Universal Parallel Computing Research Center at Illinois

Making parallel programming synonymous with programming

Marc Snir

08-09
The UPCRC@ Illinois Team
Moore’s Law Pre 2004

- Number of transistors per chip doubles every 18 months
- Performance of single thread increases
- New generation hardware provides better user experience on existing applications or support new applications that cannot run on old hardware
- People buy new PC every three years
Moore’s Law Post 2004

• Number of transistors per chip doubles every 18 months
• Thread performance does not improve; number of cores per chip doubles
  – power/clock constraints & diminishing returns on new microprocessor features
• New generation hardware provides better user experience on existing application or support new applications that cannot run on old hardware – only if these applications run in parallel & scale automatically
• Parallel Software is essential to maintaining the current business model of chip & system vendors
Goals

- **Create opportunity**: New *client* applications that require high performance and can leverage high levels of parallelism
- **Create SW to exploit opportunity**: Languages, tools, environments, processes that enable the large number of client application programmers to develop good parallel code
- **Create HW to exploit opportunity**: Architectures that can scale to 100’s of cores and provide best use of silicon a decade from now
UPCRC Illinois Activities

Compute intensive client applications:
- *Human-Computer Intelligent Interfaces*

Parallel Programming Environments:
- Programming for the masses:
  - *Concurrency safe programming languages*
  - Refactoring tools
  - Testing tools for unsafe languages
- Programming for top performance
  - Parallel libraries
  - Interactive tuning
  - Autotuning

Scalable Architectures:
- *Scalable coherence protocols*
- Architecture support for disciplined programming
- 1000 cores and beyond

Patterns

- Use new PPE to develop kernels and libraries for apps
- Provide HW support for PPE
- Leverage safe, disciplined languages for shared memory scalability

Educate parallel programmers

Codify main practices

Test expressiveness of PPE

Leverage new PPE to develop kernels and libraries for apps

Educate parallel programmers

Codify main practices

Test expressiveness of PPE
• Create Opportunity

APPLICATIONS
Applications Strategy

• **Identify application types that**
  – are likely to execute on clients
  – require much more performance than now available on a client
  – can run in parallel

• **Develop enabling parallel code (core libraries, application prototypes) for such application types**
  – hard to identify the killer app; easier to work in its “vicinity”
  – doing so gives us an understanding of the apps requirements; demonstrates feasibility
  – and leads to the creation of reusable software
• **Intelligent user interfaces require high performance on the client side (!)**
  – graphics, vision, NLP

• **Private information will be kept on the client side (?)**
  – concerns for privacy and security
  – fewer efficiencies to be achieved on server side, because of limited sharing
  – NLP, data mining, search

• **High-availability services require client performance and adaptation**
  – Provide “best answer”, given current connectivity
  – Adaptive applications (NLP)

• **More powerful client reduces app development time**
  – Games, browser
Graphics -- Motivation

- Halo3
- GTA4
- WoW
- 2nd Life

Game Engineering
Faster, Cheaper

Multicore
Visual Apps

Online
Social
Spaces
More
Realistic

Realism

(Time, $$)"

(Precomputation)

Flexibility

John Hart
3D Tele-Immersion

Next generation social communication medium
Need for Speed

Performance numbers
(between UIUC & UCB):
93+105+37=235ms

Goal: <150ms

• 3D reconstruction
  – 1280x960xK (K=\#eyes on a 3D camera) pixels to process in a macro-frame on a single PC
  – \(N\) cameras employed: \#\(N\) PCs needed

• 3D rendering
  – \(N\times M\) streams to renderer (\(N\): avg. \#cameras, \(M\): \#sites)
• Simple, race-free, coarse grain parallelism for the masses
• Data parallel libraries for SIMD/GPU performance
• Better testing and refactoring tools for the sequential -> parallel port

PROGRAMMING LANGUAGES
Parallel Programming vs. Concurrent Programming

- **Concurrent programming**: concurrency is part of the application specification (HARD!)
  - reactive code: system code, GUI, OLTP
  - *inherently nondeterministic*: external concurrent interactions
  - focused on concurrency management and synchronization: *mutual exclusion*, *atomic transactions*.

