Lecture 19—Reduced-resource computation, STM, Languages for HPC

ECE 459: Programming for Performance

March 19, 2013

Part I

Reduced-Resource Computation

Trading Accuracy for Performance

Consider Monte Carlo integration.
It illustrates a general tradeoff: accuracy vs performance.
You'll also implement this tradeoff manually in A4
(with domain knowledge).

Martin Rinard generalized the accuracy vs performance tradeoff with:

- early phase termination [OOPSLA07]
- loop perforation [CSAIL TR 2009]

Early Phase Termination

We've seen barriers before.
No thread may proceed past a barrier until all of the threads reach the barrier.

This may slow down the program: maybe one of the threads is horribly slow.

Solution: kill the slowest thread.

Early Phase Termination: Objection

“Oh no, that’s going to change the meaning of the program!”

Early Phase Termination: When is it OK anyway?

OK, so we don't want to be completely crazy.

Instead:

- develop a statistical model of the program behaviour.
- only kill tasks that don't introduce unacceptable distortions.

When we run the program:

get the output, plus a confidence interval.

Early Phase Termination: Two Examples

Monte Carlo simulators:
Raytracers:

- already picking points randomly.

In both cases: spawn a lot of threads.

Could wait for all threads to complete;
or just compensate for missing data points,
assuming they look like points you did compute.

Early Phase Termination: Another Justification

In scientific computations:

- using points that were measured (subject to error);
- computing using machine numbers (also with error).

Computers are only providing simulations, not ground truth.
Actual question: is the simulation is good enough?

Loop Perforation

Like early-phase termination, but for sequential programs:
throw away data that's not actually useful.

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

$$\Downarrow$$

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

This gives a speedup of ~ 2 if `numbers[]` is nice.

Works for video encoding: can't observe difference.

Applications of Reduced Resource Computation

Loop perforation works for:

- evaluating forces on water molecules (summing numbers);
- Monte-Carlo simulation of swaption pricing;
- video encoding.

More on the video encoding example:
Changing loop increments from 4 to 8 gives:

- speedup of 1.67;
- signal-to-noise ratio decrease of 0.87%;
- bitrate increase of 18.47%;
- visually indistinguishable results.

Applications of Reduced Resource Computation

Loop perforation works for:

- evaluating forces on water molecules (summing numbers);
- Monte-Carlo simulation of swaption pricing;
- video encoding.

More on the video encoding example:

Changing loop increments from 4 to 8 gives:

- speedup of 1.67;
- signal-to-noise ratio decrease of 0.87%;
- bitrate increase of 18.47%;
- visually indistinguishable results.

Video Encoding Skeleton Code

```
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.

Part II

Software Transactional Memory

STM: Introduction

Instead of programming with locks, we have transactions on memory.

- Analogous to database transactions

An old idea; recently saw some renewed interest.

A series of memory operations either all succeed; or
all fail (and get rolled back), and are later retried.

STM: Benefits

Simple programming model: need not worry about lock granularity or deadlocks.

Just group lines of code that should logically be one operation in an `atomic` block!

It is the responsibility of the implementer to ensure the code operates as an atomic transaction.

STM: Implementing a Motivating Example

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

[Note: bank transfers aren't actually atomic!]

With locks we have two main options:

- Lock everything to do with modifying accounts
  (slow; may forget to use lock).
- Have a lock for every account
  (deadlocks; may forget to use lock).

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

STM: Drawbacks

Rollback is key to STM.
    But, some things cannot be rolled back.
    (write to the screen, send packet over network)

Nested transactions.
    What if an inner transaction succeeds,
yet the transaction aborts?

Limited transaction size:
    Most implementations (especially all-hardware)
have a limited transaction size.

Basic STM Implementation (Software)

In all atomic blocks, record all reads/writes to a log.

At the end of the block, running thread verifies that no other threads have modified any values read.

If validation is successful, changes are **committed**. Otherwise, the block is **aborted** and re-executed.

Note: Hardware implementations exist too.

Basic STM Implementation Issues

Since you don't protect against dataraces (just rollback), a datarace may trigger a fatal error 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.

STM Implementations

Note: Typically STM performance is no worse than twice as slow as fine-grained locks.

- Toward.Boost.STM (C++)
- SXM (Microsoft, C#)
- Built-in to the language (Clojure, Haskell)
- AtomJava (Java)
- Durus (Python)

STM Summary

Software Transactional Memory provides a more natural approach to parallel programming:

no need to deal with locks and their associated problems.

Currently slow,

but a lot of research is going into improving it. (futile?)

Operates by either completing an atomic block,
or retrying (by rolling back) until it successfully completes.

Part III

High-Performance Languages

Introduction

DARPA began a supercomputing challenge in 2002.

Purpose:

  create multi petaflop systems (floating point operations).

Three notable programming language proposals:

- X10 (IBM) [looks like Java]
- Chapel (Cray) [looks like Fortran/math]
- Fortress (Sun/Oracle)

Machine Model

We've used multicore machines and will talk about clusters (MPI, MapReduce).

These languages are targeted somewhere in the middle:

- thousands of cores and massive memory bandwidth;
- Partitioned Global Address Space (PGAS) memory model:
    - each process has a view of the global memory
    - memory is distributed across the nodes, but processors know what global memory is local.

Parallelism

These languages require you to specify the parallelism structure:

- Fortress evaluates loops and arguments in parallel by default;
- Others use an explicit construct, e.g. `forall`, `async`.

In terms of addressing the PGAS memory:

- Fortress divides memory into locations, which belong to regions (in a hierarchy; the closer, the better for communication).
- Similarly, places (X11) and locales (Chapel).

These languages make it easier to control the locality of data structures and to have distributed (fast) data.

X10 Example

```
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;
  }
}
```

X10 Example: explained

This solves Assignment 1, but distributes the computation.

Could 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.

HPC Language Summary

- Three notable supercomputing languages: X10, Chapel, and Fortress (end-of-life).
- Partitioned Global Adress Space memory model: allows distributed memory with explicit locality.
- Parallel programming aspect to the languages are very similar to everything else we've seen in the course.

Part IV

Overall Summary

Today's Lecture

Three topics:

- Reduced-Resource Computation.
- High-Performance Languages.
- Software Transactional Memory.

ECE students: Good luck on your FYDP!