Architecting Parallel Software
Keutzer and Mattson

The Challenge of Parallelism
Programming parallel processors is one of the challenges of our era

- NVIDIA Tegra 2 system on a chip (SoC)
  - Dual-core ARM Cortex A9
  - Integrated GPU, Lots of DSP
  - 1 GHz
  - 2 double-precision GFLOPs peak (CPUs only)

- NVIDIA Fermi
  - 16 cores, 48-way multithreaded
  - 4-wide Superscalar, dual-issue, 3
  - 2-wide SIMD (half-pumped)
  - 2 MB (16 x 128 KB) Registers, 1
  - MB (16 x 64 KB) L1 cache, 2 MB L2 Cache

- Tilera Tile64
  - 64 processors
  - Each tile has L1, L2, can run OS
  - 443 billion operations/sec
  - 50-833 MHz
  - 50 Gbytes/sec memory bandwidth

Outline
- What doesn’t work
- Pieces of the problem … and solution
- General approach to architecting parallel sw
- Detail on Structural Patterns
- Detail on Computational Patterns
- High-level examples of architecting applications

Assumption #1:
How not to develop parallel code

- Initial Code
- Profiler
- Performance profile
- Re-code with more threads
- Not fast enough
- Fast enough
- Ship it
- Lots of failures
- N PE’s slower than 1
Architecting Parallel Software
Keutzer and Mattson

---

**Steiner Tree Construction Time By Routing Each Net in Parallel**

<table>
<thead>
<tr>
<th>Benchmark</th>
<th>Serial</th>
<th>2 Threads</th>
<th>3 Threads</th>
<th>4 Threads</th>
<th>5 Threads</th>
<th>6 Threads</th>
</tr>
</thead>
<tbody>
<tr>
<td>adaptec1</td>
<td>1.68</td>
<td>1.68</td>
<td>1.70</td>
<td>1.69</td>
<td>1.69</td>
<td>1.69</td>
</tr>
<tr>
<td>newblue1</td>
<td>1.80</td>
<td>1.80</td>
<td>1.81</td>
<td>1.81</td>
<td>1.81</td>
<td>1.82</td>
</tr>
<tr>
<td>newblue2</td>
<td>2.60</td>
<td>2.60</td>
<td>2.62</td>
<td>2.62</td>
<td>2.62</td>
<td>2.61</td>
</tr>
<tr>
<td>adaptec2</td>
<td>1.87</td>
<td>1.86</td>
<td>1.87</td>
<td>1.88</td>
<td>1.88</td>
<td>1.88</td>
</tr>
<tr>
<td>adaptec3</td>
<td>3.32</td>
<td>3.32</td>
<td>3.34</td>
<td>3.34</td>
<td>3.34</td>
<td>3.34</td>
</tr>
<tr>
<td>adaptec4</td>
<td>3.20</td>
<td>3.20</td>
<td>3.21</td>
<td>3.21</td>
<td>3.21</td>
<td>3.21</td>
</tr>
<tr>
<td>adaptec5</td>
<td>4.91</td>
<td>4.90</td>
<td>4.92</td>
<td>4.92</td>
<td>4.92</td>
<td>4.92</td>
</tr>
<tr>
<td>newblue3</td>
<td>2.54</td>
<td>2.55</td>
<td>2.55</td>
<td>2.55</td>
<td>2.55</td>
<td>2.55</td>
</tr>
<tr>
<td><strong>average</strong></td>
<td><strong>1.00</strong></td>
<td><strong>1.0011</strong></td>
<td><strong>1.0044</strong></td>
<td><strong>1.0049</strong></td>
<td><strong>1.0046</strong></td>
<td><strong>1.0046</strong></td>
</tr>
</tbody>
</table>

---

**So What’s the Alternative?**

---

**Outline**
- What doesn’t work
- Pieces of the problem … and solution
- General approach to architecting parallel sw
- Detail on Structural Patterns
- Detail on Computational Patterns
- High-level examples of architecting applications

---

**Hint: What is this person thinking of?**

Edward Lee, “The Problem with Threads”

