The State of Parallel Programming

Burton Smith
Technical Fellow
Microsoft Corporation

Parallel computing is mainstream

- Uniprocessors are reaching their performance limits
  - More transistors per core increases power but not performance
- Meanwhile, logic cost ($ per gate-Hz) continues to fall
  - What are we going to do with all that hardware?
- New microprocessors are multi-core and/or multithreaded
  - Single sockets have uniform shared memory architecture
  - Multiple chips may result in non-uniform shared memory
- New “killer apps” will need more performance
  - Better human-computer interfaces
  - Semantic information processing and retrieval
  - Personal robots
  - Games!
- *How should we program parallel applications?*

Parallel programming languages today

- Threads and locks
  - Threads running in uniform shared memory
- Fortran with automatic vectorization
  - Vector (SIMD) parallel loops in uniform shared memory
- Fortran or C with OpenMP directives
  - Barrier-synchronized SPMD threads in uniform shared memory
- Co-array Fortran, UPC, and Titanium
  - Barrier-synchronized SPMD threads addressing non-uniform shared memory regions
- Fortran or C with MPI
  - Threads in non-shared memory regions communicate via coarse-grained messages
- These are all pretty low-level

We need better parallel languages

Better parallel languages should introduce abstractions to improve programmer productivity:

- Implement high level data parallel operations
  - Programmer-defined reductions and scans, for example
- Exploit architectural support for parallelism
  - SIMD instructions, inexpensive synchronization
- Provide for abstract specification of locality
- Present a transparent performance model
- Make data races impossible

For the last goal, something must be done about *variables*

Shared memory

- Shared memory has several benefits:
  - It is a good delivery vehicle for high bandwidth
  - It permits unpredictable data-dependent sharing
  - It can provide a large synchronization namespace
  - It facilitates high level language implementations
- Language implementers like it as a target
- Non-uniform shared memory even scales pretty well
- But *shared variables* bring along a big drawback:
  stores *do not commute* with loads or other stores
  - Data races are the usual result
- Shared memory is an unsatisfactory programming model

Are *pure functional languages* a viable alternative?

Pure functional languages

- Imperative languages basically *schedule values into variables*
  - Parallel versions of such languages are prone to data races
  - The basic problem: there are too many parallel schedules
- Pure functional languages avoid races by avoiding variables
  - In a sense, they compute new constants from old ones
  - Data races are impossible (because loads commute)
  - We can reclaim dead constants pretty efficiently
- Program transformations are enabled by this concept
  - High-level operations become practical
  - Abstracting locality may also be easier
- But: functional languages can’t mutate state in parallel
  - Monads can do it, but only serially
- The problem is maintaining state *invariants* consistently

Maintaining invariants

- Serial iteration or recursion *perturbs* and then *restores* its invariant in local (lexical) fashion
  - Composability depends on maintaining the truth of the invariant
  - "Hoare logic" is built on all this
  - It's mostly automatic to us once we learn to program
  - What about maintaining invariants given parallel updates?
- Two requirements must be met for the state updates
  - Pairs of state updates must be prevented from interfering
    - That is, they must be *isolated* in some fashion
  - Also, updates must finish once they start
    - …lest the next update see the invariant false
    - We say the state updates must be *atomic*
- Updates that are isolated and atomic are called *transactions*

Commutativity and determinism

- If statements *p* and *q* preserve invariant *I* and do not "interfere", then their parallel composition { *p* || *q* } also preserves *I* †
- If *p* and *q* are performed in isolation and atomically, *i.e.* as transactions, then they will not interfere ‡
- Operations may or may not commute with respect to state
  - But we always get *commutativity with respect to the invariant*
- This leads to a weaker form of determinism
  - Long ago some of us called it "good nondeterminism"
  - It's the kind of determinism operating systems rely on

† Susan Owicki and David Gries. Verifying properties of parallel programs: An axiomatic approach. CACM **19**(5):279−285, May 1976.

‡ Leslie Lamport and Fred Schneider. The "Hoare Logic" of CSP, And All That. ACM TOPLAS **6**(2):281−296, Apr. 1984.

