for students in terms of using a parallel machine and observing/understanding speed-up. We found that the most expedient way to do this was to employ the Cilk system [21]. Students with minimal background can begin using Cilk and observed speed-up on examples such as various versions of the merge sort, parallel array sum, etc.
When we did this work, we only had access to classical symmetric multiprocessor (SMP) clusters that most institutions have. Our most capable machine had eight dual-core CPUs. Also we installed the “classical Cilk” from MIT [10], as the industrial version Cilk++ was not available at the very beginning. Nevertheless this segment of experience, and the reading of Cilk papers was very fulfilling.

**Introduction to Threading and Message Passing:** We found the online tutorials at [11] to be a very good introduction to several concurrency paradigms, including threading and message passing. We applied our tools Inspect and ISP to verify many of their examples for a collection of standard safety properties. This exercise proved to be a valuable learning aid to our students – both in becoming familiar with Pthreads and MPI, in learning the use of dynamic verification tools, and in detecting known bugs in these examples.

### 3 Teaching Memory Models

We now turn to a topic that is considered too esoteric by many people. We show that by choosing simple examples, this topic can indeed be made very accessible and ‘gut-level.’ All the material discussed in this section can be covered in two assignments in about two weeks. We divide our presentation into language level memory models (Section 3.1), architectural memory models (Section 3.2), and the use of memory ordering checking tools (Section 3.3). We present all these topics as given out in our assignments. We skip a formal presentation of these topics which have been covered in many other places.

#### 3.1 Language Level Memory Models

Most compilers (e.g., gcc) are unaware of what is “safe” to do in a multithreaded or multi-core situation. To study this, let us experiment with `interlock1.c` below.

```c
#include <pthread.h>
int p1=0, p2=0;
char aa=0, bb=0; // handshake bits
void * thread_routine1(void * arg)
{
  do {
    aa = 1; while(!bb); bb = 0; p1++; if (!(p1 % 10)) printf("progress1\n");
  } while(1);
}
void * thread_routine2(void * arg)
{
  do {
    bb = 1; while(!aa); aa = 0; p2++; if (!(p2 % 10)) printf("progress2\n");
  } while (1);
}
int main()
{
  pthread_t t1, t2; pthread_create(&t1, 0, thread_routine1, 0);
  pthread_create(&t2, 0, thread_routine2, 0); pthread_join(t1, 0);
  pthread_join(t2, 0); return 0;
}
```

Now, the students were asked to answer these questions:
gcc, no optimization: Do gcc -o interlock1 interlock1.c -lpthread and run interlock1 on a uniprocessor and then a multiprocessor machine. Did you see the execution “hang”? Explain in a few sentences.

gcc, with optimization: Include the -O3 flag. Now what are the results you obtain? (It must hang!). Explain in a few sentences as to why. Will it hang on a uniprocessor?

The students could appreciate that due to the store/load optimization not being done, the unoptimized versions did not hang, while the optimized versions did. They were encouraged to do gcc -S interlock1.c and then look at the assembly code emitted in interlock1.s.

3.2 Architectural Memory Models

The students were now shown how the unoptimized code itself would break on a multiprocessor. They had to re-run the code multiple times. (Often times, they had to repeatedly abort the execution through control-C and re-run the code. If they were persistent, they could get the Caught violation message to get printed!)

```c
#include <pthread.h>
int aa=0, bb=0, zz=0;
int c1=0, c2=0;
void * thread_routine1(void * arg)
{ do {
    c1++; aa = 1; if (!bb) zz++; zz = 0; bb = 0;
    } while(1);
}
void * thread_routine2(void * arg)
{ do {
    c2++; bb = 1; if (!aa) zz++; zz = 0; aa = 0;
    } while (1);
}
void * thread_routine3(void * arg)
{ while(1)
    if (zz > 1) printf("Caught violation, c1=%d, c2=%d, zz=%d\n", c1, c2, zz);
}
int main()
{ pthread_t t1, t2, t3;
  pthread_create(&t3, 0, thread_routine3, 0);
  pthread_create(&t1, 0, thread_routine1, 0);
  pthread_create(&t2, 0, thread_routine2, 0);
  pthread_join(t1, 0); pthread_join(t2, 0);
  pthread_join(t3, 0);
  return 0;
}
```

Next, the students were asked to put in an mfence into the assembly file to prevent the store to load reordering. They tried – in vain – to trigger the violation! In our experience, such hands-on experiences get the students to understand what memory models are basically about.