Threads, locks, semaphores, data races
Principles of SW Design

After 15 years in industry, at one time overseeing the technology of 25 software products, my best principle to facilitate good software design is modularity:

- Modularity helps:
  - Architect: Makes overall design sound and comprehensible
  - Project manager:
    - As a manager I am able to comfortably assign different modules to different developers
    - I am also able to use module definitions to track development
    - Build a PERT chart for development progress
    - Build a "control panel" for current software quality
  - Module implementors: As a module implementor I am able to focus on the implementation, optimization, and verification of my module with a minimum of concern about the rest of the design
  - Modularity helps to identify key computations

What’s life like without modularity?

- Spaghetti code
- Wars over the interpretation of the specification
- Waiting on other coders
- Wondering why you didn’t touch anything and now your code broke
- Hard to verify your code in isolation, and therefore hard to optimize
- Hard to parallelize without identifying key computations

Modularity will help us obviate all these

- Parnas, “On the criteria to be used on composing systems into modules,” CACM, December 1972.

Big Step:
Architectural Styles (Garland and Shaw, 1996)

- Pipe and filter
- Object oriented
- Event based
- Layered
- Agent and repository
- Process control

Object-Oriented Programming is Not Enough

Focused on:
- Program modularity
- Data locality
- Architectural styles
- Design patterns

Neglected:
- Application concurrency
- Computational details
- Parallel implementations
What’s missing?: Is an executing software program more like?

a) A building  b) A factory

We need to consider the machinery – but what is the machinery?

Computations are the Machinery

HPC knows a lot about computations, application concurrency, efficient programming, and parallel implementation

\[ 2_t = \sum_i b_i + f_i \]
\[ t = Wt \]
\[ z = f_0 + f_1 + f_2 \]

minimize \( ||Wz|| \)

subject to \( Fz = \bar{p} \)
\( ||Gz - \bar{z}|| < \epsilon \)

High-end simulation in the physical sciences consists of seven algorithms:

- Structured Grids (including locally structured grids, e.g. AMR)
- Unstructured Grids
- Fast Fourier Transform
- Dense Linear Algebra
- Sparse Linear Algebra
- Particles
- Monte Carlo

Well-defined targets from algorithmic and software standpoint.

Remainder of this talk will consider one of them (structured grids) in detail.
Par Lab’s contribution: from 7 to 13 families of computations

Understanding computations is not enough either

Unfortunately ... HPC approach to software architecture architecture
Technically this is known as a monolithic architecture

Outline
- What doesn’t work
- Pieces of the problem ... and solution
- General approach to architecting parallel sw
- Detail on Structural Patterns
- Detail on Computational Patterns
- High-level examples of architecting applications
Alexander's Pattern Language

Christopher Alexander's approach to (civil) architecture:

- Each pattern describes a problem which occurs over and over again in our environment, and then describes the core of the solution to that problem, in such a way that you can use this solution a million times over, without ever doing it the same way twice. - Page x, *A Pattern Language*, Christopher Alexander

Alexander's 253 (civil) architectural patterns range from the creation of cities (2. distribution of towns) to particular building problems (232. roof cap)

A pattern language is an organized way of tackling an architectural problem using patterns

Main limitation:

- It's about civil not software architecture!!!

Uses of Patterns

Patterns give names and definitions to key elements of design

This enables us to better:

- Teach design – a palette of defined design principals
  - Gives ideas to new programmers – approaches you may not have considered
  - Gives a set of finiteness to experienced programmers – if you’ve considered all the patterns then you can rest assured you’ve considered the key approaches
- Guide design – articulate design decisions succinctly
- Communicate design – improve documentation, facilitate maintenance of software

Uses of Patterns

Patterns capture and preserve bodies of knowledge about key design decisions

- Useful implementation techniques
- Likely challenges/bottlenecks that will come with the use of this pattern (e.g. repository bottleneck in agent and repository)

