New Paradigms for Multicore Programming

Krste Asanovic
The Parallel Computing Laboratory
UC Berkeley

Xilinx Emerging Technology Symposium
San Jose, CA
February 1, 2012
Berkeley researchers from many backgrounds meeting since Feb. 2005 to discuss parallelism

- Krste Asanovic, Eric Brewer, Ras Bodik, Jim Demmel, Kurt Keutzer, John Kubiatowicz, Dave Patterson, Koushik Sen, Kathy Yelick, …
- Circuit design, computer architecture, massively parallel computing, computer-aided design, embedded hardware and software, programming languages, compilers, scientific programming, and numerical analysis
- Tried to learn from successes in high-performance computing (LBNL) and parallel embedded (BWRC)


**Goal:** To enable most programmers to be productive writing efficient, correct, portable SW for 100+ cores & scale as cores increase every 2 years (!)
Past parallel projects often dominated by hardware architecture:

- *This is the one true way to build computers, software must adapt to this breakthrough!*
- E.g., ILLIAC IV, Thinking Machines CM-2, Transputer, Kendall Square KSR-1, Silicon Graphics Origin 2000 …

Or sometimes by programming language:

- *This is the one true way to write programs, hardware must adapt to this breakthrough!*
- E.g., Id, Backus Functional Language FP, Occam, Linda, HPF, Chapel, X10, Fortress …

Applications usually an afterthought
Par Lab’s original “bets”

- Let compelling applications drive research agenda
- Software platform: data center + mobile client
- Identify common programming patterns
- Productivity versus efficiency programmers
- Autotuning and software synthesis
- Build-in correctness + power/performance diagnostics
- OS/Architecture support applications, provide flexible primitives not pre-packaged solutions
- FPGA simulation of new parallel architectures: RAMP
- Co-located integrated collaborative center

Above all, no preconceived big idea - see what works driven by application needs.
Par Lab Timeline

Win UPCRC Competition

“Berkeley View” Techreport

Initial Meetings

UPCRC Phase-I

UPCRC Phase-II

Par Lab End of Project Party!

You are here

“Next” project initial meetings

Talk Outline

- The “big ideas” that emerged from Par Lab
  - Patterns for parallel programming
  - Communication-avoiding algorithms
  - Specializers: Pattern-specific compilers
  - Effective composition of parallel modules

- What’s next?
  - Tackling hardware specialization with patterns
  - Communication-avoiding hardware
  - Hardware+software specializers
  - Effective hardware+software composition
Content-Based Image Retrieval
(Kurt Keutzer)

- Built around Key Characteristics of personal databases
  - Very large number of pictures (>5K)
  - Non-labeled images
  - Many pictures of few people
  - Complex pictures including people, events, places, and objects

Query by example

Image Database

1000’s of images

Relevance Feedback

Similarity Metric

Candidate Results

Final Result
Stroke treatment time-critical, need supercomputer performance in hospital.

Goal: 1.5D Fluid-Solid Interaction analysis of Circle of Willis (3D vessel geometry + 1D blood flow).

Based on existing codes for distributed clusters.
**Parallel Browser**

(Ras Bodik)

- **Readable Layouts**
- **Augmented Reality**

- **Original goal:** Desktop-quality browsing on handhelds (Enabled by 4G networks, better output devices)
- **Now:** Better development environment for new mobile-client applications, merging characteristics of browsers and frameworks (Silverlight, Qt, Android)
New user interfaces with pressure-sensitive multi-touch gestural interfaces

Programmable virtual instrument and audio processing

Audio

120-channel speaker array
Music Software Structure

- GUI Service
- Front-end
- Audio Processing Plug-in
- Filter Plug-in
- Oscillator Bank Plug-in
- Audio Processing & Synthesis Engine
- Network Service
- File Service
- Solid State Drive
- 120-Channel Spherical Speaker Array

Input → Audio Processing → Output

End-to-end Deadline
Speech: Meeting Diarist
(Nelson Morgan, Gerald Friedland, ICSI/UCB)

