Assumption #1: How not to develop parallel code

Initial Code -> Profiler -> Performance profile -> Ship it

Re-code with more threads

Not fast enough

Fast enough

Lots of failures

N PE’s slower than 1
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>3.33</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>average</td>
<td>1.00</td>
<td>1.0011</td>
<td>1.0044</td>
<td>1.0049</td>
<td>1.0046</td>
<td>1.0046</td>
</tr>
</tbody>
</table>

Assumption #2: Nor this

Initial Code

Super-compiler

Performance profile

Tune compiler

Not fast enough

Fast enough

30 years of HPC research don't offer much hope
Automatic parallelization?

- Aggressive techniques such as speculative multithreading help, but they are not enough.
- Ave SPECint speedup of 8% will climb to ave. of 15% once their system is fully enabled.
- There are no indications auto par. will radically improve any time soon.
- Hence, I do not believe Auto-par will solve our problems.

Results for a simulated dual core platform configured as a main core and a core for speculative execution.

Assumption #3: This won’t help either

- Code in new cool language
- Profiler -> Performance profile
- Re-code with cool language
- Not fast enough
- Fast enough
- Ship it

After 200 parallel languages where’s the light at the end of the tunnel?
Parallel Programming environments in the 90’s

<table>
<thead>
<tr>
<th>Abbreviation</th>
<th>Full Form</th>
</tr>
</thead>
<tbody>
<tr>
<td>ACE</td>
<td>Common Programming Environment for Distributed Systems</td>
</tr>
<tr>
<td>ACTs</td>
<td>Architecture for Computing Technology</td>
</tr>
<tr>
<td>ADE</td>
<td>Advanced Development Environment</td>
</tr>
<tr>
<td>ARPA</td>
<td>Advanced Research Projects Agency</td>
</tr>
<tr>
<td>ASPIRE</td>
<td>Advanced Scientific Programming Integrated Research Environment</td>
</tr>
<tr>
<td>ADEP</td>
<td>Advanced Development Environment for Parallel Systems</td>
</tr>
<tr>
<td>APE</td>
<td>Advanced Programming Environment</td>
</tr>
<tr>
<td>ARMS</td>
<td>Advanced Research in Multithreading Systems</td>
</tr>
<tr>
<td>ASK</td>
<td>Advanced Scientific Knowledge</td>
</tr>
<tr>
<td>ADEPT</td>
<td>Advanced Development Environment for Parallel Systems</td>
</tr>
<tr>
<td>ADEPT-II</td>
<td>Advanced Development Environment for Parallel Systems II</td>
</tr>
<tr>
<td>ADEL</td>
<td>Advanced Development Environment for Large-Scale Systems</td>
</tr>
<tr>
<td>ADELE</td>
<td>Advanced Development Environment for Large-Effort</td>
</tr>
<tr>
<td>ADEP-II</td>
<td>Advanced Development Environment for Parallel Systems II</td>
</tr>
<tr>
<td>ADEP-III</td>
<td>Advanced Development Environment for Parallel Systems III</td>
</tr>
<tr>
<td>ADEP-IV</td>
<td>Advanced Development Environment for Parallel Systems IV</td>
</tr>
<tr>
<td>ADEP-V</td>
<td>Advanced Development Environment for Parallel Systems V</td>
</tr>
<tr>
<td>ADEP-VI</td>
<td>Advanced Development Environment for Parallel Systems VI</td>
</tr>
<tr>
<td>ADEP-VII</td>
<td>Advanced Development Environment for Parallel Systems VII</td>
</tr>
<tr>
<td>ADEP-VIII</td>
<td>Advanced Development Environment for Parallel Systems VIII</td>
</tr>
<tr>
<td>ADEP-IX</td>
<td>Advanced Development Environment for Parallel Systems IX</td>
</tr>
<tr>
<td>ADEP-X</td>
<td>Advanced Development Environment for Parallel Systems X</td>
</tr>
</tbody>
</table>

So What’s the Alternative?
Principles of SW Design

- After 15 years in industry, at one time overseeing the technology of 25 software products, my two best principles to facilitate good software design are:
  - Use of modularity
  - Definition of invariants

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
- 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
- Identify invariants and key computations

Non-Principles of SW Design

- 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.
  - But modularity alone is not enough, because ...
Modularity is important .... But ...

