Lecture 14 — Early Termination, Reduced-Resource Computation

Patrick Lam & Jeff Zarnett
patrick.lam@uwaterloo.ca, jzarnett@uwaterloo.ca

Department of Electrical and Computer Engineering
University of Waterloo

September 4, 2022

Quitting Time

Knowing when to quit is wise.

Two Strategies

There are two basic ideas.

- Skip some parts of work
- Intentionally reduce accuracy to speed things up.

You may implement these strategies when you're writing an exam.

Your program might support only one.

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.

Oh noooo!

Example:

Remember, Remember...

Many problems are mathematically hard in nature: to find the optimal solution you have to consider every possibility.

Well, what this strategy presupposes is: don’t.

This Route Sucks

Imagine the travelling salesperson problem, just for the sake of an example.

There are *n* points to visit and you want to minimize the amount of travel time.

This Route Sucks

The only way to know if a solution is best is to consider every possible route.

One way we can know if we're wasting time is to remember previous outcomes.

This Route Sucks

The solution we're evaluating will have some travel cost in units (maybe kms).

If the currently-accumulated cost in kms is larger than the total of the thus-far best solution, give up.

Another idea?

Close Enough is Good Enough

Another approach is to stop as soon as you have a solution that's reasonable.

If our target is to get total travel under 500 km then we can stop searching as soon as we find one that satisfies this constraint.

You Have Ten Chances

You can also choose to reduce the amount of effort by trying, say, five or ten different possibilities and seeing which of those is the best.

There's no guarantee you'll get an optimal solution.

Interesting to think about: what does Google Maps do?

Trading Accuracy for Performance

Consider Monte Carlo integration.
It illustrates a general tradeoff: accuracy vs performance.
Martin Rinard generalized the accuracy vs performance tradeoff with:

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

Early Phase Termination: Two Examples

Monte Carlo simulators:
Raytracers:

- already picking points randomly.

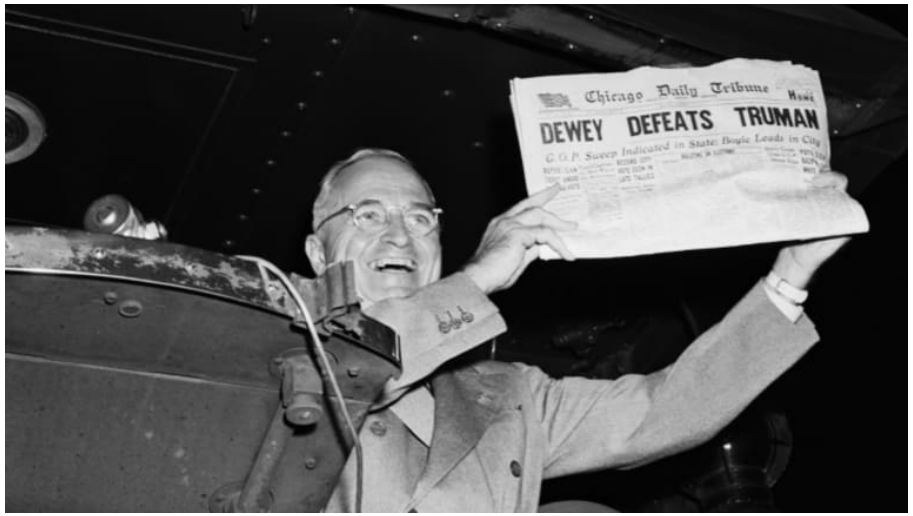
In both cases: spawn a lot of threads.

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

Calling the Election

In other cases, some threads simply take too long, but we don't need all of them to produce a result.

