Top Five Reasons

Why sequential programming models could be the best way to program many-core systems

Wen-mei Hwu
University of Illinois, Urbana-Champaign
Preface

• My first MICRO:
  – “The 18th Annual Workshop on Microprogramming”

• MICRO-24, From the Program Chairs:
  – “For the first time in twenty years, the word microprogramming does not appear in the title of this conference” – Wolfe, Nakatani, Mulder
  – MICRO is now the premiere forum on instruction-level microarchitecture and compiler work.

What about the next 15 years?
The Next Software Challenge

• Today, TLP models potentially make more effective use of area and power than large ILP CPU’s
  – Multi-core, Many-core, Cell BE™, GPGPU, PPU, etc.
  – Scaling from 4-core to 1000-core chips could happen in the next 15 years

• All semiconductor market domains converging to concurrent system platforms
  – PCs, game consoles, mobile handsets, servers, supercomputers, networking, etc.

We need to make these systems effectively execute valuable, demanding apps.
Is this new?

"Most high-end application developers are painfully aware of the tricks needed to explicitly manage memory hierarchies and communication in today's parallel systems."

Human vs. Machine-level Programming Models

- Pentium
- Itanium
- NVIDIA GPU

- Parallelizing Compiler Domain API
- Domain API

- Hardware
- Hardware
- Hardware

MPI, OpenMP
One Plausible View

• One might be able to stretch sequential programming models to small scale multi-core systems.
• As for large scale, many-core systems, explicitly parallel programming models must be used.

I would like to convince you that it is the other way around.
One Important Disclaimer

Sequential Programming Model ≠ Legacy sequential code

• Code change will likely be needed whether we choose sequential or parallel programming model.
  – Legacy code was not developed to give tools deep knowledge about the computation being performed.
  – E.g., source code does not express high-level properties and assumptions
Another Important Disclaimer

Sequential Programming Model ≠ Sequential Algorithms

- When there is no parallelism in the algorithm, all is lost.
  - The question is how to go from parallel algorithms to parallel machine code
Attacking from two ends

- Traditional GP compilers cover traditional apps (pit)
- Parallel programming models and OpenMP, MPI cover simpler, earlier super-apps (peach skin)
- Concurrent Systems to cover new super-apps
  - Reflective of real-world, abundant parallelism but complex in code and data structures (peach flesh)
Now, the list...
Reason #5

- Developing a non-trivial, explicitly parallel application is an expensive proposition
  - Understanding and performing all optimizations needed is hard, transactional memory might help.
  - Optimized parallelization will likely involve finessing libraries and infrastructure software.
  - Data structure and pointer marshalling/finessing
  - The larger the number of cores, the less intuitive the optimization process will be.
Parallelization models – MPEG-4

Operations performed on 16x16 macroblocks
- Red: Motion Estimation
- Green: Motion Compensation, Frame Subtraction
- Blue: DCT & Quantization
- Orange: Dequantization, IDCT, Frame Addition
- Grey: Main Memory Access

Need solution space exploration.
NAMD

- Smaller code, comparable data structure complexity, simpler forms of parallelism
- Many people year

December 11, 2006  MICRO-39 Keynote
Traditional Sequential Programming Model

- Tools and HW know little about the computation being executed
- Programmers know little about the reality of parallel execution
- OOO perfect example, VLIW slight deviation where tools know more
Traditional Parallel Programming Model

Applications (large number)

Transistors (very large number)

- Tools know little about the computation being executed, creating tremendous pressure for programmers to mutate the source code for performance.
- Programmers asked to manage details of parallel execution and are assumed to know enough to do so well.
A New Sequential Programming Model

- Tools have in-depth knowledge about the computation being executed.
- Programmers write "analyzable" source code and assert assumptions and critical properties of code, most likely at function call interfaces.
Reason #4

- Verification and support of a hand parallelized program is even more expensive
  - Application-level testing needs to cover a much larger state space
  - Reproducing bugs can be very challenging, tools/lifeguards might help.
  - Should contain the verification complexity through guarantees at the interface level.

Reality of Relative Cost and Importance

- Any value from lower triangle creates 10x leverage
- App developer test sequential code and tool developers / HW guarantee correctness of parallelization
- The cost and complexity in the tool stack can be drastically reduced with coding styles and programmer assertions
Reason #3

• Programs developed in explicitly parallel form will unlikely scale with Moore’s Law
  – Performance comes from optimizations that craft the computation and data accesses to best get around limitations
  – Every new generation or new platform will likely require app developers to redo their application
Reason #2

• Tools must have in-depth knowledge of the apps regardless of programming models being used anyway
  – Most developers will need tools to restructure their data structures and computation even in explicitly parallel code.
  – Why not just take it one more step further?
Reason #1

- Evolution of tools take place at much faster pace than that of human race.
  - Although both have shown lack luster results in the past thirty years, I would bet on tool changes much more readily than human changes.
FAQ

• Is there any worthwhile research problem left in automatic parallelizing tools?
• Is there any evidence that automatic parallel programming tools will ever work?
• Where is the human in the picture?
• Why is the current work any different from what people did in the past 30 years?
• Can anyone make any money out of parallel programming tools?
• Are you saying that explicitly parallel programming efforts should not be made?
• Is the compiler going to work on production software with real build systems?
Now, Some Opportunities …
Microarchitecture Circa 2016

- High-Frequency, Wide Issue, Speculative, Yale Patt Processor
- Massively Parallel, Modest frequency Multi-threaded HEP-style Processor
- Low-power MT processor
- On-chip Network Fabric
- Sequential Programming Box
- HW accelerators
- Memory and I/O System

December 11, 2006 MICRO-39 Keynote
Anatomy of a Tough App

Main Body
- custom memory management
- pointer arithmetic
- implicit data structure assumptions
- parallel vs. sequential semantics
- ...

API and Lib
- source missing

Complex
- Build System

December 11, 2006
A Closer Look at a One App

MPEG-4 H.263 encoder

(a) Original: Affine expression intraprocedural array disambiguation analysis with a context-insensitive, field-insensitive pointer analysis
Progress is not linear!

(b) Original + interprocedural array disambiguation
(c) Original + non-affine expression array disambiguation
(d) Original + heap-sensitive, context-sensitive pointer analysis
(e) Original + field-sensitive pointer analysis
Compiler-visible parallelism - combined pointer analysis options

(f) Combination #1: Original + interprocedural array disambiguation + context- & heap-sensitive pointer analysis

(g) Combination #2: Combination #1 + non-affine expression array disambiguation

(h) Combination #3: Combination #2 + field-sensitive pointer analysis

(i) Final: Combination #3 with value constraint and relationship inference analyses
Deep analysis can be scalable

Exhaustive Based (Naïve) Problem Size versus Depth

Goes from exponential growth to largely flat

Compaction Based Problem Size versus Depth
Accuracy Vary
(IJPEG call graph resolution)
Conclusion

• Concurrency for the mass is a tough business
• There is a huge amount of research opportunities
• MICRO community is extremely well positioned to drive this movement
• Any questions? – Yale Patt.