Pop quiz: Is software more like?

a) A building       b) A factory

What computations we do is as important than how we do them ....

<table>
<thead>
<tr>
<th>Apps</th>
<th>Dwarves</th>
<th>Embedded</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>
Architecting Parallel Software

- We believe the solution to parallel programming is developing a good software architecture.
- A *software architecture* is a hierarchical composition of:
  - Computational patterns – the atoms, the machinery
  - Structural patterns – the molecular bonds, the layout
- This software architecture naturally gives:
  - Modularity supplied through structural patterns
    - Efficient management
    - Efficient implementation
    - Efficient verification
  - Computational efficiency supplied through computational patterns

Outline

- Architecting Parallel Software
- Structural Patterns
- Computational Patterns
- Examples
- Summary
Identify the SW Structure

Structural Patterns

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

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**

[Diagram showing various machinery and factory setup]
Architecting the Whole Application

- SW Architecture of Large-Vocabulary Continuous Speech Recognition
  - Analogous to the design of an entire manufacturing plant
  - Raises appropriate issues like scheduling, latency, throughput, workflow, resource management, capacity etc.

Outline

- Architecting Parallel Software
- Structural Patterns
  - Pipe and filter
  - Iterator
  - Map Reduce
- Computational Patterns
- Examples
- Summary
Elements of a structural pattern

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

Inventory of Structural Patterns

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

- We build arbitrarily complex software structures out of these nine patterns
Pattern 1: Pipe and Filter

- Filters embody computation
- Only see inputs and produce outputs
- No global or shared state

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

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

Example of Iterator Pattern: Training a Classifier: SVM Training

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

- Architecting Parallel Software
- Structural Patterns
- Computational Patterns
  - Linear Algebra
  - Spectral Methods
  - Dynamic programming
- Examples
- Summary

Inventory of Key Computations

We build arbitrarily complex computations out of these thirteen computational patterns.
CP1: Linear Algebra

- **Vector Space**: A set closed under + has identity and inverse elements, scalar multiplication
- **Linear Map**: Operator $T$ on vectors $u, v$, scalar $\alpha$ s.t. $T(u + v) = Tu + Tv$, and $T(\alpha v) = \alpha T(v)$
- **Matrix**: An $m \times n$ array of numbers representing a linear map from $\mathbb{R}^n$ to $\mathbb{R}^m$
- **Linear Equations**: $Ax = b$
- **Eigenvalues/vectors**: $Ax = \lambda x$

Basic Linear Algebra Subroutines (BLAS)

- Three "Levels", known as BLAS, characterized by intrinsic ratio of computation to memory movement

<table>
<thead>
<tr>
<th>Level</th>
<th>Example</th>
<th># mem refs</th>
<th># flops</th>
<th>q</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>xAXPY: $y = y + \alpha x$</td>
<td>$3n$</td>
<td>$2n^1$</td>
<td>$2/3$</td>
</tr>
<tr>
<td>2</td>
<td>xGEMV: $y = y + Ax$</td>
<td>$n^2$</td>
<td>$2n^2$</td>
<td>$2$</td>
</tr>
<tr>
<td>3</td>
<td>xGEMM: $C = C + AB$</td>
<td>$4n^2$</td>
<td>$2n^3$</td>
<td>$n/2$</td>
</tr>
</tbody>
</table>
"Spectral Methods" are a broad class of numerical algorithms for solving PDEs, but notions of Spectral Analysis (i.e. convenient changes of basis) are important in every application area.

In Magnetic Resonance Imaging (MRI), images are collected in "k-space" -- i.e. an MRI scan produces a Fourier Domain image.

