The State of Parallel Programming

Burton Smith
Technical Fellow
Microsoft Corporation
The von Neumann Premise

“Instructions are executed one at a time...”

• We have relied on this premise for about 60 years
• Now it (and some things it brought) must change
  – Serial language programs schedule values into variables
  – Parallel execution makes this scheme hazardous
• Serial programming is easier than parallel programming, at least at the moment
  – But serial programs are quickly becoming slow programs
• We need parallel programming paradigms that will make everyone who writes programs successful
• The stakes for our field’s vitality are high

Programming must be reinvented
Parallel Programming Successes

• There have been two promising techniques:
  – Isolated memory updates
  – Functional programming
• Neither is completely satisfactory by itself
  – Isolated memory still needs communication and synchronization
  – Functional languages need parallel mutable state updates
• Data bases show the synergy between the two ideas
  – SQL is a “mostly functional” language
  – Transactions implement dynamic isolation of updates
Functional Programming

• Expressions on immutable values are scheduled and executed in dependence order
• There are language variants to suit any taste:
  – Strict or lenient or hybrids of these
  – Higher order functions or not
  – Declarative (e.g. comprehensions) or imperative
• Functional languages can also be efficient
  – Both Sisal and NESL competed well with Fortran on Crays
  – SQL and Excel can and do execute in parallel
• We must make our languages more functional
  – In doing so we must also make them more accessible
Maintaining Invariants

• Serial programming lets us perturb and then restore invariants in local (lexical) fashion
  – “Hoare Logic” depends on this property
  – It’s mostly automatic once we have learned to program
• State updates must be non-interfering
  – That is, they must be isolated in some fashion
  – Otherwise, “too many cooks” would spoil the dish
• And we must finish every update we start
  – lest the next update see the invariant false
  – Therefore the state updates must be atomic
Commutativity (AKA Determinism)

- If statements $p$ and $q$ preserve invariant $I$ and do not “interfere”, then their parallel composition $\{ p \mid \mid q \}$ also preserves $I$.
- If $p$ and $q$ are performed in isolation and atomically, i.e. as transactions, then they will not interfere.
- Operations may 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 we called it “good nondeterminism”
  - It’s the kind of determinism operating systems rely on.

Implementing Isolation

• Analyzing
  – Proving concurrent updates are disjoint in space or time

• Locking
  – Meanwhile handling deadlock in some way

• Copying
  – Often used for wait-free updates

• Partitioning
  – The partitions can be dynamic

• Serializing
  These schemes are often combined, e.g. serializing access to shared mutable state within each block of a partitioned memory
Implementing Atomicity

• Atomicity means “all or nothing” execution
  – If something goes wrong, all state changes must be undone
• Isolation without atomicity isn’t worth much
  – But atomicity is usually crucial even in the serial case
• Techniques:
  – Deferring state changes until the atomic action “commits”
  – Logging, *i.e.* remembering and restoring original state values
  – Compensating, *i.e.* reversing the computation “in place”
• *Transactional Memory* means lexically scoped atomicity and isolation for arbitrary memory references
  – TM is a hot topic these days
• There is a lot of compiler optimization work ahead
  – to make atomicity and isolation as efficient as possible
Styles of Parallelism

• We need to support a mix of programming ideas
  – A combination of functional and transactional styles
  – A combination of data and task parallelism
  – A combination of message passing and shared memory
  – A combination of declarative and imperative specification
  – A combination of implicit and explicit syntax

• We may need several languages to accomplish this
  – After all, we use multiple languages today
  – Language interoperability (e.g. via .NET) will help greatly

• Parallelism and locality must be abstract but visible
  – So that compilers can adapt code to many target systems
Implementing Parallelism

• Exploitable parallelism grows as task granularity shrinks
  – But dependences among tasks become more numerous
• Dependence enforcement needs blocking synchronization
  – A task needing a value from another must wait for it
• Today’s OSes and hardware don’t encourage blocking
  – OS thread preemption, *etc.* makes blocking dangerous
  – Instruction sets also encourage non-blocking behavior
• User-level work scheduling and synchronization are needed
  – No privilege change to stop or restart a subcomputation
  – Locality (*e.g.* cache content) can be better preserved
A Fine-Grain Example: Speech

Today’s Speech Stack:
• Many heterogeneous stages
• Modest parallelism in each stage
• No help from threads and locks

Tomorrow’s Speech Stack:
• 2D or 3D FFTs
• Better model recognition
• Integration with semantics

~5000 Tri-phone HMMs

Word HMMs, etc.
Synchronizing Streaming

• Streaming in parallel needs *producer-consumer* synchronization
  – Barriers wind up serializing the stages
• Basically, this implies FIFOs in memory
• One-item FIFOs are especially useful
  – This is just “full/empty bit” synchronization
  – We can avoid adding extra bits to existing hardware
• Producers and consumers can share caches
  – Both bandwidth and latency are thereby improved
Parallel Debugging and Tuning

• Today, debugging relies on single-stepping and `printf()`
  – Single-stepping a parallel program is much less effective
• Conditional program and data breakpoints are helpful
  – To stop when an invariant fails to be true
• Support for ad-hoc data perusal is also very important
  – Debugging is a form of data mining
• Serial program tuning tries to discover where the program counter spends its time
  – The answer is usually found by sampling the PC
• In contrast, parallel program tuning tries to discover where there is insufficient parallelism
  – A good way is to log resource consumption counters and a system-consistent timestamp at the key events
• Visualization is a big deal for both debugging and tuning
Conclusions

• We must now rethink the basics of programming
• There is interesting work for everyone
• New opportunities are likely to emerge