Reinventing Computing

Burton Smith
Microsoft
Times are changing

Intel CPU Trends
(sources: Intel, Wikipedia, K. Olukotun)
Thread parallelism is upon us†

- Uniprocessor performance is leveling off
  - Instruction-level parallelism is nearing its limit
  - Power per chip is painfully high for client systems
- Meanwhile, logic cost ($ per gate-Hz) continues to fall
  - What are we going to do with all that hardware?
- Newer microprocessors are multi-core and/or multithreaded
  - So far, it’s just “more of the same” architecturally
- We expect new “killer apps” will need more performance
  - Semantic analysis and query
  - Improved human-computer interfaces (e.g. speech, vision)
  - Games!
- Which and how much thread parallelism can we exploit?
  - This is a good question for both hardware and software

†And it’s high time!
Limits to instruction-level parallelism

- There have been two prominent approaches to ILP:
  - Vector instructions, including MMX and the like
  - The HPS\(^\dagger\) canon: out-of-order issue, in-order retirement, register renaming, branch prediction, speculation, …
- Neither scheme generates much concurrency given a lot of:
  - Control-dependent computation
  - Data-dependent memory addressing (e.g. pointer-chasing)
- In practice, we are limited to one or two instructions/clock
  - If you doubt this, ask your neighborhood computer architect

Power and performance

- Two ways to scale multiprocessor speed by a factor $\sigma$:
  - Scale the number of running processors by $\sigma$
    - Power will scale by $\sigma$
  - Scale the clock frequency by $\sigma$
    - For a fixed process, voltage will scale by the same amount
    - Dynamic power will scale by $\sigma^3 \left( \frac{1}{2} CV^2f \right)$
    - Static power will scale by $\sigma \left( V_{\text{leakage}} \right)$
    - Total power lies somewhere in between
- Clock scaling is worse when $\sigma > 1$
  - This is part of the reason times are changing
- Clock scaling is better when $\sigma < 1$
  - Moral: if your multiprocessor is fully used but too hot, scale down voltage and frequency rather than processors
Power aside, can faster clocks save us?

- Clock rate scales inversely with feature size
- Transistor count scales inversely with feature size squared
- So a uniprocessor faces both a problem and a solution
  - Cache miss latencies (measured in clocks) get longer
  - Bigger caches are obviously where the transistors need to go
- To cut average latency (in clocks) in half given the same concurrency, bandwidth (in bits/clock) must also be halved
- How much bigger does the cache have to be†?
  - For dense matrix-matrix multiply or dense LU, 4x bigger
  - For sorting or FFTs, the square of its former size
  - For sparse or dense matrix-vector multiply, forget it
  - Your mileage will definitely vary

**Latency, Bandwidth, & Concurrency**

- In any system that transports items from input to output without creating or destroying them,
  
  \[ \text{latency} \times \text{bandwidth} = \text{concurrency} \]

- Queueing theory calls this result *Little’s Law* 

- \( \text{bandwidth} = 2 \)

- \( \text{concurrency} = 6 \)

- \( \text{latency} = 3 \)
The von Neumann model

- We have relied on it for some 60 years
- Now it (and some stuff it brought along) must change
  - Serial execution lets programs *schedule values into variables*
  - Parallel execution makes this scheme hazardous
- Serial programming is easier than parallel programming
  - Trouble is, serial programs are now 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
- Computing must be reinvented
How did we get into this fix?

- Microprocessors kept getting faster at a pretty decent pace
  - Better than 1000-fold in the last 20 years
- HPC was drawn into a spiral of specialization
  - “HPC applications are those things HPC systems do well”
  - DARPA HPCS goals are partly a reaction to this tendency
- University research on parallel systems has dried up
  - No interest?
  - No money?
  - No need?
Research is (still) needed

• Languages for mainstream parallel computing
• Compilation techniques for parallel programs
• Debugging and performance tuning of parallel programs
• Operating systems for parallel computing at all scales
• Computer architecture for mainstream parallel computing
Parallel languages

- A diversity of styles is needed, often in a single program
  - Both data parallelism and task parallelism
  - Both shared memory and message passing
  - Both imperative and declarative
  - Both functional and transactional
- It’s essential that parallelism be exposed to the compiler
- It’s essential that locality be exposed to the compiler
- Generality would be nice
- Microsoft needs a good answer to the question:
  “What programming language should we use to code the version of Microsoft Office® we plan to ship in 2010?”
Language questions