Fourier and Wavelet representations are different Spectral analyses that expose different properties of images convenient for solving our problems.
Spectral Methods Pattern: Fast Transforms

- Spectral Methods rely on representations of data in "convenient" bases that produce working, computationally feasible algorithms.
- Changing a basis is, in general, an $O(N^2)$ matrix-vector multiplication. The matrices representing "convenient" bases factor into $O(N \log N)$ fast transforms!

$$f_b = \sum_{j=0}^{N-1} x_j \omega_j^b$$

$$F = \begin{bmatrix} 1 & 1 & 1 & 1 \\ \omega & \omega^2 & \omega^3 & \omega^4 \\ \omega^2 & \omega^4 & \omega^6 & \omega^8 \\ \omega^3 & \omega^6 & \omega^9 & \omega^{12} \end{bmatrix} = \begin{bmatrix} 1 & 1 & 1 & 1 \\ 1 & -1 & -1 & 1 \\ 1 & -\omega & -\omega^2 & \omega^3 \\ 1 & -\omega^2 & -\omega^4 & \omega^6 \end{bmatrix}$$

$$F_x = \begin{bmatrix} I_x & D_x \\ I_x & -D_x \end{bmatrix} = \begin{bmatrix} F_x \text{ Reuse} & F_x \text{ Odd} \end{bmatrix}$$

Spectral Methods Pattern: Libraries

- Fast transform algorithms like the FFT are notoriously difficult to optimize.
- Luckily, implementations of the FFT exist for every platform. E.G:
  - FFTW and SPIRAL: Highly successful auto-tuners for FFT (and others) on PCs and workstations
  - CUFFT for Cuda on Nvidia GPUs
CP3: Dynamic Programming

- Class of problems for which the optimal solution can be built up from optimal solutions to sub-problems.
- Principle of optimality: Optimal cover for a tree consists of a best match at the root of the tree plus the optimal cover for the sub-trees starting at each input of the match.

```
Best cover for this match uses best covers for x, y, z
Choose least cost tree-cover at root
Best cover for this match uses best covers for p, z
```