3.3 Teaching Tools for Memory Models

In this segment, they were given a demo of a SAT-based memory consistency checker called mpec [12, 13]. They were also asked to run ARCHTEST – an
execution based calibrator of actual multi-processor machines [14] – on our lab machines. We then asked the students to study the reference document on the x86 memory model [15], asking them to re-state the memory model – as best as possible – in the parlance of Collier’s classifications. While such exercises are likely to receive different answers, they at least allow the students to help apply what they learn in one context to another. These were followed by the reading of the papers [17, 18] which further reinforced the need for rigorous concurrency specifications pertaining to compilation.

4 Teaching Threading

Students were now taught the basic use of the Inspect tool and model check the actual Pthreads/C code of a (buggy) producer/consumer routine using it [20]. To the best of our knowledge, Inspect is the only tool employing DPOR to model check Pthreads/C programs for a given test harness. We also encouraged some students to verify a C# implementation using CHESS. The fact that programs such as the following can be push-button verified using Inspect almost instantaneously was gratifying. Inspect was also endowed with an error-trace stepping facility using an Emacs-based user interface, thus helping users debug effectively. While both Inspect and CHESS can suffer from interleaving explosion, we could compare these tools against some criteria: (i) soundness - Inspect is sound for a test harness; for all practical purposes, CHESS is also sound, as it can search fully up to a target pre-emption bound (a pre-emption bound of 3 is usually adequate); (ii) race detection - Inspect does race detection, while CHESS will have this feature in the next release, (iii) liveness - CHESS supports a pragmatic version of liveness while Inspect does not currently have a notion of liveness.

5 Teaching Message Passing

A large body of material has been assembled to teach message passing using MPI. Here we summarize the material:

- Various examples on MPI programming have been assembled at [25], with many of them corresponding to [11].
- Parts of a matrix multiplication challenge - originally proposed by Siegel [26]. These have been solved using ISP.
- The same matrix multiplication examples were run on our local cluster machine and the speedups were plotted [26].

All in all, these examples capture the abilities of ISP as well as its graphical user interface. The references cited here show how we painstakingly can step through many related MPI programs. (Note: This part of our work was the result of an independent project.)
5.1 Pedagogical Material from Pacheco’s MPI Textbook

One exercise we are engaged in is solving as many of the exercises as we can from the popular book on MPI programming, by Pacheco. The current website representing our work is at [16]. We believe that once this project is finished, anyone wanting to teach MPI using this textbook will find ISP to be a ready companion. (The author of [19] was recently contacted by us; he was very appreciative of ISP’s bug-hunting abilities.)

A ‘Red Herring: A strange bug that we encountered in solving examples from Pacheco’s book is now briefly summarized. In one exercise, an MPI process is to receive from a process whose rank is calculated as $(e-1)\%2$. In one instance, $e$ happened to be 0, and so the rank turned out to be $-1\%2$ which C promptly evaluates to be $-1$. Now, $\text{MPI_PROC_NULL}$ was defined to be $-1$ in the MPI system. The MPI standard further says that receiving from $\text{MPI_PROC_NULL}$ is tantamount to a no op! Therefore, all said and done, in this example, the MPI receive was rendered a no op due to a C “feature.” (Mathematicians may regard $-1\%2$ as 1, in which case the receive would not end up being a no op.) In reality, what happened is that this triggered a bug in ISP which was not designed to support MPI receives from a null process. This bug is now being fixed; after the fix, ISP would deadlock on this example because, essentially we are losing one ‘MPI receive’ due to this. This will orphan the corresponding MPI send, causing a deadlock.

The moral of this story is that it takes a lot of domain knowledge before one can debug concurrent programs.

6 Concluding Remarks

We described a class in progress where several approaches to verify concurrent programs and to understand concurrency situations are being taught. This is just one instructors’ approach to make up a syllabus as the class went on; however, it is also unclear how else to teach such a class in such a rapidly evolving area. In the end, we were pleased that we took a hands-on approach so that the students obtained a firm grounding on many slippery concurrency notions.

The class is still in progress, as are all the side verification projects (e.g., solving examples from [1] using CHESS). We hope to have a full report by the final deadline. The class students have also chosen a number of projects; for example: (i) writing a Ray-tracer using Cilk++; (ii) verifying the non-blocking queues from [1] using CHESS; (iii) implementing a work-stealing algorithm in Pthreads/C and verifying using Inspect; (iv) several other projects involving the study of various concurrency APIs, and their multi-core realizations.

All in all, this paper presents a detailed experience report with citations that may prove to be useful to others. If there is any definite advice to impart, it is to use tools and testing experiments well before theoretical topics are taught. For instance, our syllabus in Section 2 shows that we are teaching the harder conceptual material from [1] only after many hands-on exercises have been done.
In our opinion, this will ensure that the material is appreciated in the right context by students who may otherwise be overwhelmed and/or think that the concurrency notions are disconnected with reality.

References