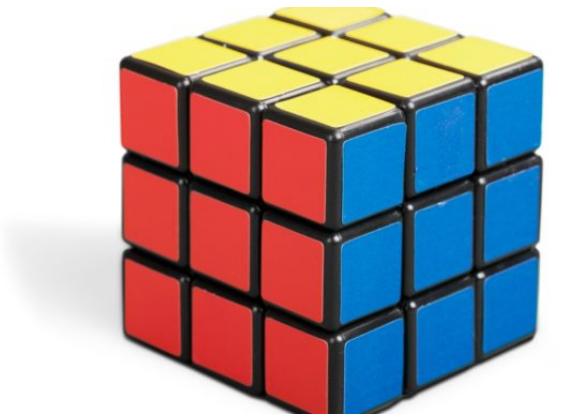
If we are evaluating some protocol where the majority wins, we can stop as soon as sufficient results have been returned.



A Solution Exists!

For some categories of problem, we know not only that a solution will exist, but also how many steps it takes to solve (optimally).

Consider the Rubik's Cube:



Brute force?

Rubik's Cube

Okay, that's fun to talk about, but it's always better if we see it in action?

Let's play around with `https://rubiks-cube-solver.com/`

Austerity Programs for Computer Programs

We do more with less!

Well, you can use `float/f32` instead of `double/f64`.

But you can also work with integers to represent floating point numbers.

But when is it appropriate?

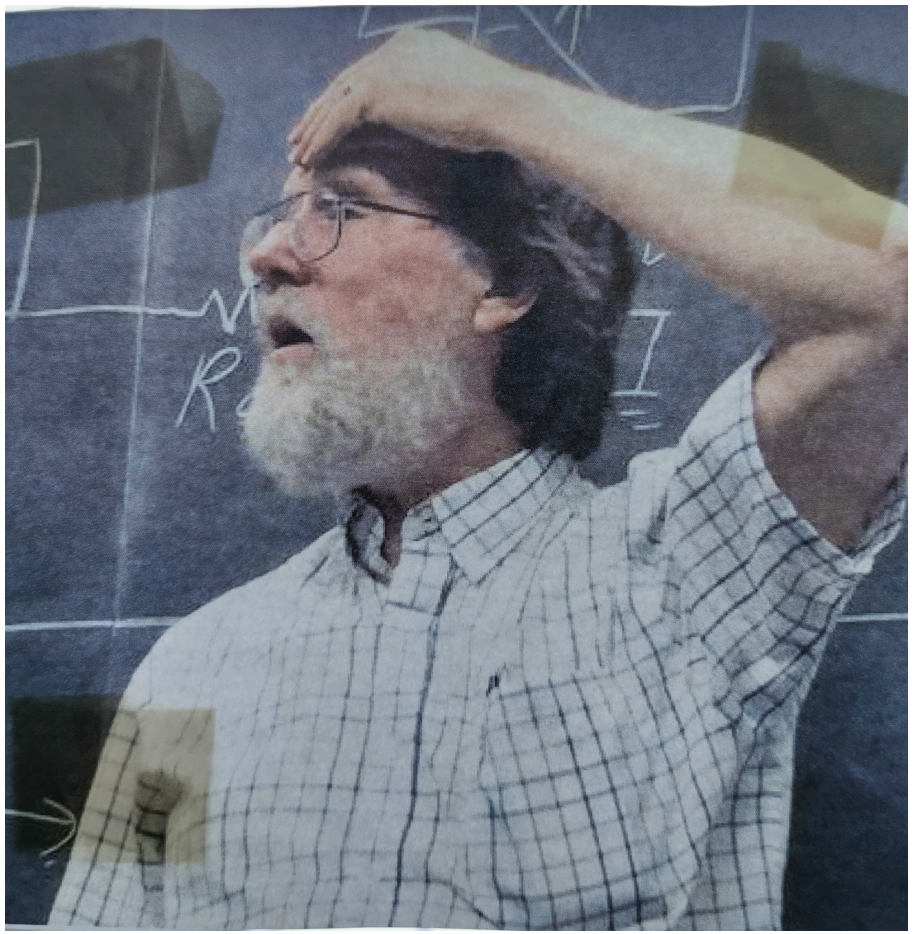
You're entering points that were measured (with some error) and you're computing using machine numbers (also with some error).

The question is whether the simulation is good enough.

What resistors are going to be put in your circuit board?

is there any point in calculating it down to five decimal places when the resistors you buy have a tolerance of $\pm 5\%$?