Mapping a circuits into a Cell Library

```
subject tree
```

37/66

38/66
Example of Optimal Tree Covering

AOI\text{21} \quad 4 + 3 = 7

INV \quad 11 + 2 = 13

NAND2 \quad 2 + 6 + 3 = 11

INV \quad 2

NAND2 \quad 3 + 3 = 6

NAND2 \quad 3

NAND2 \quad 3

NAND2 \quad 2 + 6 + 3 = 11

INV \quad 11 + 2 = 13

NAND2 \quad 2 + 6 + 3 = 11

NAND2 \quad 3 + 3 = 6

NAND2 \quad 3

NAND2 \quad 3

NAND2 \quad 2 + 6 + 3 = 11

INV \quad 11 + 2 = 13

NAND2 \quad 2 + 6 + 3 = 11

NAND2 \quad 3 + 3 = 6

NAND2 \quad 3

NAND2 \quad 3

NAND2 \quad 2 + 6 + 3 = 11

INV \quad 11 + 2 = 13

NAND2 \quad 2 + 6 + 3 = 11

NAND2 \quad 3 + 3 = 6

NAND2 \quad 3

NAND2 \quad 3

NAND2 \quad 2 + 6 + 3 = 11

INV \quad 11 + 2 = 13

NAND2 \quad 2 + 6 + 3 = 11

NAND2 \quad 3 + 3 = 6

NAND2 \quad 3

NAND2 \quad 3

NAND2 \quad 2 + 6 + 3 = 11

INV \quad 11 + 2 = 13

NAND2 \quad 2 + 6 + 3 = 11

NAND2 \quad 3 + 3 = 6

NAND2 \quad 3

NAND2 \quad 3

NAND2 \quad 2 + 6 + 3 = 11

INV \quad 11 + 2 = 13

NAND2 \quad 2 + 6 + 3 = 11

NAND2 \quad 3 + 3 = 6

NAND2 \quad 3

NAND2 \quad 3

NAND2 \quad 2 + 6 + 3 = 11

INV \quad 11 + 2 = 13

NAND2 \quad 2 + 6 + 3 = 11

NAND2 \quad 3 + 3 = 6

NAND2 \quad 3

NAND2 \quad 3

NAND2 \quad 2 + 6 + 3 = 11

INV \quad 11 + 2 = 13

NAND2 \quad 2 + 6 + 3 = 11

NAND2 \quad 3 + 3 = 6

NAND2 \quad 3

NAND2 \quad 3

NAND2 \quad 2 + 6 + 3 = 11

INV \quad 11 + 2 = 13

NAND2 \quad 2 + 6 + 3 = 11

NAND2 \quad 3 + 3 = 6

NAND2 \quad 3

NAND2 \quad 3

NAND2 \quad 2 + 6 + 3 = 11

INV \quad 11 + 2 = 13

NAND2 \quad 2 + 6 + 3 = 11

NAND2 \quad 3 + 3 = 6

NAND2 \quad 3

NAND2 \quad 3

NAND2 \quad 2 + 6 + 3 = 11

INV \quad 11 + 2 = 13

NAND2 \quad 2 + 6 + 3 = 11

NAND2 \quad 3 + 3 = 6

NAND2 \quad 3

NAND2 \quad 3

NAND2 \quad 2 + 6 + 3 = 11

INV \quad 11 + 2 = 13

NAND2 \quad 2 + 6 + 3 = 11

NAND2 \quad 3 + 3 = 6

NAND2 \quad 3

NAND2 \quad 3

NAND2 \quad 2 + 6 + 3 = 11

INV \quad 11 + 2 = 13

NAND2 \quad 2 + 6 + 3 = 11

NAND2 \quad 3 + 3 = 6

NAND2 \quad 3

NAND2 \quad 3

NAND2 \quad 2 + 6 + 3 = 11

INV \quad 11 + 2 = 13

NAND2 \quad 2 + 6 + 3 = 11

NAND2 \quad 3 + 3 = 6

NAND2 \quad 3

NAND2 \quad 3

NAND2 \quad 2 + 6 + 3 = 11

INV \quad 11 + 2 = 13

NAND2 \quad 2 + 6 + 3 = 11

NAND2 \quad 3 + 3 = 6

NAND2 \quad 3

NAND2 \quad 3

NAND2 \quad 2 + 6 + 3 = 11

INV \quad 11 + 2 = 13

NAND2 \quad 2 + 6 + 3 = 11

NAND2 \quad 3 + 3 = 6

NAND2 \quad 3

NAND2 \quad 3

NAND2 \quad 2 + 6 + 3 = 11

INV \quad 11 + 2 = 13

NAND2 \quad 2 + 6 + 3 = 11

NAND2 \quad 3 + 3 = 6

NAND2 \quad 3

NAND2 \quad 3

NAND2 \quad 2 + 6 + 3 = 11

INV \quad 11 + 2 = 13

NAND2 \quad 2 + 6 + 3 = 11

NAND2 \quad 3 + 3 = 6

NAND2 \quad 3

NAND2 \quad 3

NAND2 \quad 2 + 6 + 3 = 11

INV \quad 11 + 2 = 13

NAND2 \quad 2 + 6 + 3 = 11

NAND2 \quad 3 + 3 = 6

NAND2 \quad 3

NAND2 \quad 3

NAND2 \quad 2 + 6 + 3 = 11

INV \quad 11 + 2 = 13

NAND2 \quad 2 + 6 + 3 = 11

NAND2 \quad 3 + 3 = 6

NAND2 \quad 3

NAND2 \quad 3

NAND2 \quad 2 + 6 + 3 = 11

