

Lecture Topics

- parallelism
 - is it hard to find?
 - is it hard to implement?

Administrivia

- return midterm: show graph, discuss stats, and go over difficult problems
- come talk to me if you scored < 60
- discussion of a few of the problems
 - A. problem asked specifically about practical issues rather than general issues; good general issues got 2/5; about half lost a few points
 - D. almost no one got it! seemed to miss the question, which was about how to simplify testing of failure-handling code for dynamic memory management; one answer: overload new and insert controlled failures; answer to a different question: use exceptions instead of checking return values to provide quality X
 - I.1. several answers were acceptable
 - N^2 lookups to create at $O(\log_2(N\epsilon))$ each
 - N^2 zeroes written + $N^2 \epsilon$ iterations ($O(1)$ iterator assumed)
 - answer is then either N^2 or $N^2 / \log_2(N\epsilon)$
 - in latter case, could assume $O(\log_2(N\epsilon))$ iterator, too
 - hard to tell apart from “knowing” which ones to look up
 - and $\epsilon \log_2(N\epsilon)$ likely to be < 1

Is Parallelism Hard to Find?

- my view: almost never
 - nearly always trivial from a task specification viewpoint
 - took me substantial effort
 - to find a problem that is fundamentally serial
 - or at least requires you to solve a hard math problem
 - as a result, we get industry speakers telling us how easy it is to use parallelism (e.g., IBM Cell compiler author, IBM researcher, and others)
- note: easy to see does not imply easy to use
- some practical concerns
 - may be hard to see parallelism by looking at code
 - may be hard to derive task specification from code
 - task specification may not make sense for code
 - e.g., library used for many purposes
 - what if best parallelism source for a task is not within library?
 - may be hard to rewrite sequential code cleanly to expose parallelism
- all of these issues refer to existing code
- one view
 - leveraging parallelism in existing code is hard
 - writing parallel code is not hard
 - it's just a matter of education
- fairly common view
 - both among people who have and haven't written parallel code
 - resurrected roughly every five years for the last 20
 - resulting in introduction of undergraduate parallelism classes
 - draw your own conclusions
 - but keep the new landscape in mind w.r.t. current generation

Is Parallelism Hard to Implement?

- my view: generally, yes
 - the joke version (it's funny because there's a great deal of truth in it): Only graduate students write parallel programs.
 - Pfister's version, in case the undergraduates don't fully get it: "parallelism is the wave of the future, and graduate students are inexpensive, intelligent, and motivated." (p. 221)
- is there a fundamental reason for the difficulty? maybe...
 - note
 - the following observation was made based on a talk by Matt Frank
 - credit to him, blame for being overly trite to me
 - parallelism (even "data parallelism") is a control abstraction:
 - two (or more) operations can be done at the same time
 - in a way in which they don't interfere with each other
 - everything we've talked about so far in this class has been data abstraction!
 - That difference should worry you.
 - (Yes, people have tried to create data-centric expression of parallelism: dataflow, pipeline parallelism, stream processing; it works sometimes.)
- let's carry that observation a little further...
 - start with some task and find one or more sources of parallelism
 - each source of parallelism is like a bunch of mini-programs
 - mini-program scale can vary
 - could be as big as a whole program (e.g., run RigelSim with different cache sizes to measure the impact of cache size on performance)
 - could be as small as an instruction (e.g., add an array of integers)
- now let's think about using these mini-programs

A Brief Introduction to the Challenges

- atomicity
 - atomicity is ALWAYS with respect to something
 - To a high-energy photon, atoms are not.
 - that said
 - do a pair of mini-programs need to execute atomically with respect to one another?
 - that is, does the end result have to appear as though the programs' execution did not interleave at all?
 - or, are there pieces of the mini-programs that must not appear to interleave?
 - note the “appear” part; to whom or to what?
 - for example
 - mini-programs that update your bank account
 - each pair of updates (read, change, write back) needs to be atomic with respect to one another
 - not a major issue for mini-programs with no shared data
 - Do two mini-programs share data?
 - sometimes easy to know, but sometimes not
 - Do two insertions into a hash table share data?
 - Are two graph node updates based on all of the nodes' neighbors atomic?
 - Can I make them atomic with a bipartite graph?
 - common failure types: algorithmic errors
 - programmer *thinks* that operations are independent
 - simply hasn't considered input data for which they are not
 - or some other programmer may reuse code but not understand assumptions that imply independence

- precedence/dependence: what is it?
 - atomicity does not constrain relative order
 - some mini-programs may have to finish before others start (data dependence)
 - results from one mini-program may eliminate the need to execute another mini-program (control dependence)

- common issues with precedence
 - programmer fails to express constraint
 - simple example: bipartite graph + discretized diff. eq's.
 - do updates need to alternate strictly?
 - or can I let them run ahead so long as they're atomic?
 - depends...
 - static problem (heat)? Yes! Run, run, run...
 - dynamic problem (em3d)? No way: physics matters!
 - speculation can be dangerous
 - TimeWarp discrete event simulator
 - try to execute state machine even if inputs may be en route
 - state machine emulation in software may cause crashes
 - is isolation + rollback ready for bizarre exceptions?