A histogramming example

```
const double in[N];      //input to be histogrammed
const int bin(double);   //the binning function
int hist[M] = {0};       //histogram, initially 0
for(i = 0; i < N; i++)
{  // (∀k)(hist[k] = |{j|0≤j<i ∧ bin(in[j])=k}|)
  int k = bin(in[i]);
  hist[k]++;
}  // (∀k)(hist[k] = |{j|0≤j<N ∧ bin(in[j])=k}|)
```

Don’t try this in parallel with a pure functional language!

Histogramming in parallel

```
const double in[N];      //input to be histogrammed
const int bin(double);   //the binning function
int hist[M] = {0};       //histogram, initially 0
forall i in 0..N-1
{   // (∀k)(hist[k] = |{j|j∈Σ ∧ bin(in[j])=k}|)
  int k = bin(in[i]);
  lock hist[k];
  hist[k]++;
  unlock hist[k];
}   // (∀k)(hist[k] = |{j|0≤j<N ∧ bin(in[j])=k}|)
```

- **Σ** is the nondeterministic set of values i processed "so far"
- The loop instances commute with respect to the invariant
- Premature reads of **`hist[]`** get non-deterministic "garbage"

Abstracting isolation

```
const double in[N];    //data to be histogrammed
const int bin(double); //the binning function
int hist[M] = {0};     //histogram, initially 0
forall i in 0..N-1
  atomic
  {
      int k = bin(in[i]);
      hist[k]++;
  }
```

- The abstraction permits compiler optimization
  - There may be several alternative implementations
- The word “atomic” may be misleading here
  - Does it also mean isolated?
  - Does it isolate the call to **`bin`**? The **`forall`**?

Language axioms based on invariants

- In “Hoare logic” for serial programs, we have axioms like

$$\frac{\{B \wedge I\}\ \mathsf{S}\ \{I\}}{\{I\}\ \text{while}\quad \mathsf{B}\ \text{do}\ \mathsf{S}\ \{\neg B \wedge I\}}$$

- In the parallel proof logic of Owicki and Gries we write

$$\frac{\{I\}\mathsf{S}_1\{I\} \wedge \{I\}\mathsf{S}_2\{I\} \wedge \ldots \wedge \{I\}\mathsf{S}_k\{I\}}{\{I\}\ \text{cobegin}\quad \mathsf{S}_1 \parallel \mathsf{S}_2 \parallel \ldots \parallel \mathsf{S}_k\ \text{coend}\ \{I\}}$$

- For the forall example shown previously, we might write

$$\frac{\{I_\Sigma \wedge \mathsf{i} \notin \Sigma\}\ \mathsf{S_i}\ \{I_{\Sigma \cup \{\mathsf{i}\}}\}}{\{I_\phi\}\ \text{forall}\quad \mathsf{i}\ \text{in}\ \mathsf{D}\ \text{do}\ \mathsf{S_i}\ \{I_\mathsf{D}\}}$$

- where I_X is a predicate on the set X and **`i`** is free in $\mathbf{S_i}$

Examples

- Data bases and operating systems mutate state in parallel
- Data bases use transactions to achieve consistency via atomicity and isolation
  - SQL programming is pretty simple
  - SQL is unfortunately not a general-purpose language
- Operating systems use locks to provide isolation
  - Failure atomicity is usually left to the OS programmer
  - Deadlock is avoided by controlling lock acquisition order
- A general purpose parallel programming language should be able to handle applications like these easily

Implementing isolation

- Analyzing
  - Proving concurrent state updates are disjoint in space or time
- Locking
  - while handling deadlock, *e.g.* with lock monotonicity
- Buffering
  - An “optimistic” scheme, often used for wait-free updates
- Partitioning
  - Partitions can be dynamic, *e.g.* as in **`quicksort`**
- Serializing

These schemes can be nested, *e.g.* serializing access to shared mutable state within each block of a partition

Isolation in existing languages

- Statically, in space
  - MPI, Erlang
- Dynamically, in space
  - Refined C, Jade
- Statically, in time
  - Serial execution
- Dynamically, in time
  - Single global lock
- Statically, in both space and time
  - Dependence analysis
- Semi-statically, in both space and time
  - Inspector-executor model
- Dynamically, in both space and time
  - Multiple locks

