L23: Future Parallel Programming Languages

November 30, 2010

Final Project

Purpose:
- A chance to dig deeper into a parallel programming model and explore concepts.
- Present results to work on communication of technical ideas.

Write a non-trivial parallel program that combines two parallel programming languages/models. In some cases, just do two separate implementations.
- OpenMP + SSE-3
- OpenMP + CUDA (but need to do this in separate parts of the code)
- TBB + SSE-3
- MPI + OpenMP
- MPI + SSE-3
- MPI + CUDA

Present results in a poster session on the last day of class.

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)

Administrative

Grades to date
- Sriram will mail out this afternoon

CHPC accounts

Schedule for the rest of the semester
- "Final Exam" = long homework
  - Handed out Thursday (plus review)
  - Open notes, open text but do not discuss (ask for help)
  - Return electronically or physically by Dec. 14
- Projects
  - 1 page status report due Friday, Dec. 3 at 11:59PM
    - handin cs4961 pstatus <file, ascii or PDF ok>
  - Where are you and what do you have left
  - How have you addressed questions in the feedback
- Poster session dry run (to see material) Dec. 7
- Final Report (2-4 pages) also due Dec. 14
Outline

• Parallel Programming Language Concepts
• Chapel, an example Global View Language
• Transactional Memory
• Sources for today’s lecture
  - Book, chapters 9-10
  - Brad Chamberlain, Cray
  - Transactional Coherence and Consistency, ASPLOS 2004, Stanford University

Some Philosophical Grounding

• Most programs of the future will be parallel programs
• Not every university even teaches parallel programming, and most offer it as an elective to undergraduates
• Where will companies find parallel programmers?
  - Hide parallelism by making it implicit, only a few experts need to understand details
  - Libraries ala STL
  - High-level programming models, domain-specific tools
  - Examples?
• General conventional wisdom (propagated by Berkeley)
  - 90% of parallel programmers are only slightly more sophisticated than sequential programmers
  - 10% of experts understand how to obtain performance (this class)

A Broader Look at Parallel Programming Languages

• What are some important features of parallel programming languages (Ch. 9)?
  - Correctness
  - Performance
  - Scalability
  - Portability

And what about ease of programming?
Related to correctness but sometimes at odds with performance and scalability

Correctness concepts

• P-Independence
  - If and only if a program always produces the same output on the same input regardless of number or arrangement of processors
• Global view
  - A language construct that preserves P-independence
  - Example (Chapel in today’s lecture)
• Local view
  - Does not preserve P-independent program behavior
  - Example from previous lecture?
• Transactional memory (later in lecture)
  - Eliminate need for fine-grain synchronization through a higher level abstraction
**Performance**

- Role of programming language in achieving performance
  - Relationship between language, architecture and application
    - ?

- How well does language achieve performance across a range of architectures?

- How much work does the programmer have to do to achieve performance? Is it possible?

---

**Scalability**

- Weak scaling
  - Program continues to achieve speedup on more processors on a larger problem size
  - Example: A program for problem size M that runs with a speedup of \( T_s/T_p \) on N processors will achieve a similar speedup on a problem size of 2M on 2N processors
  - Limitations to weak scaling?

- Strong scaling
  - Program achieves speedup on more processors with the same problem size
  - Speedup of program on problem size M on 2N processors higher than speedup of same problem and N processors
  - This is very, very hard, but part of how we will achieve exascale performance.

- Importance of scaling on multi-core?

---

**Portability**

- Will a program written in parallel programming language “L” run on all the platforms on which I want to run my program, now and in the future?
  - May require a compiler implementation effort
  - Often this is the #1 barrier to acceptance of a new language
  - Contemporary positive and negative examples?

- Will it achieve “performance portability”, meaning it runs not only correctly but efficiently?

---

**What is a PGAS Language?**

- PGAS = Partitioned Global Address Space
  - Present a global address space to the application developer
  - May still run on a distributed memory architecture
  - Examples: Co-Array Fortran, Unified Parallel C

- Modern parallel programming languages present a global address space abstraction
  - Performance? Portability?

- A closer look at a NEW global view language, Chapel
  - From DARPA High Productivity Computing Systems program
  - Language design started around 2003
  - Also X10 (IBM) and Fortress (Sun)
Chapel Domain Example: Sparse Matrices

Recall sparse matrix-vector multiply computation

```plaintext
for (j=0; j<nr; j++) {
    for (k = rowstr[j]; k<rowstr[j+1]-1; k++)
        t[j] = t[j] + a[k] * x[colind[k]]; 
}
```

Chapel Formulation

Declare a dense domain for sparse matrix
const dnsDom = [1..n, 1..n];
Declare a sparse domain
var spsDom: sparse subdomain(dnsDom);
Var spsArr: [spsDom] real;
Now you need to initialize the spsDom. As an example, 
spsDom = [(1,2),(2,3),(2,7),(3,6),(4,8),(5,9),(6,4),(9,8)];
Iterate over sparse domain:
forall (i,j) in spsDom
    result[i] = result[i] + spsArr(i,j) * input[j];

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
  - 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 . . .
Transaction (TCC) Cycle

- Speculatively execute code and buffer
- Wait for commit permission
  - "Phase" provides commit ordering, if necessary
  - Imposes programmer-requested order on commits
  - Arbitrate with other CPUs
- Commit stores together, as a block
  - Provides a well-defined write ordering
  - To other processors, all instructions within a transaction "appear" to execute atomically at transaction commit time
  - Provides "sequential" illusion to programmers Often eases parallelization of code
  - Latency-tolerant, but requires high bandwidth
- And repeat!

A Parallelization Example

- Simple histogram example
  - Counts frequency of 0-100% scores in a data array
  - Unmodified, runs as a single large transaction (1 sequential code region)

```c
int* data = load_data();
int i, buckets[101];
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 < 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?