Architecting Parallel Software with Patterns

Identify the Software Structure

- Pipe-and-Filter
- Agent-and-Repository
- Event-based
- Process Control
- Layered Systems
- Model-view controller
- Iterator
- MapReduce
- Arbitrary Task Graphs
- Puppeteer

Identify the Key Computations

- Graph Algorithms
- Dynamic programming
- Dense/Sparse Linear Algebra
- (Un)Structured Grids
- Graphical Models
- Finite State Machines
- Backtrack Branch-and-Bound
- N-Body Methods
- Circuits
- Spectral Methods
Decompose Tasks
• Group tasks
• Order Tasks

Decompose Data
• Identify data sharing
• Identify data access

Identify the Software Structure

Architecting Parallel Software

Identify the SW Structure

Structural Patterns
• Pipe-and-Filter
• Agent-and-Repository
• Event-based
• Process Control
• Layered Systems
• Model-view controller
• Iterator
• MapReduce
• Arbitrary Task Graphs
• Puppeteer

These define the structure of our software but they do not describe what is computed.

Analogy: Layout of Factory Plant

Identify key computations....

Computational patterns describe the key computations but not how they are implemented.
Analogy: Machinery of the Factory

Analogy: Architected Factory

Raises appropriate issues like scheduling, latency, throughput, workflow, resource management, capacity etc.

Architecting Parallel Software

Structural Patterns

- Pipe-and-Filter
- Agent-and-Repository
- Event-based
- Layered Systems
- Model-view-controller
- Arbitrary Task Graphs
- Puppeteer
- Iterator/ BSP
- MapReduce

Computational Patterns

- Graph- Algorithms
- Dynamic- Programming
- Dense- Linear- Algebra
- Sparse- Linear- Algebra
- Unstructured- Grids
- Structured- Grids
- Graphical- Models
- Finite- State- Machines
- Backtrack- Branch-and- Bound
- N- Body- Methods
- Circuits
- Spectral- Methods
- Monte- Carlo

Remember this Poor Guy ...

Edward Lee,
“The Problem with Threads”

Re-code with more threads

Threads, locks, semaphores, data races
What’s this person thinking of …?

Need to integrate the insights into computation provided by HPC with the insights into program structure provided by software architectural styles.

Outline

- What doesn’t work
- Pieces of the problem … and solution
- General approach to architecting parallel sw
- Detail on Structural Patterns
- Detail on Computational Patterns
- High-level examples of architecting applications

Inventory of Structural Patterns

1. pipe and filter
2. iterator
3. MapReduce
4. blackboard/agent and repository
5. process control
6. Model View Controller
7. layered
8. event-based coordination
9. puppeteer
10. static task graph

Elements of a structural pattern

- Components are where the computation happens
- A configuration is a graph of components (vertices) and connectors (edges)
- A structural patterns may be described as a family of graphs.

Connectors are where the communication happens
Pattern 1: Pipe and Filter

- Filters embody computation
- Only see inputs and produce outputs

Filter 1
Filter 2
Filter 3
Filter 4
Filter 5
Filter 6
Filter 7

Pipes embody communication
May have feedback

Examples?

Examples of pipe and filter

- Almost every large software program has a pipe and filter structure at the highest level

Examples?

Pattern 2: Iterator Pattern

- Variety of functions performed asynchronously
- Synchronize results of iteration
- Exit condition met?

Initialization condition

Examples?

Example of Iterator Pattern: Training a Classifier: SVM Training

Update surface
Identify Outlier

All points within acceptable error?

Examples?
Pattern 3: MapReduce

To us, it means

- A map stage, where data is mapped onto independent computations
- A reduce stage, where the results of the map stage are summarized (i.e. reduced)

Examples?

