ECE459: Programming for Performance Winter 2013

Lecture 19 — March 19, 2013

Patrick Lam *version 1*

This week, we'll talk about techniques for sacrificing accuracy in favour of performance; software transactional memory; and special-purpose languages for high-performance computing.

Reduced-resource computation

In Assignment 4, you will manually implement an optimization that trades off accuracy for performance. In that specific case, I'll outline how some domain knowledge could enable you to skimp out on some unnecessary computation: points that are far away contribute only very small forces.

In [RHMS10], Martin Rinard summarizes two of his novel ideas for automatic or semiautomatic optimizations which trade accuracy for performance: early phase termination [Rin07] and loop perforation [HMS$^+$09]. Both of these ideas are applicable to code we've learned about in this class.

Early phase termination

We've talked about barriers quite a bit. Recall that the idea is that no thread may proceed past a barrier until all of the threads reach the barrier. Waiting for other threads causes delays. Killing slow threads obviously speeds up the program. Well, that's easy.

"Oh no, that's going to change the meaning of the program!"

Let's consider some arguments about when it may be acceptable to just kill (discard) tasks. Since we're not completely crazy, we can develop a statistical model of the program behaviour, and make sure that the tasks we kill don't introduce unacceptable distortions. Then when we run the program, we get an output and a confidence interval.

Two Examples. When might this work? Monte Carlo simulations are a good candidate; you're already picking points randomly. Raytracers can work as well. Both of these examples could spawn a lot of threads and wait for all threads to complete. In either case, you can compensate for missing data points, assuming that they look like the ones that you did compute.

Also recall that, in scientific computations, you're entering points that were measured (with some error) and that you're computing using machine numbers (also with some error). Computers are only providing simulations, not the ground truth; the question is whether the simulation is good enough.

Loop perforation

You can also apply the same idea to sequential programs. Instead of discarding tasks, the idea here is to discard loop iterations. Here's a simple example: instead of the loop,

```
for (i = 0; i < n; i++) sum += numbers[i];
```

simply write,

```
for (i = 0; i < n; i += 2) sum += numbers[i];
```

and multiply the end result by a factor of 2. This only works if the inputs are appropriately distributed, but it does give a factor 2 speedup.

Example domains. In [RHMS10], we can read that loop perforation works for evaluating forces on water molecules (in particular, summing numbers); Monte-Carlo simulation for swaption pricing; and video encoding. In that example, changing loop increments from 4 to 8 gives a speedup of 1.67, a signal to noise ratio decrease of 0.87%, and a bitrate increase of 18.47%, producing visually indistinguishable results. The computation looks like this:

```
min = DBL_MAX;
index = 0;
for (i = 0; i < m; i++) {
  sum = 0;
  for (j = 0; j < n; j++) sum += numbers[i][j];
  if (min < sum) {
    min = sum;
    index = i;
  }
}
```

The optimization changes the loop increments and then compensates.

Software Transactional Memory

Developers use software transactions by writing `atomic` blocks. These blocks are just like `synchronized` blocks, but with different semantics.

```
atomic {
  this.x = this.z + 4;
}
```

You're meant to think of database transactions, which I expect you to know about. The `atomic` construct means that either the code in the atomic block executes completely, or aborts/rolls back in the event of a conflict with another transaction (which triggers a retry later on).

Benefit. The big win from transactional memory is the simple programming model. It is far easier to program with transactions than with locks. Just stick everything in an atomic block and hope the compiler does the right thing with respect to optimizing the code.

Motivating Example. We'll illustrate STM with the usual bank account example[1].

```
transfer_funds(Account* sender, Account* receiver, double amount) {
  atomic {
    sender->funds -= amount;
    receiver->funds += amount;
  }
}
```

Using locks, we have two main options:

- Big Global Lock: Lock everything to do with modifying accounts. (This is slow; and you might forget to grab the lock).
- Use a different lock for every account. (Prone to deadlocks; may forget to grab the lock).

With STM, we do not have to worry about remembering to acquire locks, or about deadlocks.

Drawbacks. As I understand it, three of the problems with transactions are as follows:

- I/O: Rollback is key. The problem with transactions and I/O is not really possible to rollback. (How do you rollback a write to the screen, or to the network?)
- Nested transactions: The concept of nesting transactions is easy to understand. The problem is: what do you do when you commit the inner transaction but abort the nested transaction? The clean transactional fa cade doesn't work anymore in the presence of nested transactions.
- Transaction size: Some transaction implementations (like all-hardware implementations) have size limits for their transactions.

Implementations. Transaction implementations are typically optimistic; they assume that the transaction is going to succeed, buffering the changes that they are carrying out, and rolling back the changes if necessary.

One way of implementing transactions is by using hardware support, especially the cache hardware. Briefly, you use the caches to store changes that haven't yet been committed. Hardware-only transaction implementations often have maximum-transaction-size limits, which are bad for programmability, and combining hardware and software approaches can help avoid that.

[1]Apparently, bank account transactions aren't actually atomic, but they still make a good example.