- Laptops/Handhelds at meeting coordinate to create speaker identified, partially transcribed text diary of meeting
Meeting Diarist Software Architecture

Speech Processing

- Speaker Diarization
- "who spoke when"
- Indexing, Search, Retrieval
- "who said what"
- Question Answering
- higher-level analysis

Audio Signal

- Speech Recognition
- "what was said"
- Relevant Web Scraping
- "what are the main points"

Network Service

- "what's relevant to this"

Browser-Based Interactive GUI

- Marge: We're going to school for the Parent/Teacher meetings. We'll bring dinner home.
- Lisa: What are you going to bring home?
- Homer: Well it depends. If both of you have been good, pizza. If not then poison.
- Lisa: What if one of us has been good and one of us has been bad?
- Bart: Poisoned pizza?
- Homer: Oh no! I'm not making two stops.
### Types of Programming (or “types of programmer”)

<table>
<thead>
<tr>
<th>Domain-Level (No formal CS)</th>
<th>Example Languages</th>
<th>Example Activities</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Max/MSP, SQL, CSS/Flash/Silverlight, Matlab, Excel</td>
<td>Builds app with DSL and/or by customizing app framework</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Productivity-Level</th>
<th>Haskell, Lua, Scala</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Python/Ruby/Langage</td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Efficiency-Level (MS in CS)</th>
<th>Example Languages</th>
<th>Example Activities</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>C/C++/FORTRAN assembler</td>
<td>Uses hardware/OS primitives, builds programming frameworks (or apps)</td>
</tr>
<tr>
<td></td>
<td>Java/C#</td>
<td>Provides hardware primitives and OS services</td>
</tr>
</tbody>
</table>

Where & how to make parallelism visible?
How to make parallelism visible?

- In a new general-purpose parallel language?
  - An oxymoron?
  - Won’t get adopted
  - Most big applications written in >1 language

- Par Lab is betting on Computational and Structural Patterns at all levels of programming (Domain thru Efficiency)
  - Patterns provide a good vocabulary for domain experts
  - Also comprehensible to efficiency-level experts or hardware architects
  - *Lingua franca* between the different levels in Par Lab
Motifs common across applications

App 1
Dense

App 2
Sparse

App 3
Graph Trav.

Berkeley View Motifs ("Dwarfs")
How do compelling apps relate to 12 motifs?

