Architecting Parallel Software with Patterns

Kurt Keutzer, EECS, Berkeley

with thanks to Tim Mattson, Intel and the PALLAS team:
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 single-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, 0.75 MB L2 Cache

Tilera Tile64
- 64 processors
- Each tile has L1, L2, can run OS
- 443 billion operations/sec.
- 500-833 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

Lots of failures

N PE’s slower than 1

Ship it
Hint: What is this person thinking of?

Edward Lee, “The Problem with Threads”

Re-code with more threads

Threads, locks, semaphores, data races
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
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

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

\[ x_c \leftarrow \sum_j g_{cj} \cdot x_j \]

\[ x \leftarrow W S \times \{W^* x\} \]

\[ x \leftarrow F(P^T y + P_c^T P_c F^* x) \]

minimize \( ||Wx||_1 \)

s.t. \( F_\Omega x = y \),
\[ ||Gx - x||_2 < \varepsilon \]

\[ \frac{\partial V}{\partial t} + \frac{1}{2} \sigma^2 S^2 \frac{\partial^2 V}{\partial S^2} + rS \frac{\partial V}{\partial S} - rV = 0. \]

\[ J(w) = \int_\Omega \psi_1(|I(x + w) - I(x)|^2)dx + \]
\[ \gamma \int_\Omega \psi_2(|\nabla I(x + w) - \nabla I(x)|^2)dx + \]
\[ \alpha \int_\Omega \psi_3(|\nabla u|^2 + |\nabla v|^2)dx \]

13
Defining Software Requirements for Scientific Computing

Phillip Colella
Applied Numerical Algorithms Group
Lawrence Berkeley National Laboratory
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

<table>
<thead>
<tr>
<th>Dwarves</th>
<th>Apps</th>
<th>Embed</th>
<th>SPEC</th>
<th>DB</th>
<th>Games</th>
<th>ML</th>
<th>HPC</th>
<th>CAD</th>
<th>Health</th>
<th>Image</th>
<th>Speech</th>
<th>Music</th>
<th>Browser</th>
</tr>
</thead>
<tbody>
<tr>
<td>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>
<td></td>
</tr>
<tr>
<td>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>
<td></td>
</tr>
<tr>
<td>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>
<td></td>
</tr>
<tr>
<td>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>
<td></td>
</tr>
<tr>
<td>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>
<td></td>
</tr>
<tr>
<td>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>
<td></td>
</tr>
<tr>
<td>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>
<td></td>
</tr>
<tr>
<td>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>
<td></td>
</tr>
<tr>
<td>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>
<td></td>
</tr>
<tr>
<td>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>
<td></td>
</tr>
<tr>
<td>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>
<td></td>
</tr>
<tr>
<td>Monte Carlo</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>
<td></td>
</tr>
<tr>
<td>N-Body</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>
<td></td>
</tr>
</tbody>
</table>
Unfortunately ... HPC approach to software architecture

Technically this is known as a *monolithic* architecture
How can we integrate these insights?

• We wish to find an approach to building software that gives equal support for two key problems of software design – how to structure the software and how to efficiently implement the computations
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
Architecting Parallel Software

Decompose Tasks
- Group tasks
- Order Tasks

Decompose Data
- Identify data sharing
- Identify data access

Identify the Software Structure

Identify the Key Computations
Identify the SW Structure

Structural Patterns

• Pipe-and-Filter
• Agent-and-Repository
• Event-based coordination
• Iterator
• MapReduce
• Process Control
• Layered Systems

These define the structure of our software but they do not describe what is computed.
Analogy: Layout of Factory Plant
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
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

**Software architecture**

**computational patterns**
- Linear Algebra
- Spectral
- Stencil
- Graph algorithms
- FSMS

**structural patterns**
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
- Connectors are where the communication happens

- A configuration is a graph of components (vertices) and connectors (edges)
- A structural patterns may be described as a family of graphs.
Pattern 1: Pipe and Filter

- Filters embody computation
- Only see inputs and produce outputs

- 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.
Pattern 2: Iterator Pattern

- Variety of functions performed asynchronously
- Synchronize results of iteration
- Exit condition met?
  - Yes
  - No
- Iterate
- Initialization condition

Examples?
Example of Iterator Pattern: Training a Classifier: SVM Training

- Update surface
- Identify Outlier
- All points within acceptable error?
- Yes
- No

Iterator Structural Pattern
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
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

*Source: Adapted from Shaw & Garlan 1996, p27-31.*

Examples?
Examples of Process Control

- **Return air**
- **Furnace**
- **Hot air**
- **Temperature sensor**
- **Thermostat**
- **Temperature-setting control**
- **Gas-valve control**

**User timing constraints**

**Process control structural pattern**

**Timing constraints**

**Controller**

**Circuit**

**Launching transformations**

**Speed? Power?**
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
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
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
  - I showed you six – four more will be all you ever need
- Computational patterns: identify key computations to be parallelized
- Orchestration of computational and structural patterns creates architectures which greatly facilitates the development of parallel programs:

Patterns: https://patterns.eecs.berkeley.edu/
More examples
CBIR Application Framework

Feature Extraction

Image histograms are common to many feature extraction procedures, and are an important feature in their own right

- Agent and Repository: Each agent computes a local transform of the image, plus a local histogram. Results are combined in the repository, which contains the global histogram

- The data dependent access patterns found when constructing histograms make them a natural fit for the agent and repository pattern
Train Classifier:
SVM Training

Train Classifier

Update Optimality Conditions

Select Working Set, Solve QP

MapReduce

Gap not small enough?

Iterator

Iterate

New Images

Choose Examples

Feature Extraction

Learn Classifier

Results

User Feedback

Exercise Classifier
Exercise Classifier: SVM

Classification

1. Compute dot products
2. Compute Kernel values, sum & scale
3. SV
4. Test Data
5. Output

Dense Linear Algebra

MapReduce