INV \quad 11 + 2 = 13

NAND2 \quad 2 + 6 + 3 = 11

NAND2 \quad 3 + 3 = 6

NAND2 \quad 3

NAND2 \quad 3

NAND2 \quad 2 + 6 + 3 = 11

INV \quad 11 + 2 = 13

NAND2 \quad 2 + 6 + 3 = 11

NAND2 \quad 3 + 3 = 6

NAND2 \quad 3

NAND2 \quad 3

NAND2 \quad 2 + 6 + 3 = 11

INV \quad 11 + 2 = 13

NAND2 \quad 2 + 6 + 3 = 11

NAND2 \quad 3 + 3 = 6

NAND2 \quad 3

NAND2 \quad 3

NAND2 \quad 2 + 6 + 3 = 11

INV \quad 11 + 2 = 13

NAND2 \quad 2 + 6 + 3 = 11

NAND2 \quad 3 + 3 = 6

NAND2 \quad 3

NAND2 \quad 3

NAND2 \quad 2 + 6 + 3 = 11

INV \quad 11 + 2 = 13

NAND2 \quad 2 + 6 + 3 = 11

NAND2 \quad 3 + 3 = 6

NAND2 \quad 3

NAND2 \quad 3

NAND2 \quad 2 + 6 + 3 = 11

INV \quad 11 + 2 = 13

NAND2 \quad 2 + 6 + 3 = 11

NAND2 \quad 3 + 3 = 6

NAND2 \quad 3

NAND2 \quad 3

NAND2 \quad 2 + 6 + 3 = 11

INV \quad 11 + 2 = 13

NAND2 \quad 2 + 6 + 3 = 11

NAND2 \quad 3 + 3 = 6

NAND2 \quad 3

NAND2 \quad 3

NAND2 \quad 2 + 6 + 3 = 11

INV \quad 11 + 2 = 13

NAND2 \quad 2 + 6 + 3 = 11

NAND2 \quad 3 + 3 = 6

NAND2 \quad 3

NAND2 \quad 3

NAND2 \quad 2 + 6 + 3 = 11

INV \quad 11 + 2 = 13

NAND2 \quad 2 + 6 + 3 = 11

NAND2 \quad 3 + 3 = 6

NAND2 \quad 3

NAND2 \quad 3

NAND2 \quad 2 + 6 + 3 = 11

INV \quad 11 + 2 = 13

NAND2 \quad 2 + 6 + 3 = 11

NAND2 \quad 3 + 3 = 6

NAND2 \quad 3

NAND2 \quad 3

NAND2 \quad 2 + 6 + 3 = 11

INV \quad 11 + 2 = 13

NAND2 \quad 2 + 6 + 3 = 11

NAND2 \quad 3 + 3 = 6

NAND2 \quad 3

NAND2 \quad 3

NAND2 \quad 2 + 6 + 3 = 11

INV \quad 11 + 2 = 13

NAND2 \quad 2 + 6 + 3 = 11

NAND2 \quad 3 + 3 = 6

NAND2 \quad 3

NAND2 \quad 3

NAND2 \quad 2 + 6 + 3 = 11

INV \quad 11 + 2 = 13

NAND2 \quad 2 + 6 + 3 = 11

NAND2 \quad 3 + 3 = 6

NAND2 \quad 3

NAND2 \quad 3
Speech Recognition

<table>
<thead>
<tr>
<th>Observations</th>
<th>Time</th>
</tr>
</thead>
<tbody>
<tr>
<td>Wreck</td>
<td></td>
</tr>
<tr>
<td>Recognize</td>
<td></td>
</tr>
<tr>
<td>a</td>
<td></td>
</tr>
<tr>
<td>nice</td>
<td></td>
</tr>
<tr>
<td>beach</td>
<td></td>
</tr>
<tr>
<td>speech</td>
<td></td>
</tr>
</tbody>
</table>

Outline

- Architecting Parallel Software
- Structural Patterns
- Computational Patterns
- Examples
  - Classification using Support Vector Machines
    - Speech Recognition
  - Summary

© Kurt Keutzer
Feature Extraction

- Image is reduced to a set of low-dimensional feature vectors

- Build Scale-Space Representation
- Select Interest Points and Support Regions
- Build Descriptors
- Map Reduce
- Map Reduce
- Structured Grid
- Structured Grid

"Image Feature Extraction for Mobile Processors", Mark Murphy, Hong Wang, Kurt Keutzer IISWC '09
Train Classifier: SVM Training

- Train Classifier
- Update Optimality Conditions
- Select Working Set, Solve QP
- MapReduce
- Iterate
- Iterator Pattern

Exercise Classifier: SVM Classification

- Test Data
- SV
- Compute dot products
- Compute Kernel values, sum & scale
- Output
- Dense Linear Algebra
- MapReduce
Support-Vector Machine Mini-Framework