Implementation issues. Since atomic sections don't protect against data races, but just rollback to recover, a datarace may still trigger problems in your program.

```
atomic {
  x++;
  y++;
}
```

```
atomic {
  if (x != y)
     while (true) { }
}
```

In this silly example, assume initially `x = y`. You may think the code will not go into an infinite loop, but it can.

High-performance Languages

In recent years, there have been three notable programming language proposals for high-performance computing: X10[2], Project Fortress[3], and Chapel[4]. I'll start with the origins and worldview of these projects, and then discuss specific features. You can find a good overview in [LY07].

Current Status

You can download sample implementations of these languages, and they seem to be actively-developed (for varying definitions of active). Fortress has been formally wrapped up[5]. Chapel and X10 seem to have more activity (recent releases, workshops).

Context. The United States Defense Advanced Research Projects Agency (DARPA) started the "High Productivity Computing Systems"[6] project in 2002. This project includes both hardware and languages for high-performance computing; in this lecture, we focus on the languages and touch on the hardware that they were built to run on, which includes IBM's Power 775 and the Cray XC30 ("Cascade").

Machine Model. We've talked about both multicore machines and clusters. The target HPCS machine is somewhere in the middle; it will include hundreds of thousands of cores (hundreds of thousands) and massive memory bandwidth (petabytes/second). The memory model is a variation of PGAS, or Partitioned Global Address Space, which is reminiscent of the GPU memory model: some memory is local, while other memory is shared. It is somewhere between the MPI message-passing model and the threading shared-memory model.

Unlike in MPI, in these languages you write one program with a global view on the data, and the compiler figures out (using your hints) how to distribute the program across nodes. (In MPI, you have to write specific, and different, code for the server and client nodes.)

[2]`http://x10-lang.org`
[3]`http://java.net/projects/projectfortress`, `http://projectfortress.sun.com/Projects/Community`
[4]`http://chapel.cray.com`
[5]`https://blogs.oracle.com/projectfortress/entry/fortress_wrapping_up`
[6]`http://www.hpcwire.com/features/17883329.html`

Base Languages. X10 looks like a Java variant. Fortress takes some inspiration from Fortran, but is meant to look like mathematical notation (possibly rendered to math). Chapel also doesn't aim to look like any particular existing language. All of these language support object-oriented features, like classes.

Parallelism. All three of these languages require programmers to specify the parallelism structure of their code (in various ways). Fortress evaluates loops and arguments in parallel by default, or implicitly (if it chooses to), while the other languages have optional constructs like `forall` and `async` to specify parallelism.

Visible Memory Model. As we've said before, all of these languages expose a single global memory; it may perhaps be costly to access some parts of the memory, but it's still transparent. Fortress programs divide the memory into locations, which belong to regions. Regions are classified into a hierarchy, and nearby regions would tend to have better communication. X10 has places instead of regions. Places run code and contain storage. Chapel uses locales.

As a developer, you get to control locality of your data structures in these languages (by allocating at the same place or at different places, for instance). They make it easy to write distributed data structures.

X10 Code Example. Let's solve Assignment 1 using X10.

```
import x10.io.Console;
import x10.util.Random;

class MontyPi {
  public static def main(args:Array[String](1)) {
    val N = Int.parse(args(0));
    val result=GlobalRef[Cell[Double]](new Cell[Double](0));
    finish for (p in Place.places()) at (p) async {
      val r = new Random();
      var myResult:Double = 0;
      for (1..(N/Place.MAX_PLACES)) {
        val x = r.nextDouble();
        val y = r.nextDouble();
        if (x*x + y*y <= 1) myResult++;
      }
      val ans = myResult;
      at (result) atomic result()() += ans;
    }
    val pi = 4*(result()())/N;
  }
}
```

This code distributes the computation.

We could also replace `for (p in Place.places())` by `for (1..P)` (where P is a number) for a parallel solution. (Also, remove `GlobalRef`).

- `async`: creates a new child activity, which executes the statements.
- `finish`: waits for all child `asyncs` to finish.
- `at`: performs the statement at the place specified; here, the processor holding the result increments its value.

Bonus Link

`http://x10-lang.org/documentation/practical-x10-programming/performance-tuning.html`

References

[HMS+09] Henry Hoffmann, Sasa Misailovic, Stelios Sidiroglou, Anant Agarwal, and Martin Rinard. Using code perforation to improve performance, reduce energy consumption, and respond to failures. Technical Report MIT-CSAIL-TR-2009-042, MIT CSAIL, Cambridge, MA, September 2009.

[LY07] Ewing Lusk and Katherine Yelick. Languages for high-productivity computing: The DARPA HPCS language project. *Parallel Processing Letters*, 17(1):89–102, March 2007.

[RHMS10] Martin Rinard, Henry Hoffmann, Sasa Misailovic, and Stelios Sidiroglou. Patterns and statistical analysis for understanding reduced resource computing. In *Proceedings of Onward! 2010*, pages 806–821, Reno/Tahoe, NV, USA, October 2010. ACM.

[Rin07] Martin Rinard. Using early phase termination to eliminate load imbalances at barrier synchronization points. In *Proceedings of OOPSLA 2007*, pages 369–386, Montreal, Quebec, Canada, October 2007.