CS 612: Software Design for High-performance Architectures

Administration

- Instructor: Keshav Pingali
  - 457 Rhodes Hall
  - pingali@cs.cornell.edu
- TA: Milind Kulkarni
  - 490 Rhodes Hall
  - milind@cs.cornell.edu

Course content

- Course objective:
  - understand how to build intelligent software systems
- Course work:
  - 2 or 3 programming assignments
  - substantial course project
    - independent reading
    - implementation work
    - presentation

What is intelligence?

- Intelligence: ability to adapt effectively to the environment by
  - changing oneself
  - changing the environment
  - finding a new environment

- Objective of adaptation:
  - more efficient existence
  - improved chances of survival

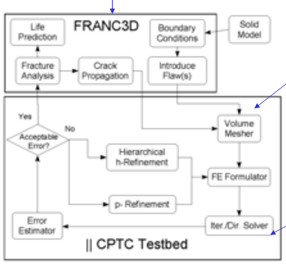
What does this have to do with software?

Computing environments are changing

- Current picture:
  - Monolithic application runs on one platform
    - Resources such as processors/memory are bound to application before it starts
      - Code optimized statically for those resources
    - Survivability: machine crashes → restart program
- Where we going: grid computing
  - Application may run in a distributed fashion across the net on several platforms during execution
    - Resources are bound dynamically
      - Code cannot be statically optimized for resources
    - Programs run for longer than the hardware MTBF
      - Cannot afford to restart application every time hardware fails

Grid Simulation Example

Fracture specialist: Cornell



Mesh generation: William&Mary

- Problems require meshes with $O(10^6)$ elements
- Time: 2-3 hours on // mc.s

Linear system solvers: MSU

- Large sparse systems $Ax = b$ where A has $O(10^7)$ elements
- Time: ~1 hour on // mc.s

Advantages of grid-based simulation

- Simplifies project management
  - no need for everyone to agree on a common implementation language or hardware platform
  - need agree only on data exchange format (XML/SOAP)
- Avoids software maintenance problem
  - each project site maintains its own code but makes it available to other partners as a web service
- In future
  - computations scheduled for execution wherever there are free resources
  - computations may even migrate during execution where more resources become available

Implications for software

- Software needs to be adaptive
  - adaptation for efficiency
    - application must be optimized dynamically when computation starts on or migrates to a new platform
  - adaptation for survival
    - adapt gracefully to processor and link failures: self-healing software

➔ Software must become more intelligent

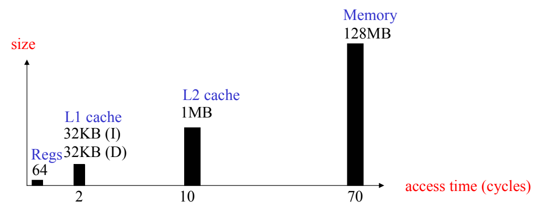
Adaptation for Efficiency

Performance bottleneck

"…The CPU chip industry has now reached the point that instructions can be executed more quickly than the chips can be fed with code and data. Future chip design is memory design. Future software design is also memory design. ….
Controlling memory access patterns will drive hardware and software designs for the foreseeable future."

Richard Sites, DEC

Memory Hierarchy of SGI Octane



- R10 K processor:
  - 4-way superscalar, 2 fpo/cycle, 195MHz
- Peak performance: 390 Mflops
- Experience: sustained performance is less than 10% of peak
  - Processor often stalls waiting for memory system to load data

Memory-wall solutions

- Latency avoidance:
  - multi-level memory hierarchies (caches)
- Latency tolerance:
  - Pre-fetching
  - multi-threading
- Techniques are not mutually exclusive:
  - Most microprocessors have caches and pre-fetching
  - Modest multi-threading is coming into vogue
  - Our focus: memory hierarchies

Hiding latency in numerical codes

- Most numerical kernels: $O(n^3)$ work, $O(n^2)$ data
  - all factorization codes
    - Cholesky factorization: $A = LL^T$ (A is spd)
    - LU factorization: $A = LU$
    - LU factorization with pivoting: $A = LU$
    - QR factorization: $A = QR$ (Q is orthogonal)
  - BLAS-3: matrix multiplication

  ➔use latency avoidance techniques
- Matrix-vector product: $O(n^2)$ work, $O(n^2)$ data
  - use latency tolerance techniques such as pre-fetching
  - particularly important for iterative solution of large sparse systems

Software problem

- Caches are useful only if programs have locality of reference
  - temporal locality: program references to given memory address are clustered together in time
  - spatial locality: program references clustered in address space are clustered in time
- Problem:
  - Programs obtained by expressing most numerical algorithms the straight-forward way do not have much locality of reference
  - Worrying about locality when coding algorithms complicates the software process enormously.

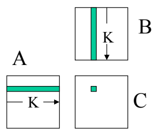
Example: matrix multiplication

```
DO I = 1, N    //assume arrays stored in row-major order
  DO J = 1, N
    DO K = 1, N
      C(I,J) = C(I,J) + A(I,K)*B(K,J)
```

- Great algorithmic data reuse: each array element is touched O(N) times!
- All six loop permutations are computationally equivalent (even modulo round-off error).
- However, execution times of the six versions can be very different if machine has a cache.