- **Parallel programming**: concurrent execution is introduced to improve performance (EASY?)
  - transformational code, e.g. scientific computing, signal processing
  - *inherently deterministic*: external interactions are sequential
  - focused on the generation of parallelism and on consumer-producer synchronization

- **Multi-core creates significant new demand for parallel programming, but no significant new demand for concurrent programming.**
Parallelism Need Not Be Hard

• Some forms of parallelism are routinely used:
  – vector operations (APL/Fortran90), domain-specific dataflow languages (Cantata/Verilog/Simulink), concurrent object languages (Squeak, Seaside, Croquet, Scratch)…

• Work on shared memory programming has been almost exclusively focused on (hard) concurrent programming

• Investments on SW to support parallel programming have been minuscule and focused on expert programmers and “one-size-fits-all” solutions
What Would Make Parallel Programming Easier?

- **Isolation**: effect of the execution of a module does not depend on other concurrently executing modules.

- **Concurrency safety**: Isolation is enforced by language

- **Determinism**: program execution is, by default, deterministic; nondeterminism, if needed, is introduced via explicit notation.

- **Sequential semantics**: sequential operational model, with simple correspondence between lexical execution state and dynamic execution state

- **Parallel performance model**: work, depth
Why is Determinism Good?

• Testing is much easier (single execution per input)
• Debugging is much easier (linear time)
• Easy to understand: execution equivalent to sequential execution
• Easy to incrementally parallelize code
• Can use current tools and methodologies for program development

• Nondeterminism seldom (if ever) needed for performance parallelism
  – with exceptions such as chaotic relaxation, branch & bound, some parallel graph algorithms
  – when needed, can be highly constrained (limited number of nondeterministic choices)
Parallelism mostly comes from parallel loops – we focus discussion on loops, for simplicity

- **Implicit parallelism (C):** Write sequential (for) loops and hope the compiler parallelizes
  
  Always safe, seldom efficient
  
  ✗ Often fails (compiler lacks information that user has)
  
  ~ Performance model is defined after compilation (compiler can report which loops parallelize) – but is not defined by source code and is compiler dependent

- **Explicit parallelism (OpenMP):** Write parallel (forall) loops and force compiler to parallelize
  
  Usually efficient, never safe
  
  ✓ Clear performance model

  ✗ Unsafe – races are not detected or prevented
Current State of the Art (2)

• **Speculative parallelism (C):** Write sequential loops and have compiler parallelize speculatively
  × Not efficient (without HW support)
  × Performance model unclear
  × Hard to catch, during development, “performance bugs”

• **Functional parallelism (NESL, Haskell):** Disallow mutable variables
  ✓ Clear performance model (see NESL)
  × Not efficient (esp. w.r.t. memory use)
  × Hard (Impossible?) to express certain parallel algorithms – Need to support shared references to mutable objects (graph, array)
Goal: Safe Parallelism

• Programmer provides additional assertions on effects of concurrent tasks (read and write sets)
• Compiler *enforces* assertions *at run-time* and *uses* them *at compile-time* to prove that parallel execution is safe
• Should work for all/most loops that have no races, without undue programming effort
• Should be cheap to enforce *at run-time* (ideally, free)
  – much cheaper than enforcing the “race-free” assertion, i.e. detecting races at run-time
  – will often be subsumed by checks to enforce type safety, memory safety, etc.
• Fallback position: May trust assertions about “native” routines from trusted developers
Implicit vs. Explicit Parallelism

• **Explicit safe parallelism**
  – `forall` construct is annotated; compiler generates error if iterates cannot be proven to be independent
  ✓ proof mechanism is defined by language semantics

• **Implicit safe parallelism**
  – `for` construct is annotated; compiler lets user know whether loop parallelizes
  ✗ proof mechanism is defined by compiler technology

• **Same technology in both cases; different pragmatic choices**
  – clear, compiler independent performance model vs. more continuity with current languages and more flexibility in advancing compiler analysis
• How do coherence protocols scale and provide better support for current software? (Bulk -- J. Torrellas)
• How do we take advantage of and better support new parallel languages? (DeNovo -- S. Adve)
• How do we scale to >1000 cores? (Rigel -- S Patel)

ARCHITECTURE
Fundamental Issues

• Coherence protocols handle accesses to each memory location individually – at great expense. Codes are written using “bulk transactions” that read or write sets of variables; can we take advantage of this?
  – Chunk code execution adaptively (Bulk)
  – Use information on synchronization operations (DeNovo)
  – Expose check-in/check-out interface to user (Rigel)

• Architecture work pays attention to mutual exclusion, but not to producer-consumer
The Bulk Multicore