Examples of Map Reduce

- General structure:
  - Map a computation across distributed data sets
  - Reduce the results to find the best/(worst), maxima/(minima)

- Support-vector machines (ML)
  - Map to evaluate distance from the frontier
  - Reduce to find the greatest outlier from the frontier

- Speech recognition
  - Map HMM computation to evaluate word match
  - Reduce to find the most-likely word sequences

Pattern 4: Agent and Repository

Agent and repository: Blackboard structural pattern

Agents cooperate on a shared medium to produce a result

Key elements:

- **Blackboard**: repository of the resulting creation that is shared by all agents (circuit database)
- **Agents**: intelligent agents that will act on blackboard (optimizations)
- **Manager**: orchestrates agents access to the blackboard and creation of the aggregate results (scheduler)

Example: Compiler Optimization

Optimization of a software program

- Intermediate representation of program is stored in the repository
- Individual agents have heuristics to optimize the program
- Manager orchestrates the access of the optimization agents to the program in the repository
- Resulting program is left in the repository
Example: Logic Optimization

- Optimization of integrated circuits
- Integrated circuit is stored in the repository
- Individual agents have heuristics to optimize the circuitry of an integrated circuit
- Manager orchestrates the access of the optimization agents to the circuit repository
- Resulting optimized circuit is left in the repository

Pattern 5: Process Control

- Process control:
  - Process: underlying phenomena to be controlled/computed
  - Actuator: task(s) affecting the process
  - Sensor: task(s) which analyze the state of the process
  - Controller: task which determines what actuators should be effected

Examples?

Pattern 9: Puppeteer

- Need an efficient way to manage and control the interaction of multiple simulators/computational agents
- Puppeteer Pattern – guides the interaction between the tasks/puppets to guarantee correctness of the overall task
- Puppeteer: 1) schedules puppets 2) manages exchange of data between puppets
- Difference with agent and repository?
  - No central repository
  - Data transfer between tasks/puppets

Examples?
Video Game

Framework
- Change Control Manager
- Interfaces
- Input
- Physics
- Graphics
- AI

Model of circulation
- Modeling of blood moving in blood vessels
- The computation is structured as a controlled interaction between solid (blood vessel) and fluid (blood) simulation codes
- The two simulations use different data structures and the number of iterations for each simulation code varies
- Need an efficient way to manage and control the interaction of the two codes

Pattern 10: Static Task Graph
Tasks receive inputs and produce outputs
All data sharing is through explicit messaging (arrow “→” means message passing communication)
Task configuration is statically defined and may not be changed at runtime

Example: one game architecture
There exist fixed dependencies between subsystems
Can be modeled as an arbitrary task graph
Example: Moving the zombie
- Keyboard → AI → Physics → Graphics
Outline

- What doesn’t work
- Pieces of the problem … and solution
- General approach to architecting parallel sw
- Detail on Structural Patterns
- Detail on Computational Patterns
- High-level examples of architecting applications

You explore these every class

- Structured Grid
- Dense Matrix
- Sparse Matrix
- Spectral (FFT)
- Monte Carlo
- N-Body
- Graph Algorithms
- Graphical Models
- Backtrack / B&B
- Finite State Mach.
- Dynamic Prog.
- Unstructured Grid
- Structured Grid
- Dense Matrix
- Sparse Matrix
- Spectral (FFT)
- Monte Carlo
- N-Body

Automatic Speech Recognition
Inference engine based system
- Used in Sphinx (CMU, USA), HTK (Cambridge, UK), and Julius (CSRC, Japan) [10,15,9]
- Modular and flexible setup
  - Shown to be effective for Arabic, English, Japanese, and Mandarin
Iterative Refinement Structural Pattern

- One iteration per time step
- Identify the set of probable states in the network given acoustic signal given current active state set
- Prune unlikely states
- Repeat

Inference Engine in LVCSR

- Three steps of inference
  0. Gather operands from irregular data structure to runtime buffer
  1. Perform observation probability computation
  2. Perform graph traversal computation

Parallelism in the inference engine:

0. Gather operand
1. $P(x_t|x_s)$
2. $m[t][s_t]$

Each Filter is a Map Reduce

0. Gather operands

1. $m[t][s_t] = \max_{s_{t-1}} m[t-1][s_{t-1}] \cdot P(s_t|x_{t-1}) \cdot P(x_t|s_t)$

- Gather and coalesce each of the above operands for every $s_t$
- Facilitates opportunity for SIMD