- How can message passing be enhanced linguistically?
  - Richer type signatures?
  - Protocols and contracts?
  - Distributed transactions?
- What are the best ways to express data parallelism?
  - Map, reduce, scan, and the like?
  - Foralls on enumerable domains?
  - Divide-and-conquer via recursion?
- How can locality be expressed well at a high level?
- Are atomic transactions what functional languages need to become both general purpose and parallel? (LACSI ’01)
- Can we expect programmers to supply invariants?
  - What if the invariants supplanted imperative stuff?
Parallel compilers

- There is a claim going around that compilers won’t help
  - “Automatic parallelization is a demonstrated failure!”
- That canard won’t fly
  - Vectorizing and even parallelizing compilers (for the right architecture) have been a tremendous success
  - They have enabled machine-independent languages
  - What they do can be termed parallelism packaging
  - Even manifestly parallel programs need it
- What failed is parallelism discovery, especially in-the-large
  - Dependence analysis is principally a local success
- Locality discovery in-the-large has also been a non-starter
  - Locality analysis is another word for dependence analysis
  - The jury is still out on large-scale locality packaging
Compiler questions

- How can compilers best accomplish parallelism packaging for varying numbers of heterogeneous processors?
- How can locality be effectively packaged?
  - Can we exploit cache sharing in both space and time?
- Can garbage collection be made to scale well enough?
Parallel debugging and tuning

- Nondeterministic execution is a fact of multithreaded life
  - The trick is to avoid nonreterministic answers
  - This is basically an exercise in invariant preservation
    - Invariant preservation is a generalization of commutativity
  - Some race detection and invariant checking tools exist
- Debugging is now based on single-stepping and `printf()`
  - Single-stepping a parallel program is less effective
- It is likely that debugging and testing will be very expensive unless we aggressively pursue these problems
- Parallel tuning of shared-memory systems is primitive
  - We are in better shape for messaging between address spaces
  - Part of the problem is performance-model transparency
Debugging and tuning questions

- How can nondeterministic bugs (races) be discovered?
  - What about “high-order data races” (invariant violations)?
- Can debugging be liberated from the von Neumann style?
  - Should data breakpoints replace program breakpoints?
- How can we peruse program state in more ad-hoc ways?
  - Why is \texttt{printf()} still king?
- What is needed to diagnose parallelism bottlenecks?
  - What new instrumentation is needed?
  - How should performance-diagnostic data be gathered?
  - How should the data be presented to the user?
- Is parallel performance tuning even possible without the processors having a better idea of what time it is?
Parallel operating systems

- Operating systems must stop trying to schedule threads
  - They can certainly keep allocating processors, though
- Threads should be scheduled at user level
  - There’s no need for a change of privilege
  - Optimization becomes much more possible
  - Blocked thread state can be first-class
- Demand paging is a bad idea for most parallel applications
  - Everything ends up waiting on the faulting computation
  - An isoefficiency premise of demand paging is false here
Operating system questions

- How should processors be allocated to competing processes?
  - The OS giveth and the OS taketh away preemptively?
  - The process requests and relinquishes processors?
- How should memory be allocated to competing processes?
- What should a file system intended for use by parallel applications look like?
Parallel hardware

- Hardware has had a good-sized head start at parallelism
  - That doesn’t mean it’s way ahead!
- Artifacts of the single program counter assumption abound
  - Interrupts, for example
  - Most of these are easy to fix
- A bigger issue is support for fine-grain parallelism
  - Thread granularity depends on the amount of state per thread and on how much it costs to swap it when the thread blocks
- Another is whether all processors should look the same
  - There are several constructive options for heterogeneity
  - Not all of them are easy to integrate into a single program
  - Still fewer are easy compiler targets
- The biggest issue may be how to maintain system balance
**Hardware questions**

- What are profitable directions for processor heterogeneity?
- How should transactional memory be implemented?
- How should synchronization be enhanced?
- How can primary storage bandwidth be kept in balance?
- How can secondary storage bandwidth be kept in balance?
- How should I/O architecture be changed?
Conclusions

- It is time we rethought some of the basics of computing
- There is lots of work for everyone to do
  - I’ve left people out, especially in applications
- It’s scary, and lots of fun at the same time