No, and if you took a circuits course with Prof. Barby he would be very disappointed if you said yes.



Quake III: “fast inverse square root”.

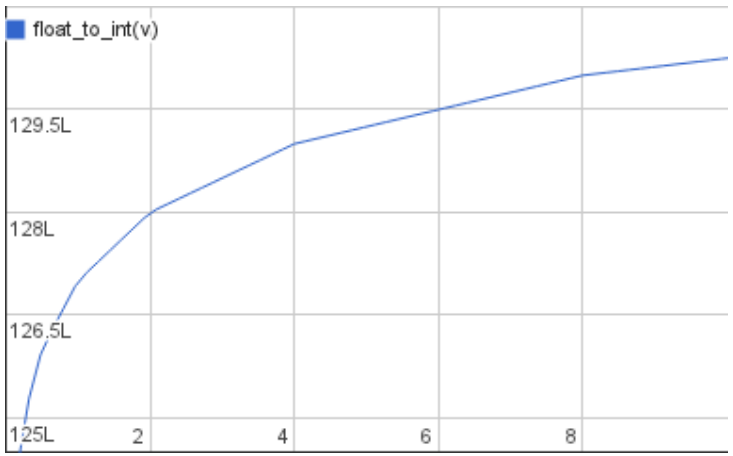
For graphics processing, sometimes you want to calculate $1/\sqrt{x}$. . Square root (or similar) is usually calculated by some interpolation or root-finding method

Fast Inverse Square Root

```
float FastInvSqrt(float x) {
  float xhalf = 0.5f * x;
  int i = *(int*)&x;            // evil floating point bit level hacking
  i = 0x5f3759df - (i >> 1);  // what the fuck?
  x = *(float*)&i;
  x = x*(1.5f-(xhalf*x*x));
  return x;
}
```

Now this probably seems like dark magic, and it is.

Float to Int?



Oh, never mind...

The clever hack is somewhat obsoleted now by the fact that CPU instructions now exist to give you fast inverse square root.

This was obviously not something you could rely on in 1999.

The N-Body Simulation

A common physics problem that programmers are asked to simulate is the N-Body problem.