<table>
<thead>
<tr>
<th></th>
<th>Embedded</th>
<th>SPEC</th>
<th>DB</th>
<th>Games</th>
<th>ML</th>
<th>CAD</th>
<th>HPC</th>
<th>Health</th>
<th>Image</th>
<th>Speech</th>
<th>Music</th>
<th>Browser</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>1</strong> Finite State Mach.</td>
<td>🟥</td>
<td>🟥</td>
<td>🟥</td>
<td>🟥</td>
<td>🟥</td>
<td>🟥</td>
<td>🟥</td>
<td>🟥</td>
<td>🟥</td>
<td>🟥</td>
<td>🟥</td>
<td>🟥</td>
</tr>
<tr>
<td><strong>2</strong> Circuits</td>
<td>🟥</td>
<td>🟥</td>
<td>🟥</td>
<td>🟥</td>
<td>🟥</td>
<td>🟥</td>
<td>🟥</td>
<td>🟥</td>
<td>🟥</td>
<td>🟥</td>
<td>🟥</td>
<td>🟥</td>
</tr>
<tr>
<td><strong>3</strong> Graph Algorithms</td>
<td>🟥</td>
<td>🟥</td>
<td>🟥</td>
<td>🟥</td>
<td>🟥</td>
<td>🟥</td>
<td>🟥</td>
<td>🟥</td>
<td>🟥</td>
<td>🟥</td>
<td>🟥</td>
<td>🟥</td>
</tr>
<tr>
<td><strong>4</strong> Structured Grid</td>
<td>🟥</td>
<td>🟥</td>
<td>🟥</td>
<td>🟥</td>
<td>🟥</td>
<td>🟥</td>
<td>🟥</td>
<td>🟥</td>
<td>🟥</td>
<td>🟥</td>
<td>🟥</td>
<td>🟥</td>
</tr>
<tr>
<td><strong>5</strong> Dense Matrix</td>
<td>🟥</td>
<td>🟥</td>
<td>🟥</td>
<td>🟥</td>
<td>🟥</td>
<td>🟥</td>
<td>🟥</td>
<td>🟥</td>
<td>🟥</td>
<td>🟥</td>
<td>🟥</td>
<td>🟥</td>
</tr>
<tr>
<td><strong>6</strong> Sparse Matrix</td>
<td>🟥</td>
<td>🟥</td>
<td>🟥</td>
<td>🟥</td>
<td>🟥</td>
<td>🟥</td>
<td>🟥</td>
<td>🟥</td>
<td>🟥</td>
<td>🟥</td>
<td>🟥</td>
<td>🟥</td>
</tr>
<tr>
<td><strong>7</strong> Spectral (FFT)</td>
<td>🟥</td>
<td>🟥</td>
<td>🟥</td>
<td>🟥</td>
<td>🟥</td>
<td>🟥</td>
<td>🟥</td>
<td>🟥</td>
<td>🟥</td>
<td>🟥</td>
<td>🟥</td>
<td>🟥</td>
</tr>
<tr>
<td><strong>8</strong> Dynamic Prog</td>
<td>🟥</td>
<td>🟥</td>
<td>🟥</td>
<td>🟥</td>
<td>🟥</td>
<td>🟥</td>
<td>🶨</td>
<td>🟥</td>
<td>🟥</td>
<td>🶨</td>
<td>🶨</td>
<td>🶨</td>
</tr>
<tr>
<td><strong>9</strong> Particle Methods</td>
<td>🶨</td>
<td>🶨</td>
<td>🶨</td>
<td>🶨</td>
<td>🶨</td>
<td>🶨</td>
<td>🶨</td>
<td>🶨</td>
<td>🶨</td>
<td>🶨</td>
<td>🶨</td>
<td>🶨</td>
</tr>
<tr>
<td><strong>10</strong> Backtrack/ B&amp;B</td>
<td>🶨</td>
<td>🶨</td>
<td>🶨</td>
<td>🶨</td>
<td>🶨</td>
<td>🶨</td>
<td>🶨</td>
<td>🶨</td>
<td>🶨</td>
<td>🶨</td>
<td>🶨</td>
<td>🶨</td>
</tr>
<tr>
<td><strong>11</strong> Graphical Models</td>
<td>🶨</td>
<td>🶨</td>
<td>🶨</td>
<td>🶨</td>
<td>🶨</td>
<td>🶨</td>
<td>🶨</td>
<td>🶨</td>
<td>🶨</td>
<td>🶨</td>
<td>🶨</td>
<td>🶨</td>
</tr>
<tr>
<td><strong>12</strong> Unstructured Grid</td>
<td>🶨</td>
<td>🶨</td>
<td>🶨</td>
<td>🶨</td>
<td>🶨</td>
<td>🶨</td>
<td>🶨</td>
<td>🶨</td>
<td>🶨</td>
<td>🶨</td>
<td>🶨</td>
<td>🶨</td>
</tr>
</tbody>
</table>
Past algorithms: FLOPs expense, Moves cheap

From architects, numerical analysts interacting, learn that now Moves expensive, FLOPs cheap

New theoretical lower bound of moves to FLOPs

Success of theory and practice: real code now achieves lower bound of moves to great results

Even Dense Matrix: 8.8X speedup over Intel MKL Quad 4-Core Nehalem for QR Decomposition.

Widely applicable: any arbitrary nested loops...

Recent result: 1.33x peak on Cray XT4 using C-A Strassen matrix multiply
The QR decomposition of tall-skinny matrices is a key computation in many applications:
- Linear least squares
- K-step Krylov methods
- Stationary video background subtraction

Communication-avoiding QR is a recent algorithm proven to be “communication-optimal”:
- Turns tall-skinny QR into compute-bound problem

CAQR performs up to 13x better for tall-skinny matrices than existing GPU libraries.