- Support-Vector Machine Framework used to achieve:
  - 9-35x speedup for training
  - 81-138x for classification
  - 2200 downloads since release

Fast support vector machine training and classification, Catanzaro, Sundaram, Keutzer, International Conference on Machine Learning 2008

Outline

- Architecting Parallel Software
- Structural Patterns
- Computational Patterns
- Examples
  - CBIR
  - Speech Recognition
- Summary
- Inference engine based system
  - Used in Sphinx (CMU, USA), HTK (Cambridge, UK), and Julius (CSRC, Japan)
- Modular and flexible setup
  - Shown to be effective for Arabic, English, Japanese, and Mandarin

- Architecting Parallel Software
  - Decompose Tasks
    - Group tasks
    - Order Tasks
  - Decompose Data
    - Data sharing
    - Data access

Identify the Software Structure

Identify the Key Computations
Recognition is a process of graph traversal/dynamic programming. Each time-step we need to identify the likely states in the recognition network given the observation acoustic signal. From a set of active states we want to compute the next set of active states using probabilities of acoustic symbols and state transitions. What **Structural pattern** is this?

**Inference Engine Architecture**

- Inference Engine Architecture

```
<table>
<thead>
<tr>
<th>Structural Patterns</th>
<th>Model-View-Controller</th>
</tr>
</thead>
<tbody>
<tr>
<td>Pipe-and-Filter</td>
<td>Iterative Refinement</td>
</tr>
<tr>
<td>Agent-and-Repository</td>
<td>Map-Reduce</td>
</tr>
<tr>
<td>Process-Control</td>
<td>Layered-Systems</td>
</tr>
<tr>
<td>Event-Based/Implicit Invocation</td>
<td>Arbitrary-Static-Task-Graph</td>
</tr>
<tr>
<td>Puppeteer</td>
<td></td>
</tr>
</tbody>
</table>
```

**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

```
<table>
<thead>
<tr>
<th>Structural Patterns</th>
<th>Model-View-Controller</th>
</tr>
</thead>
<tbody>
<tr>
<td>Pipe-and-Filter</td>
<td>Iterative Refinement</td>
</tr>
<tr>
<td>Agent-and-Repository</td>
<td>Map-Reduce</td>
</tr>
<tr>
<td>Process-Control</td>
<td>Layered-Systems</td>
</tr>
<tr>
<td>Event-Based/Implicit Invocation</td>
<td>Arbitrary-Static-Task-Graph</td>
</tr>
<tr>
<td>Puppeteer</td>
<td></td>
</tr>
</tbody>
</table>
```
Key computation: HMM Inference Algorithm

- Finds the most-likely sequence of states that produced the observation

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

Viterbi Algorithm

Legends:
- A State
- An Observation
- \( P(x_t|s_t) \)
- \( m[t][s_t] \)
- \( P(s_t|s_{t-1}) \)
- \( m[t][s_{t-1}] \)

Markov Condition:

\[ m[
\frac{t}{|s_{t}|} = \max_{s_{t-1}} P(x_t, x_{t+1}, \ldots, x_{t-1} | s_t) \]


Inference Engine in LVCSR

- Structural pattern to support three phases of inference
  0. Gather operands from irregular data structure to runtime buffer
  1. Perform observation probability computation
  2. Perform graph traversal computation

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

Structural Patterns
- Pipelined Filter
- Model-View-Controller
- Iterative-Refinement
- Agent and Repository
- Map-Reduce
- Process Control
- Layered Systems
- Event-Based/Implicit Invocation
- Puppeteer

Arbitrary Static Task Graph
Each Filter is a Map Reduce

0. Gather operands

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

- Gather and coalesce each of the above operands for every \( s_t \)
- Facilitates opportunity for SIMD

Gather operand

Coalesced data

Each Filter is Map Reduce

1. Observation probability computation

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

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

\( P(x_t|s_t) \)

GMM probability
Phase 1: Observation Probability computation Architecture

- Observation probabilities 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

Map-Reduce Structural Pattern

- Map each mixture probability computation
- Reduce the result – accumulate the total probability for that state
Each Filter is Map Reduce

