L23: Parallel Programming Retrospective

December 3, 2009

Administrative

• Schedule for the rest of the semester
  - "Midterm Quiz" = long homework
  - Return by Dec. 15
  - Projects
    - 1 page status report due TODAY
      – handin cs4961 pstatus <file, ascii or PDF ok>
    - Poster session dry run (to see material) Dec. 8
    - Poster details (next slide)
  • Mailing list: cs4961@list.eng.utah.edu

Poster Details

• I am providing:
  • Foam core, tape, push pins, easels
  • Plan on 2ft by 3ft or so of material (9-12 slides)
• Content:
  - Problem description and why it is important
  - Parallelization challenges
  - Parallel Algorithm
  - How are two programming models combined?
  - Performance results (speedup over sequential)
• Example

Outline

• Last New Topic: Transactional Memory
• General:
  - Where parallel hardware is headed
  - Where parallel software is headed
  - Parallel programming languages
• Sources for today's lecture
  - Transactional Coherence and Consistency, ASPLOS 2004, Stanford University
  - Vivek Sarkar, Rice University
Transactional Memory: Motivation

- Multithreaded programming requires:
  - Synchronization through barriers, condition variables, etc.
  - Shared variable access control through locks . . .
- Locks are inherently difficult to use
  - Locking design must balance performance and correctness.
    Coarse-grain locking: Lock contention Fine-grain locking: Extra overhead, more error-prone
  - Must be careful to avoid deadlocks or races in locking
  - Must not leave anything shared unprotected, or program may fail
- Parallel performance tuning is unintuitive
  - Performance bottlenecks appear through low level events
    Such as: false sharing, coherence misses, . . .
  - Is there a simpler model with good performance?

Using Transactions (Specifically TCC)

- Concept: Execute transactions all of the time
- Programmer-defined groups of instructions within a program
  End/Begin Transaction Start Buffering Results
  Instruction #1
  Instruction #2 . . .
  End/Begin Transaction Commit Results Now (+ Start New Transaction)
  - Can only "commit" machine state at the end of each transaction
    - To Hardware: Processors update state atomically only at a coarse granularity.
    - To Programmer: Transactions encapsulate and replace locked "critical regions"
  - Transactions run in a continuous cycle . . .
A Parallelization Example

- **t_for transactional loop**
  - Runs as 1002 transactions (1 sequential + 1000 parallel, ordered + 1 sequential)
  - Maintains sequential semantics of the original loop

```c
int* data = load_data();
int i, buckets[101];
t_for (i = 0; i < 1000; i++) {
    buckets[data[i]]++;
}
print_buckets(buckets);
```

Conventional Parallelization of example

- Conventional parallelization requires explicit locking
  - Programmer must manually define the required locks
  - Programmer must manually mark critical regions
  - Even more complex if multiple locks must be acquired at once
  - Completely eliminated with TCC!

```c
int* data = load_data(); int i, buckets[101];
LOCK_TYPE bucketLock[101];
for (i = 0; i < 101; i++) LOCK_INIT(bucketLock[i]);
for (i = 0; i < 1000; i++) {
    LOCK(bucketLock[data[i]]);
    buckets[data[i]]++;
    UNLOCK(bucketLock[data[i]]);
}
print_buckets(buckets);
```

Other Concepts: Coherence and Fault Tolerance

- Main idea:
  - Convenience of coarse-grain reasoning about parallelism and data sharing
  - Hardware/software takes care of synchronization details
  - Well-suited to code with heavy use of locking
- If two transactions try to commit the same data?
- If a transaction fails to complete?

My Research in this Space

Compiler-based optimization and auto-tuning

Achieve high performance by exploiting architectural features...

... while freeing programmers from managing low-level details (productivity).
A Looming Software Crisis?

- Architectures are getting increasingly complex
  - Multiple cores, deep memory hierarchies, software-controlled storage, shared resources, SIMD compute engines, heterogeneity, ...

- Performance optimization is getting more important
  - Today's sequential and parallel applications may not be faster on tomorrow's architectures.
  - Especially if you want to add new capability!
  - Managing data locality even more important than parallelism.

Exascale Software Challenges

- Exascale architectures will be fundamentally different
  - Power management THE issue
  - Memory reduction to .01 bytes/flop
  - Hierarchical, heterogeneous

- Basic rethinking of the software "stack"
  - Ability to express and manage locality and parallelism for ~billion threads will require fundamental change
  - Support applications that are forward scalable and portable
  - Managing power (although locality helps there) and resilience requirements

Motivation: Lessons at the Extreme End

- HPC programmers are more willing than most to suffer to get good performance
  - But pain is growing with each new architecture
  - And application base is expanding (e.g., dynamic, graph-based applications)

- Government funding inadequate to make these systems usable

- Therefore, best hope is to leverage commodity solutions
  - Also, an interesting and fertile area of research lies in this intersection

Parallel Software Infrastructure Challenges

- Domain-specific Programming Models
- Middleware
- Application Libraries
- Programming Tools
- Languages
- Static & Dynamic Optimizing Compilers
- Multicore Back-ends
- Parallel Runtime & System Libraries
- OS and Hypervisors
Motivation: A Few Observations

- Overlap of requirements for petascale scientific computing and mainstream multi-core embedded and desktop computing.
- Many new and "commodity" application domains are similar to scientific computing.
  - Communication, speech, graphics and games, some cognitive algorithms, biomedical informatics ("RMS" applications)
- Importance of work with real applications (who is your client?)
  - Biomedical imaging, Molecular dynamics simulation, Nuclear fusion, Computational chemistry, speech recognition, knowledge discovery ...

Where is compiler research going?

- Main research directions:
  - Make parallel programming mainstream
  - Write compilers capable of self-improvement [autotuners]
  - Performance models to support optimizations for parallel code
  - Enable development of software as reliable as an airplane
  - Enable system software that is secure at all levels
  - Verify the entire software stack


Autotuning Research Themes

- Compiler-based performance tuning
  - Use vast compute & storage resources to improve performance
  - Enumerate options, generate code, try, measure, record (conceptually)
- Optimization and performance tuning built from modular, understandable chunks
  - Easier to bring up on new platforms
  - Facilitates collaboration, moving the community forward

A Systematic, Principled Approach!

Recent Research: Auto-Tuning "Compiler"

Traditional view:

<table>
<thead>
<tr>
<th>code</th>
<th>Batch Compiler</th>
<th>input data</th>
</tr>
</thead>
</table>

(Semi-)Autotuning Compiler:

<table>
<thead>
<tr>
<th>code</th>
<th>transformation script(s)</th>
<th>code Translation</th>
<th>input data (characteristics)</th>
<th>search script(s)</th>
<th>Experiments Engine</th>
</tr>
</thead>
</table>
Collaborative Autotuning in PERI

Future Directions: New Architectures

Nvidia Tesla system: 240 cores per chip, 960 cores per unit, 32 units!

- CS6963: Parallel programming for GPUs
- Paper and poster at SAAHPC and other work from class under submission
- Automatic CUDA code generation from CHiLL

NVIDIA Recognizes University Of Utah As A Cuda Center Of Excellence University of Utah is the Latest in a Growing List of Exceptional Schools Demonstrating Pioneering Work in Parallel (JULY 31, 2008—NVIDIA Corporation)