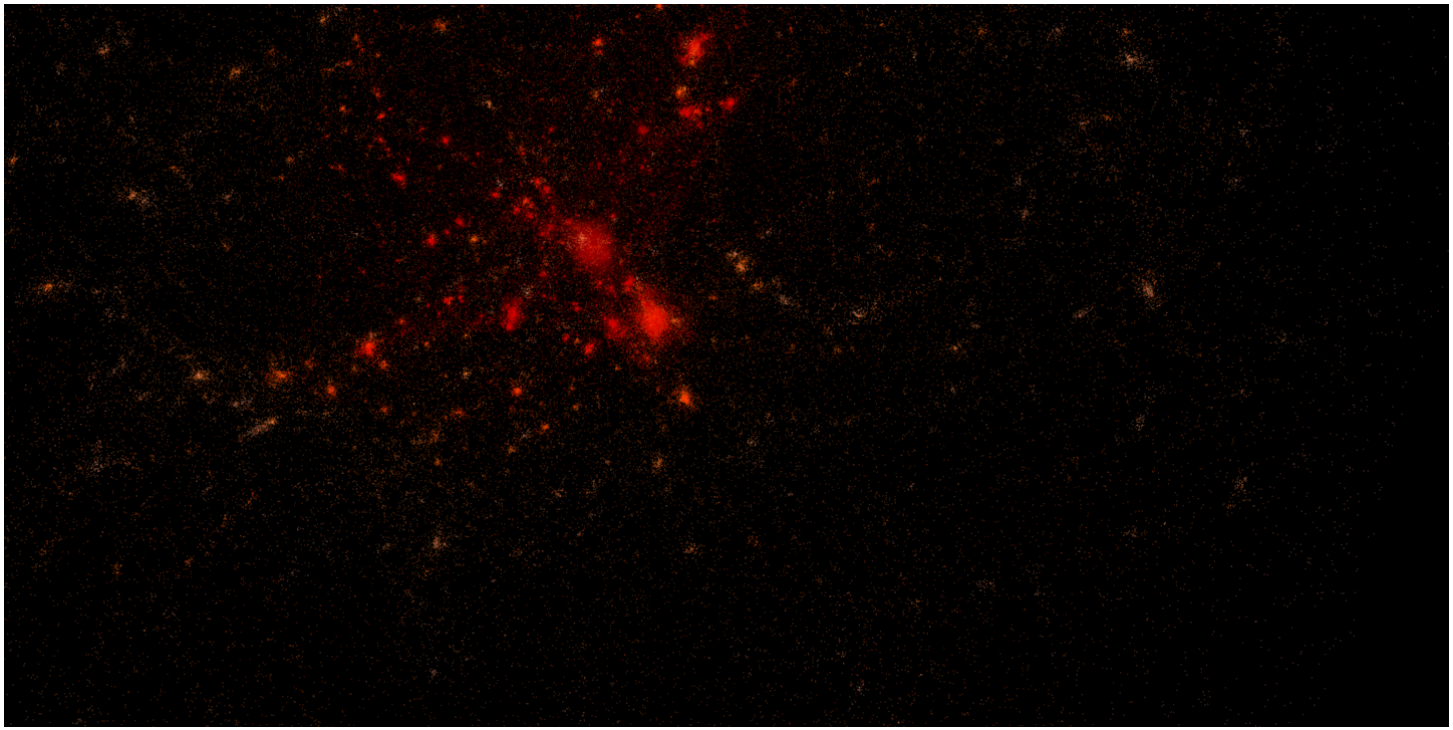


Image Credit: Michael L. Umbricht

Trade Accuracy for Performance

You could look for optimizations that trade off accuracy for performance.

As you might imagine, using `float` instead of `double`.

More? Then we need some domain knowledge.

Hint: consider the formula: $F = \frac{Gm_1m_2}{r^2}$.

The N-Body Simulation

Let's assume it's OpenCL converted and is optimized.

Can we use `float` instead of `double`?

What if we want more?

Estimation is Okay

Points that are far away contribute only very small forces.

So you can estimate them (crudely).

The idea is to divide the points into a number of “bins” which are cubes representing a locale of some sort.

Then, compute the centre of mass for each bin.

When calculating the forces: centre of mass for faraway bins; individual particles for nearby particles.

This used to be an assignment...

A more concrete explanation with an example: suppose the space is divided into $[0, 1000]^3$, so we can take bins which are cubes of length 100.

This gives 1000 bins.

To increase the accuracy, what should we do?

To increase the speed, what should we do?

Define Your Types

```
struct Point {
    x: f32,
    y: f32,
    z: f32,
    mass: f32,
}
```

We are going to improve this by adding a `bin` property to each point.

This tells us if something is close by.

Compute Centre of Mass

Compute all of the masses in parallel: create one thread per bin, and add a point's position if it belongs to the bin.

Note that this parallelizes with the number of bins.

Save Time

The payoff from all these calculations is to save time while calculating forces.

In this example, we'll compute exact forces for the points in the same bin and the directly-adjacent bins in each direction

That makes 27 bins in all, with 6 bins sharing a square, 12 bins sharing an edge, and 8 bins sharing a point with the centre bin).

If there is no adjacent bin (i.e., this is an edge), just act as if there are no points in the place where the nonexistent bin would be.

More Overhead

This does mean there is overhead in calculating bins, meaning the total amount of overhead goes up.

Is it worth it?

With 50 000 points:

- No approximations: 39 seconds
- With approximations: 37 seconds.

With 100 000 points:

- No approximations: 162 seconds
- With approximations: 147 seconds.

Of course, parallelizing this helps even more...

Loop Perforation

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

```
for i in 0 .. n { sum += numbers.get(i).unwrap(); }
```

```
for i in (0 .. n).step_by(2) { sum += numbers.get(i).unwrap(); }
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.