2. graph traversal computation

- **Map** probability computation across distributed data sets – perform multiplication as below
- **Reduce** the results to find the maximum likely states

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

LVCSR Software Architecture

Inference Engine

 Beam Search Iterations

Active State Computation Steps

Pipe and Filter

MapReduce

Iterative Refinement

Graphical Model

Dynamic Programming

Word Sequence

I think therefore I am
### Speech Recognition Results

- **Input:** Speech audio waveform
- **Output:** Recognized word sequences

- Achieved 11x speedup over sequential version
- Allows 3.5x faster than real time recognition

- Our technique is being deployed in a hotline call-center data analytics company
- Used to search content, track service quality and provide early detection of service issues

Scalable HMM based Inference Engine in Large Vocabulary Continuous Speech Recognition, Kisun You, Jike Chong, Youngmin Yi, Ekaterina Gonina, Christopher Hughes, Wonyong Sung and Kurt Keutzer, IEEE Signal Processing Magazine, March 2010

---

### Multi-media Speech Recognition

- **Prof. Dorothea Kolossa**
  - *Speech Application Domain Expert*
  - Technische Universität Berlin

Extended audio-only speech recognition framework to enable audio-visual speech recognition (lip reading)

Achieved a **20x speedup** in application performance compared to a sequential version in C++

The application framework enabled a *Matlab/Java programmer* to **effectively utilize highly parallel platform**

Dorothea Kolossa, Jike Chong, Steffen Zeiler, Kurt Keutzer, “Efficient Manycore CHMM Speech Recognition for Audiovisual and Multistream Data”, Accepted at Interspeech 2010.
Other Interesting Results

- Patterns have helped the PALLAS research group publish papers in a diverse group of leading Computer Science conferences in the last few years:
  - Interspeech 2009, Interspeech 2010 (2)
  - IEEE Signal Processing Magazine 2009
  - European Conference on Computer Vision 2010
  - International Conference on Computer Vision 2009
  - International Parallel and Distributed Processing Symposium 2012, 2011, 2009 (2)
  - Workshop on High Performance Computing in Finance at Super Computing 2009
  - Joint Annual Meeting of the International Society for Magnetic Resonance in Medicine, ISMRM 2010
  - International Conference on Machine Learning 2008
  - IEEE Transactions on Medical Imaging (2012)
- What’s the point?
  - Computational patterns give a new powerful viewpoint to efficiency programmers:
  - Enable us to disentangle the big fuzzy ball of yarn of computation
    - add 20 IQ points to our problem solving (as per Alan Kay)
  - Our Pattern language helps you to write good parallel code
## Speedups

<table>
<thead>
<tr>
<th>Application</th>
<th>Speedups</th>
</tr>
</thead>
<tbody>
<tr>
<td>MRI</td>
<td>100x</td>
</tr>
<tr>
<td>SVM-train</td>
<td>20x</td>
</tr>
<tr>
<td>SVM-classify</td>
<td>109x</td>
</tr>
<tr>
<td>Contour</td>
<td>130x</td>
</tr>
<tr>
<td>Object Recognition</td>
<td>80x</td>
</tr>
<tr>
<td>Poselet</td>
<td>20x</td>
</tr>
<tr>
<td>Optical Flow</td>
<td>32x</td>
</tr>
<tr>
<td>Speech</td>
<td>11x</td>
</tr>
<tr>
<td>Value-at-risk</td>
<td>60x</td>
</tr>
<tr>
<td>Option Pricing</td>
<td>25x</td>
</tr>
</tbody>
</table>

## 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 three – seven more will be all you need
  - Computational patterns: provide efficiency, identify key computations to be parallelized
    - I showed you three – ten more will be all you need
  - Orchestration of computational and structural patterns creates architectures which greatly facilitates the development of parallel programs:
    - I showed you three – there are many more

Patterns: [http://parlab.eecs.berkeley.edu/wiki/patterns/patterns](http://parlab.eecs.berkeley.edu/wiki/patterns/patterns)

PALLAS: [http://parlab.eecs.berkeley.edu/research/pallas](http://parlab.eecs.berkeley.edu/research/pallas)