IJK version (large cache)

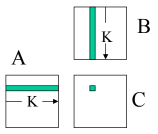
```
DO I = 1, N
  DO J = 1, N
    DO K = 1, N
      C(I,J) = C(I,J) + A(I,K)*B(K,J)
```



- Large cache scenario:
  - Matrices are small enough to fit into cache
  - Only cold misses, no capacity misses
  - Miss ratio:
    - Data size = $3 N^2$
    - Each miss brings in b floating-point numbers
    - Miss ratio = $3 N^2 /b*4N^3 = 0.75/bN = 0.019$ ($b = 4, N=10$)

IJK version (small cache)

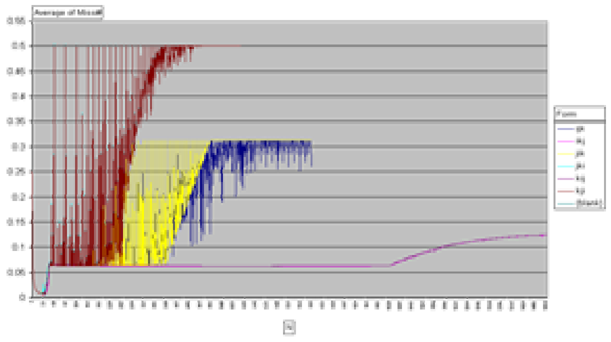
```
DO I = 1, N
  DO J = 1, N
    DO K = 1, N
      C(I,J) = C(I,J) + A(I,K)*B(K,J)
```



- Small cache scenario:
  - Matrices are large compared to cache/row-major storage
  - Cold and capacity misses
  - Miss ratio:
    - C: N^2/b misses (good temporal locality)
    - A: N^3/b misses (good spatial locality)
    - B: N^3 misses (poor temporal and spatial locality)
    - Miss ratio → 0.25 (b+1)/b = 0.3125 (for b = 4)

MMM Experiments

- Simulated L1 Cache Miss Ratio for Intel Pentium III
  - MMM with N = 1…1300
  - 16KB 32B/Block 4-way 8-byte elements



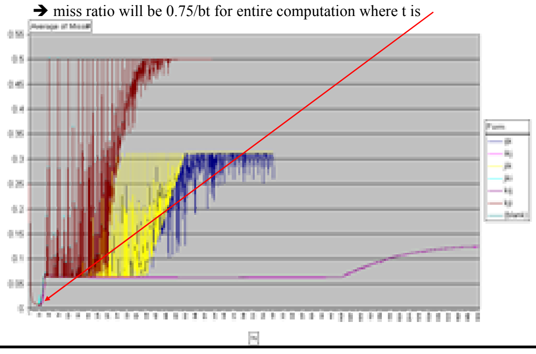
Quantifying performance differences

```
DO I = 1, N    //assume arrays stored in row-major order
  DO J = 1, N
    DO K = 1, N
      C(I,J) = C(I,J) + A(I,K)*B(K,J)
```

- Octane
  - L2 cache hit: 10 cycles, cache miss 70 cycles
- Time to execute IKJ version:
  $2N^3 + 70*0.13*4N^3 + 10*0.87*4N^3 = 73.2 N^3$
- Time to execute JKI version:
  $2N^3 + 70*0.5*4N^3 + 10*0.5*4N^3 = 162 N^3$
- Speed-up = 2.2
- Key transformation: loop permutation

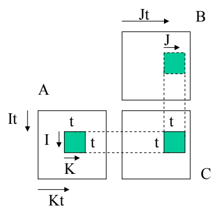
Even better…..

- Break MMM into a bunch of smaller MMMs so that large cache model is true for each small MMM
  - → large cache model is valid for entire computation
  - → miss ratio will be 0.75/bt for entire computation where t is



Loop tiling

```
DO It = 1,N, t
 DO Jt = 1,N,t
  DO Kt = 1,N,t
   DO I = It,It+t-1
    DO J = Jt,Jt+t-1
     DO K = Kt,Kt+t-1
      C(I,J) = C(I,J)+A(I,K)*B(K,J)
```



- Break big MMM into sequence of smaller MMMs where each smaller MMM multiplies sub-matrices of size txt.
- Parameter t (tile size) must be chosen carefully
  - as large as possible
  - working set of small matrix multiplication must fit in cache

Speed-up from tiling

- Miss ratio for block computation
  = miss ratio for large cache model
  = 0.75/bt
  = 0.001 (b = 4, t = 200) for Octane
- Time to execute tiled version =
  $2N^3 + 70*0.001*4N^3 + 10*0.999*4N^3 = 42.3N^3$
- Speed-up over JKI version = 4

Observations

- Locality optimized code is more complex than high-level algorithm.
- Loop orders and tile size must be chosen carefully
  - cache size is key parameter
  - associativity matters
- Actual code is even more complex: must optimize for processor resources
  - registers: register tiling
  - pipeline: loop unrolling
  - Optimized MMM code can be ~1000 lines of C code