Outperforms GPU linear algebra library (CULA) for matrices up to ~2000 columns wide.
- New algorithm for Breadth-First Search
- Highest single-node performance in November 2011, Graph500, using Intel Xeon E7-8870 (Mirasol)
  - #15: BlueGene 2048 cores 6.93 GTEPS
  - #16: Jaguar 1024 cores 6.26 GTEPS
  - #17: Mirasol 40 cores 5.12 GTEPS
  - #18: Blacklight 512 cores 4.45 GTEPS
  - #19: Todi 176 TESLA GPUs 3.05 GTEPS
  - #20: Convey 4 FPGAs 1.76 GTEPS
“Our” Pattern Language (OPL-2010)  
(Kurt Keutzer, Tim Mattson)
Only a few types of hardware platform

- Multicore
- GPU
- “Cloud”
High-level pattern constrains space of reasonable low-level mappings

**Figure 1**: overall structure of OPL showing the five layer model. Implementation strategy patterns are divided into 2 sets; one describing a program's structure and the other data structures. The concurrent execution patterns are broken down into a set of patterns that “advance a program counter” and a set that coordinates the execution of parallel threads.
Specializers: Pattern-specific and platform-specific compilers

aka. “Stovepipes”

Allow maximum efficiency and expressibility in specializers by avoiding mandatory intermediary layers
Problem: generating optimized code is like searching for needle in haystack; use computers rather than humans

Auto-tuners approach: program generates optimized code and data structures for a “motif” (~kernel) mapped to some instance of a family of architectures (e.g., x86 multicore)

Use empirical measurement to select best performing.

ParLab autotuners for stencils (e.g., images), sparse matrices, particle/mesh, collectives (e.g., “reduce”), …
SEJITS: “Selective, Embedded, Just-In Time Specialization” (Fox)

- SEJITS bridges productivity and efficiency layers through specializers embedded in modern high-level productivity language (Python, Ruby, …)
  - Embedded “specializers” use language facilities to map high-level pattern to efficient low-level code (at run time, install time, or development time)
  - Specializers can incorporate/package autotuners

Two ParLab SEJITS projects:

- **Copperhead**: Data-parallel subset of Python, development continuing at NVIDIA

- **Asp**: “Asp is SEJITS in Python” general specializer framework
  - Provide functionality common across different specializers
SEJITS Overview

Selective

Embedded

Specialization

Specializer

SEJITS

PLL Interp

OS/HW

.f()

.h()

.py

.JIT

.C

.cc/ld

.$

.so
Asp: Who Does What?

App author (PLL)
Application
Kernel

Specializer author (ELL)
Specializer
Domain-Specific Transforms
Target AST

Asp team
Asp core
Python AST
Utilities
Asp Module

3rd party libraries
Compiled libraries

SEJITS version of meeting diarizer, 250x realtime from 50 lines of Python
Composition

- All applications built as a hierarchy of modules, not just one kernel

![Diagram showing an application composed of three modules: Module 1, Module 2, and Module 3.]

Structural patterns describe the common forms of composing sub-computations:
- E.g., task graph, pipelines, agent/repository
Efficient Parallel Composition of Libraries is Hard

Gaming App Example

Libraries compete unproductively for resources!
"Harts": Hardware Threads
A Better Resource Abstraction

- *Merged* resource and computation abstraction.

- More accurate resource abstraction.
- Let apps provide own computation abstractions.
Lithe is an ABI to allow application components to co-operatively share hardware threads.