Each Filter is Map Reduce

1. observation probability computation

$$m[t][s_t] = \max_{s_{t-1}} m[t-1][s_{t-1}] \cdot P(s_t|x_{t-1}) \cdot P(x_t|s_t)$$

- Gaussian Mixture Model Probability
- Probability that given this feature-frame (e.g. 10ms) we are in this state/phone

max
Observation probability are computed from Gaussian Mixture Models:
- Each Gaussian probability in each mixture is independent.
- Probability for one phone state is the sum of all Gaussians times the mixture probability for that state.

\[ p(x_t | \mu, \Sigma) = \sum_{i} n_i g(x_t | \mu_i, \Sigma_i) \]

Each Filter is Map Reduce:
- Map: probability computation across distributed data sets – perform multiplication as below.
- Reduce: the results to find the maximally likely states.

\[ m[t][s] = \max_{s_{t-1}} m[t-1][s_{t-1}] \cdot P(s_{t-1} | s_t) \cdot P(s_t | s) \]

All together: Inference Engine in LVCSR:
- Put all together the inference engine is dynamic programming.
HMM computed with Dynamic Programming

Outline
- What doesn’t work
- Pieces of the problem ... and solution
- General approach to architecting parallel sw
- Detail on Structural Patterns
- Detail on Computational Patterns
- High-level examples of architecting applications
- Summary

Recap: Architecting Parallel Software
1. Start with a compelling, performance sensitive application.
2. Define the overall structure
3. Define computations inside structural elements
4. Compose Structural and computational patterns to yield software architecture

This Approach Works

<table>
<thead>
<tr>
<th>Application</th>
<th>Our Speedup</th>
</tr>
</thead>
<tbody>
<tr>
<td>MRI Reconstruction</td>
<td>100x</td>
</tr>
<tr>
<td>SVM Training</td>
<td>20x</td>
</tr>
<tr>
<td>SVM Deployment</td>
<td>100x</td>
</tr>
<tr>
<td>Value-at-risk</td>
<td>60x</td>
</tr>
<tr>
<td>Option Pricing</td>
<td>25x</td>
</tr>
<tr>
<td>Speech Recognition</td>
<td>11x</td>
</tr>
<tr>
<td>Video Event Detection</td>
<td>78x</td>
</tr>
<tr>
<td>Contour Detection</td>
<td>130x</td>
</tr>
<tr>
<td>Object Recognition</td>
<td>80x</td>
</tr>
<tr>
<td>Poselets</td>
<td>20x</td>
</tr>
<tr>
<td>Optical Flow</td>
<td>32x</td>
</tr>
<tr>
<td>Deformable Parts Models</td>
<td>45x</td>
</tr>
<tr>
<td>Image Convolution</td>
<td>8x</td>
</tr>
<tr>
<td>Deep Neural Network Training</td>
<td>47 - 140x</td>
</tr>
</tbody>
</table>

“Considerations When Evaluating Microprocessor Platforms” In Proceedings of the 3rd USENIX conference on Hot topics in parallelism (HotPar’11), USENIX Association, Berkeley, CA, USA.
Computational Patterns

Make me Feel Smart

For many years computation has been like a big ball of yarn
Computational patterns help us to unravel it into 13 strands
Alan Kay “Perspective is worth 100 IQ points.”
Computational patterns give us perspective on computation

Summary

- The key to productive and efficient parallel programming is creating a good software architecture – a hierarchical composition of:
  - Structural patterns: enforce modularity and expose invariants
    - The showed you six – four more will be all you ever need
  - Computational patterns: identify key computations to be parallelized
  - Orchestratio of computational and structural patterns creates architectures which greatly facilitates the development of parallel programs:

Short Course Hosted at Intel Software University Website:
- Engineering Parallel Software with Patterns
  - http://university.intel.com/
Semester Long Course Taught at Berkeley and Hosted at XSEDE
- https://cvw.cac.cornell.edu/eps/default
Patterns: https://patterns.eecs.berkeley.edu/
More examples