Atomicity

- Atomicity just means “all or nothing” execution
  - If something goes wrong, all state changes must be undone
- Isolation without atomicity isn’t worth too much
  - But atomicity is invaluable even in the serial case
- Implementation techniques:
  - Compensating, *i.e.* reversing the computation “in place”
  - Logging, *i.e.* remembering and restoring original state values
- Atomicity is challenging for distributed computing and I/O

Transactional memory

- “Transactional memory” means isolation and atomicity for arbitrary memory references within **`atomic{}`** blocks
  - There are a few difficulties adapting it to existing languages
  - TM is a hot topic these days
- There is much compiler optimization work to be done
  - to make atomicity and isolation as efficient as possible
- Meanwhile, we shouldn’t give up on other abstractions

Multi-object transactions

```
node *m;   //a node to be removed from a graph
...      // (∀m)(∀n)(n∈(m.nbr)⁺ ⇔ m∈(n.nbr)⁺)
atomic {
  for(n = m.nbr, n != NULL, n = n.nbr) {
    //remove link from n to m
    for (p = n.nbr, p != NULL, ... //etc.
  }
}
```

- Care is required when the definition of the invariant's domain depends on bindings within it, *e.g.* **`m->nbr`**
- In naive implementations, deadlock could be commonplace
- If a sequence would deadlock (*i.e.* fail), preservation of the invariant demands it be "undone", reversing its side effects

Where do the invariants come from?

- Can a compiler generate invariants from code?
  - Only sometimes, and it is quite difficult even then
- Can a compiler generate code from invariants?
  - Is this the same as intentional programming?
- Can we write invariants plus code and let the compiler make sure that the invariants are preserved by the code?
  - This is much easier, but may be less attractive
  - See Shankar and Bodik, 2007 PLDI
- Can a programming language/paradigm make it less likely that a transaction omits part of an invariant's domain?
  - *E.g.* objects with encapsulated state
- Can we at least debug our mistakes?
  - The debugger should see consistent state modulo breakpoints

Other language ideas

- Efficient exploitation of nested parallelism
  - NESL, Ct, Data-Parallel Haskell
- Parallel divide-and-conquer
  - Cilk, TPL
- Parallel speculation
  - How can mis-speculated work be stopped and deleted?
- Parallel non-procedural programming
  - Logic programming, for example
  - This is an abundant source of speculative work

Speculation deserves a closer look…

Speculation in logic programming

- A parallel logic program is a set of guarded Horn clauses*:
  $H \leftarrow T_1 \wedge \ldots \wedge T_k ; T_{k+1} \wedge \ldots \wedge T_m$
- If the *head* H unifies with some goal term in progress and all the *guard* terms $T_1 \ldots T_k$ are true, the clause *commits*
  - If not, the unification is undone and the clause aborts
- "AND-parallelism" is concurrent evaluation of some T_i
  - All the guard terms can be done together, then the others
- "OR-parallelism" is concurrent evaluation of clauses
  - Exploiting it requires speculative parallel computation
  - Internal state is updated to reflect new bindings
  - It has historically been a problem to implement efficiently
- Can fine-grain isolation and atomicity tame OR-parallelism?

*My syntax is meant to be pedagogic and is decidedly non-standard!

Dealing with mis-speculation

- There are two problems to solve:
  - Naming mis-speculated tasks
  - Killing mis-speculated tasks
- Killing tasks that haven’t actually started yet is easiest
  - All tasks should be dynamically scheduled anyway
  - To support killing without pre-emption, the compiler could break tasks up into moderate-sized chunks
  - Hardware help might make this even easier
- Nested hypotheses make an (abstract) tree of names
  - Potentially, there are tasks scattered all over it
- The language needs a “kill subtree” function
  - If task descriptions are first-class, the programmer might be able to customize the deletion machinery

Conclusions

- Functional languages equipped with isolation and atomicity can be used to write parallel programs at a high level
  - Microsoft is moving in this direction, *e.g.* with Plinq and F#
- Optimization ideas for isolation and atomicity are needed
  - As well as for parallelism packaging and locality
- We trust architecture will ultimately support these things
  - They are already “hot topics” in the academic community
- The von Neumann model needs replacing, and soon