General-purpose hardware architecture for programmability

- Novel scalable cache-coherent shared-memory (signatures & chunks)
  - Relieves programmer/runtime from managing shared data
- High-performance sequential memory consistency
  - Provides a more SW-friendly environment
- HW primitives for low-overhead program development & debug (data-race detection, deterministic replay, address disambiguation)
  - Helps reduce the chance of parallel programming errors
  - Overhead low enough to be “on” during production runs

http://iacoma.cs.uiuc.edu/bulkmulticore.pdf
Idea in Bulk Multicore

• Idea: Eliminate the commit of individual instructions at a time

• Mechanism:
  – By default, processors commit chunks of instructions at a time (e.g. 2,000 dynamic instr)
  – Chunks execute atomically and in isolation (using buffering and undo)
  – Memory effects of chunks summarized in HW address signatures

• Advantages over current:
  – Higher programmability
  – Higher performance
  – Simpler processor hardware

[CACM 2009]
Application: HW+SW for Deterministic Replay

• **Goal:** Support deterministic replay of parallel programs with minimal recording overhead and tiny logging requirements

• **Results:**
  – By using the Bulk hardware, only need to record the interleaving of the chunks. Reduced the log size requirements by over 2 orders of magnitude [DeLorean in ISCA 2008]
  – Extended Linux to have multiple Replay Spheres, enabling virtualization of the recording and replay hardware [Capo in ASPLOS 2009]
Application: HW Support for Data Race Detection

• **Goal:** Use hardware to detect data races dynamically in production run codes with very low overhead

• **Results:**
  
  – Processors automatically collect the addresses accessed in hardware signatures. An on-chip hardware module intersects the signatures in the background and identifies races.

  – Effective race detection with only 20% execution overhead

[SigRace in ISCA 2009]
DeNovo Architecture

Rethinking hardware with disciplined parallelism

• **Hypothesis 1:** Future hardware will require disciplined parallel models for
  – Scalability
  – Energy efficiency
  – Correctness (Verifiability, testability, …)

• **Hypothesis 2:** Hardware/runtime support can make disciplined models more viable
  – How do disciplined models affect hardware (& runtime)?
  – Rethink hardware from the ground up
    • Concurrency model, coherence, tasks, …
  – Co-design hardware & language concurrency models

**Goal:** Hardware that is
Scalable
Performing
Energy-efficient
Easy to design
Opportunities for Hardware

• **Disciplined software allows optimizing**
  – Communication fabric
  – Memory model and semantics
  – Task scheduling and resource management

• **Goal**
  – Unprecedented scalability and energy efficiency
Some Key Ideas

• Exploit from software
  – Structured control; region/effects; non-interference

• Communicate only the right data to the right core at the right time
  – Eliminates unscalable directory sharing lists, complex protocol races, performance thwarting indirections
  – Enables latency-, bandwidth-, and energy-efficient data transfers
  – No false sharing, efficient prefetching and producer-initiated communication
Ongoing and Future Work

• Simulation prototype of DeNovo architecture
• Broadening supported software
  – Unstructured synchronization and speculation
  – Legacy codes
• Runtime support for disciplined languages
  – Speculation, sandboxing, contract verification, …
• Virtual typed hardware/software interface
  – Language-, platform-independent virtual ISA
Architectural Framework: The Rigel Architecture

- Non-HW coherent caches
- Area-efficient core design
- Primitive support for scalable synchronization and reductions
- Cache management support for locality enhancement
- Compiler, simulator, RTL all available now
Mapping 1000 cores to 45nm

- 1024 cores, 8MB Cluster Cache, 4MB Global Cache (~3 TOps/sec)
- Synthesized Verilog @45nm for cores, cluster cache logic
- SRAM Compiler for SRAM banks
- Other Logic: interconnect, mem controllers, global cache
- Typical power ~70-99W
Questions to be addressed

• Programming models that scale from 1000 chips in a cluster to 1000 cores in a chip
• Run-time systems for scalable work distribution
• Locality management, architectural optimizations for memory bandwidth
• SIMD efficiency versus MIMD flexibility
• Power/energy optimizations for throughput oriented architectures
Summary

• Parallelism need not be hard
  – much easier than traditional concurrent programming

• Parallel programming, like programming, is a team effort that requires many different skills and many different tools
  – coarse-level parallelism for the masses, tuned libraries, refactoring tools, verification and tuning tools

• Parallel architectures can scale if they take advantage of practical constraints on communication and synchronization in real programs