[after taking the five-minute break]

Administrivia

- Why have a five-minute break?
  - break up the monotony of lecture
  - give us a chance to stretch, use the bathroom
  - come back refreshed
- Do I really want to give up five minutes of lecture?
  - I'll try to end on time.
  - In return, please don't rustle backpacks, etc., until I'm done.

- office hours
  - Tu 1-3 upstairs at miaZa's / Caribou starting next week

- introductory handout

  - introduction [pass around handout]
  - balance of grads/undergrads (~1/3 grads historically)
  - course objectives and rationale
    - understand how and when to use modern language abstractions
    - understand the engineering design process behind language evolution
    - be able to enumerate and explain the challenges for parallel programming
    - be aware of the wide range of efforts to meet these challenges and why today we have no polished answers
  - work for the class
    - three or four homework + lab problems
    - two exams
  - sharing answers
    - ok on "homework"
    - ok within team on lab problems
  - intent: not 391++, but I'll try to leave projects open-ended enough for ambitious students
  - grading
  - web board + page
  - syllabus
    - ~1/3 language abstractions from an engineer's perspective
    - ~1/4 abstraction/performance tradeoffs
    - ~1/10 challenges/difficulties in parallel programming
    - ~1/3 overview and evaluation of approaches

Motivation: Language-Architecture Interaction

- Who cares about how programming languages and architecture interact?
- data courtesy of Saman Amarasinghe@MIT
  (+ Martin Rinard + Charles Leiserson)
- application
  - dense matrix multiply ($C_{ij} = \Sigma_k A_{ik} B_{kj}$)
  - on dual quad-core Intel machines
  - 1024×1024 matrices → 2^30 multiple-adds → a few seconds?

if you…	…you lose		
ignore processor parallelism	3.5×	~10×	"real" parallelism
don't use Intel hand-optimized assembly library	2.7×		μarch!
don't bother to vectorize (MMX/SSE)	2.8×	> 100×	arch
ignore cache size	1.7×		μarch!
ignore data org. in memory (don't transpose matrix)	3.4×	> 1000×	μarch!
use Java!	2.1×		language…sort of
use objects	2.2×		(extra insts?)
allow double & integer matrices	2.4×		branches (μarch!)
use immutable objects!	220×		language
	~300000×!		

- note that 2.1× is the kind of number that managed language proponents claim as the cost of using managed code
- the real cost is having no way to get at the remaining 100×, even if you're willing to do the work
- [MMX = multimedia extensions, SSE = streaming SIMD extensions]

Motivation: Parallel Programming

- "Parallelism is not cheaper or easier, it's *faster*." –Jim Gray, 1995
  [quoted from Soumen Chakrabarti's thesis; CHECK ORIGINAL SOURCE!
  VLDB Tutorial on "A Survey of Parallel Database Techniques and Systems]

- may no longer be true!
  - Moore's law continues: 2× transistors every 18 months. [fixed area]
  - Performance growth of single core flatlined ~3-4 years ago
    (not entirely true, but sharply reduced).
  - What will we do with the transistors?

- easy (relatively) hardware answer
  - build copies
  - let the software worry about it

- Jim Gray ca. 1996, after moving to Microsoft: Performance is not relevant to Microsoft; functionality is our primary concern.

- Will the software industry figure out how to overcome the additional problems inherent to parallel code?

- Maybe…

- It's an exciting time to be a young engineer!

Thoughts on Potential Labs

- memory debug implementation
- reference-counting garbage collection implementation
- problem solving with C++: correctness and performance
- profiling and optimization exploration/experimentation
- exploration of algorithm-architecture interaction in parallel kernels
- implementation of compiler/user task library

- parallelizing applications?
  - probably not as a main class project

- retrospective from observation of my own student
  - lecture hard to apply to real problems
  - labs help with learning

Why C++?

- illustrates most/all modern language abstractions
- reasonably transparent (although more complex than C)
- flexible enough to be used in systems programming
- [because] no requirement to use opaque abstractions
- easily linked with other code (e..g, hand-optimized assembly)

Philosophical Ruminations

- languages as engineering
  - (Str, p. 45) "…language design is not just design from first principles, but an art that requires experience, experiments, and sound engineering tradeoffs."
  - (Str, p. 48) "…most people focus on syntax issues to the detriment of type issues. The critical issues in the design of C++ were always those of type, ambiguity, and access control, not those of syntax."
    - people often like what they know
    - one reason that cycle between language research and widespread use is often ~20 years
    - anecdote about reviewed paper's claim of aesthetic superiority
  - (Str, p. 38) "I always maintained a clear view of what an object looked like in memory and considered how language features affected operations on such objects."
- one contrast: language as customer lock-in (Str, p. 37)
  - basic strategy is to create user dependence on revenue-generating activity
  - examples include
    - central support for tools (development environment, compiler, debugger)
    - training
    - consultants

- philosophy and some potential drawbacks of C++
  - Java vs. C++
    - Java authors considered C++ to be the kitchen sink of languages, i.e., a language overflowing with constructs that at best any given programmer had used once, and most of which most programmers would never use at all.
    - Stroustrup believes in accommodating a wide range of programming styles, and C++ reflects that philosophy. (Str pp. 23-24)
  - exposing novice programmers to C++ can be risky
    - kid in a candy store phenomenon
    - chewing gum can be used as glue
    - guidance usually helpful or even necessary

Historical View of C++

- early goals (Str p. 21)
  - support for structure and program organization (classes, as in Simula)
  - enable fast programs and freedom to mix with other languages
    - BCPL used as example
    - historically of extreme importance in industry; also one of the reasons that IDEs (integrated development environments) have been slow to catch on
  - portability
    - use of C preprocessor and compiler
    - limited complexity (more detail on Str p. 37)
      - simple enough to attract users
      - simple enough to implement (~ one week to port compiler)
    - limited integration with OS

- attractive qualities of C (Str, p. 43)
  - flexible/expressive
  - efficient
  - available
  - portable (relatively)

- sources of C++ ideas (other than C)
  - Simula: classes, virtual functions
  - Algol: operator overloading, references, mixing statements+decls
  - BCPL: // comments
  - Ada: templates, exceptions, namespaces
  - Clu: exceptions
  - ML: exceptions
- (also considered Modula-2, Smalltalk, and Mesa)
- also see chart in chapter 0 (Str, p.6)

- strategy adopted
  - backwards compatible
  - new tools must be simple, portable, and leverage existing ones
  - no new features used implies zero overhead
  - new features have minimal overhead

- quote for external perspective (ISO TR18015 C++ Performance, p. 202)