Each component is free to map computational to hardware threads in any way they see fit

- No mandatory thread or task abstractions

Components request but cannot demand harts, and must yield harts when blocked or finished with task
**Tessellation OS: Space-Time Partitioning + 2-Level Scheduling (Kubiatowicz)**

**1st level:** OS determines coarse-grain allocation of resources to jobs over space and time

**2nd level:** Application schedules component tasks onto available “harts” (hardware thread contexts) using Lithe

---

**Diagram Details:**
- **Address Space A** and **Address Space B**
- **Tessellation Kernel (Partition Support)**
- **CPU L1**
- **L1 Interconnect**
- **L2 Bank**
- **DRAM & I/O Interconnect**
- **DRAM**

---

**Image Description:**
- A 3D diagram illustrating space and time partitioning with tessellation.
Resource Management using Convex Optimization (Sarah Bird, Burton Smith)

- Each process receives a **vector of basic resources** dedicated to it
  - e.g., fractions of cores, cache slices, memory pages, bandwidth
- Allocate minimum for QoS requirements
- Allocate remaining to meet some system-level objective
  - e.g., best performance, lowest energy, best user experience

---

**Continuously Minimize**

(subject to restrictions on the total amount of resources)

**Penalty Function**
- Reflects the app’s importance

**Resource Utility Function**
- Performance as function of resources

\[
L_a = RU_a(r(0,a), r(1,a), \ldots, r(n-1,a))
\]

\[
L_b = RU_b(r(0,b), r(1,b), \ldots, r(n-1,b))
\]

- Allocate minimum for QoS requirements
- Allocate remaining to meet some system-level objective
  - e.g., best performance, lowest energy, best user experience

**Convex Surface**

**Performance Metric** \( (L) \), e.g., latency
Par Lab Stack Summary

- Organize software around parallel patterns
  - Maximize reuse since patterns common across domains
- Each pattern implemented with efficient algorithms packaged as SEJITS specializers using autotuners
- Programmer composes functionality at high-level using productivity language
- System composes resource usage at low-level using 2-level scheduling
  - Tessellation OS at coarse-grain
  - Lithe user-level scheduler ABI at fine-grain
Par Lab Stack Overview

Application 1

Module 1
Specializer
Efficiency Level Code
Module 1 Scheduler

Module 2
TBB Code

Module 3
Wrapper
Legacy OpenMP

Module 3 Scheduler
OpenMP Scheduler

Lithe User-Level Scheduling ABI

Application 2

Tessellation OS

Hardware Resources (Cores, Cache/Local Store, Bandwidth)
Supporting QoS inside Apps

Tessellation OS

Hardware Resources (Cores, Cache/Local Store, Bandwidth)

Efficiency Level Code

Module 1
Scheduler

Lithe

TBB Code

Module 1
Scheduler

TBB Scheduler

Real-Time Cell

Efficiency Level Code

Module 2

Best-Effort Cell

TBB Code

Module 2
Scheduler

Real-Time Scheduler

Module 3
RAMP Gold

- Rapid accurate simulation of manycore architectural ideas using FPGAs
- Initial version models 64 cores of SPARC v8 with shared memory system on $750 board
- Hardware FPU, MMU, boots our OS and Par Lab stack!

<table>
<thead>
<tr>
<th></th>
<th>Cost</th>
<th>Performance (MIPS)</th>
<th>Time per 64 core simulation</th>
</tr>
</thead>
<tbody>
<tr>
<td>Software Simulator</td>
<td>$2,000</td>
<td>0.1 - 1</td>
<td>250 hours</td>
</tr>
<tr>
<td>RAMP Gold</td>
<td>$2,000 + $750</td>
<td>50 - 100</td>
<td>1 hour</td>
</tr>
</tbody>
</table>
What Next?
Why move to parallelism?

Moore’s Law continues (transistors/die, $/transistor)

But end of Dennard scaling reduces expected energy/performance gains from process technology
  - Cannot scale Vt down due to leakage/variability, Vdd stays high.

Energy/Operation is key metric for all systems
  - Limits achievable performance under power constraint
  - Limits battery life in mobile devices
  - Sets operating cost for data centers
How Parallelism Helps Energy/Op

Voltage-Frequency Scaling:
- Replace one high-voltage/high-frequency core with multiple lower-voltage/lower-frequency cores.

Simpler Cores:
- Replace one complex core with multiple simpler cores.

Can do both at once.
Future advances need more than just parallelism

- Voltage-Frequency scaling of limited benefit in future technologies
  - Not much difference between Vdd and Vt
- Move to simpler general-purpose cores is a one-time gain
  - In smartphones, cores were already relatively simple
- More transistors per die than we can power at the same time (utilization wall)

- What next?
Heterogeneity?