Restructuring compilers (1985-)

- Compiler writer given detailed machine model
  - Number of registers.
  - Cache organizations.
  - Instruction set: mul-add? vector extensions? …
- Writes restructuring/optimizing compiler
  - for the given machine model
- Experience
  - Models may give sub-optimal parameter values

Empirical optimization

- Estimate transformation parameters and optimal versions by using exhaustive search
  - ATLAS: generator for automatically tuned BLAS kernels (Dongarra et al)
- Pros:
  - Not hostage to whims of restructuring compiler
  - Self-tuning: system can generate tuned code even for machines that did not exist when it was written
- Cons:
  - Search takes a long time: many hours just to generate optimized MMM

  ➔Impractical for large programs

One focus of CS 612

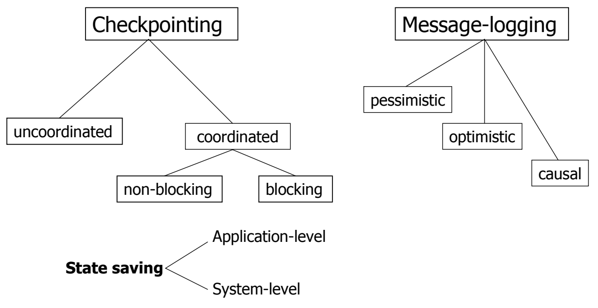
- Program analysis techniques:
  - define space of programs equivalent to original program
- Program transformation techniques:
  - from this space, select best parameterized restructured program
- Estimating parameter values
  - tile sizes, loop unrolling factors etc.

Adaptation for survival

Fault tolerance

- Fault tolerance comes in different flavors
  - Mission-critical systems: (eg) air traffic control system
    - No down-time, fail-over, redundancy
  - Computational applications
    - Restart after failure, minimizing expected time to completion of program
    - Guarantee progress
- Failure models
  - Fail-stop faults: computations die quietly without corrupting shared data or sending bad messages
  - Byzantine faults: arbitrary misbehavior is allowed

Fail-stop fault tolerance strategies



Our experience/beliefs:

- Message-logging does not work well for communication-intensive numerical applications
  - Many messages, much data
- System-level checkpoint is not as efficient as application-level
  - IBM’s BlueGene protein folding
    - Sufficient to save positions and velocities of bases
  - Alegra experience at Sandia labs
    - App. level restart file only 5% of core size

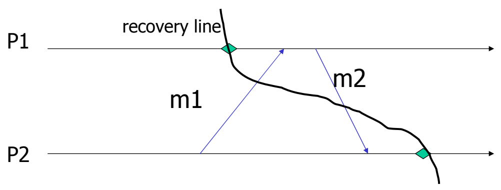
Sequential application state

- An application’s state consists of
  - Program counter
  - Call stack
  - Globals
  - Heap objects
- Optimizations:
  - Where should we checkpoint?
  - Memory exclusion
    - Live/Clean/Dead variable analysis
  - Recomputation vs. restoring
    - Protein folding example

Supporting MPI applications

- It is not sufficient to take a checkpoint of each individual process
- We need to account for the following
  - In-flight messages
  - Inconsistent messages
  - At application level, message send/receive not necessarily FIFO
    - Process can use tags to receive messages out of order

In-flight and inconsistent messages



- m1 is in-flight
  - After recovery, message is not resent but receiver wants to receive it again
- m2 is inconsistent
  - After recovery, message is resent but receiver does not want to receive it again

Previous work

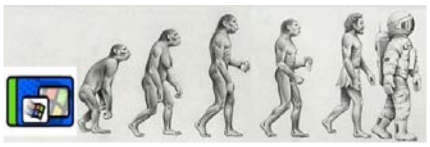
- Many distributed snapshot algorithms invented by distributed systems community
  - (eg) Chandy-Lamport protocol
- These protocols are not suitable for MPI programs
  - Developed for system-level check-pointing, not application-level check-pointing
  - Communication between processes is not FIFO
  - Process communication topology is not fixed
  - Do not handle collective communication

Discussion topics

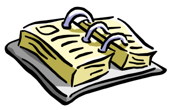
- Protocols for checkpointing of MPI programs
- Protocols for checkpointing of shared-memory programs
- Optimization of sequential saved state
- Portable checkpointing
- Applications in unrelated projects
  - SimSnap
  - Optimization of backward automatic differentiation

Summary

- Intelligence: ability to adapt to the environment by
  - changing oneself
  - changing the environment
  - finding a new environment
- Objective of adaptation:
  - more efficient existence
  - improved chances of survival
- Software has just started down this road.



Lecture schedule



- Application requirements:
  - computational science applications
- Architectural concerns:
  - memory hierarchies
- Program analysis and optimization
  - perfectly-nested loops
  - imperfectly-nested loops
  - empirical optimization
- Fault-tolerance
  - system-level and application-level protocols
  - optimizing state saving