- Much agreement that heterogeneity comes next
- But many different views on what heterogeneity means
Heterogeneous Research Today

- Large design space
- Lots of earlier work
  - many failures (e.g., NeXT DSP, IBM Cell, reconfigurable computing)
  - few successes (e.g., GP-GPU)
- Used in niche applications now, but looks inevitable for widespread hardware adoption
- How can software keep up?
- Much confusion in industry

Sound familiar? (Parallel computing 10 years ago)
Specialization >> Heterogeneity

- Do not need heterogeneity to benefit from specialization
- Heterogeneity is one way to deliver specialization
- Alternative approaches:
  - Homogeneous cores with wide variety of coprocessors/extended instruction sets
  - Homogeneous reconfigurable cores
- Can use all of the above in one system

Research question: When does core heterogeneity make sense versus richer homogeneous cores?
Types of Specialization

Less specialized

- Same core design, different VF operating points
- Same core design, runtime configurable components
- Same ISA, different μarchitectures
- Variants of same ISA (subsets, different extensions)
- Completely different ISAs
- Programmable logic (no ISA)
- Fixed-function accelerators (no programming)

More specialized

NVIDIA Tegra2
Speed/Energy Trade-offs on Image Contour Detection

- **Time to Perform Computation (Normalized to GTX 480):**
  - **Snap dragon:** 247x
  - **ASIC:** 1.0x
  - **Fermi GPU:** 1.0x
  - **Core 2 Duo:** 30x
  - **Nehalem i7:** 5.3x

- **Energy to Perform Computation in Joules:**
  - **68J**
  - **100J**
  - **222J**
  - **717J**
  - **944J**

Range of:
- 247x in speed
- 14x in energy
Speed/Energy Trade-offs on Speech Recognition

- Range of
  - 42X in speed
  - 15X in energy
Specialized Memory and Interconnect
(at least as important as cores)

- Hierarchical main memory: DRAM (graphics, commodity) + Flash/MRAM
- Coherence protocols
- Software-managed memory
- Synchronization primitives
- On-the-fly compression/decompression
There’s a HUGE design space

- How are we to make sense of this space?

*Extend patterns work into hardware*
Berkeley Hardware Pattern Language (BHPL)

Applications (including OPL patterns)

BHPL

App-to-UTL Mappings Layer

Problem: Application Computation
Solution: UTL Machine

FFT to SIMD array

UTL-to-UTL Transformation Layer

Problem: UTL violates constraint (too big, too slow)
Solution: Transformed UTL

Time-Multiplexing
Unrolling

UTL-to-RTL Transformation Layer

Problem: UTL design
Solution: RTL behavior

Microcoded Engine
In-Order Pipeline Engine

RTL-to-Technology

Problem: RTL behavior
Solution: Structural design

Interleaved Memory
FIFO
CAM
Chisel (Constructing Hardware in a Scala Embedded Language) under active development

- Generate C simulator + FPGA emulation + ASIC synthesis from one RTL description
- Supports higher-level libraries, e.g., BlueChisel – embedded atomic actions language used in cache-coherence protocol work.

Chisel is a language designed to support development of hardware specializers
Chisel Design Flow

Chisel Design Description

Chisel Compiler

C++ code

FPGA Verilog

ASIC Verilog

C++ Compiler

FPGA Tools

FPGA Emulation

ASIC Tools

GDS Layout
“Next” Project

- Catalog common hardware patterns, both computational and structural
- Develop communication-avoiding hardware algorithms for important patterns
- Develop hardware+software specializers to support massive design-space exploration
- Extend “stovepipes” into specialized hardware
Par Lab Funding

- Research supported by Microsoft (Award #024263) and Intel (Award #024894) funding and by matching funding by U.C. Discovery (Award #DIG07-10227).
- Additional support comes from Par Lab affiliates National Instruments, NEC, Nokia, NVIDIA, Samsung, and Oracle/Sun.
Questions?