Main Points of Previous Lecture

- Many approaches to parallelizing software are not working
  - Profile and improve
  - Swap in a new parallel programming language
  - Rely on a super parallelizing compiler
- My own experience has shown that a sound software architecture is the greatest single indicator of a software project’s success.
- Software must be architected to achieve productivity, efficiency, and correctness
- SW architecture >> programming environments
  - >> programming languages
  - >> compilers and debuggers
  - (>>hardware architecture)
- If we had understood how to architect sequential software, then parallelizing software would not have been such a challenge
- Key to architecture (software or otherwise) is design patterns and a pattern language
- At the highest level our pattern language has:
  - Eight structural patterns
  - Thirteen computational patterns
- Yes, we really believe arbitrarily complex parallel software can built just from these!
Outline

- Our Pattern Language for parallel programming
- Detailed example using Our Pattern Language

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!!!
Patterns embody generalizable solutions to recurrent problems.
Collections of individual patterns (e.g., Design Patterns, Gamma, Helm, Johnson, Vlissides) are great, but we need more help in the software development enterprise.
We would like a comprehensive pattern language which covers the entire process of parallel software development and implementation.
Keutzer, Mattson, the PALLAS group, and others have developed just such a pattern language.

- [http://parlab.eecs.berkeley.edu/wiki/patterns](http://parlab.eecs.berkeley.edu/wiki/patterns)
- Pattern language is overviewed in:
- Today we don't have time to go through all the patterns, but we will briefly describe the structure of OPL and show how it can be used.
Architecting Parallel Software

Application

Identify the Software Structure

- Pipe-and-Filter
- Agent-and-Repository
- Event-based
- Bulk Synchronous
- MapReduce
- Layered Systems
- Model-view controller
- Arbitrary Task Graphs
- Puppeteer
- Model-view-controller

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
- Monte-Carlo

Our Pattern Language 2010: Details

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

Compositional 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

Parallel Algorithm Strategy Patterns
- Task-Parallelism
- Data-Parallelism
- Divide and Conquer
- Pipeline
- Speculation
- Geometric-Decomposition

Implementation Strategy Patterns
- SPMD
- Data-Parallelizable
- Fork/Join
- Program structure

Parallel Execution Patterns
- Thread-Pool
- Task-Queue
- Transactions
Our Pattern Language: At a High Level

Applications

Structural Patterns

Computational Patterns

Parallel Algorithm Strategy Patterns

Implementation Patterns

Execution Patterns

The Algorithm Strategy Design Space

Start

Organize By Flow of Data
- Regular?
- Irregular?
- Pipeline
- Discrete Event

Organize By Tasks
- Linear?
- Recursive?
- Task Parallelism
- Divide and Conquer

Organize By Data
- Linear?
- Recursive?
- Geometric Decomposition
- ???

Speculation

Parallel Algorithm Strategy Patterns
- Task-Parallelism
- Divide and Conquer
- Data-Parallelism
- Pipeline
- Discrete-Event
- Geometric-Decomposition
- Speculation
Our Pattern Language: At a High Level

Applications

Structural Patterns ↔ Computational Patterns

Parallel Algorithm Strategy Patterns

Implementation Patterns

Execution Patterns

- Parallel Execution Patterns
  - MIMD
  - SIMD
- Thread-Pool
- Task-Graph
- Transactions
Outline

- Our Pattern Language for parallel programming
- Detailed example using Our Pattern Language

Outline

- Speech Recognition Application
  - Software Architecture using Patterns
    - Identify Structural Patterns
    - Identify Computational Patterns
  - Parallelization: (for each module)
    - Algorithm strategy pattern
    - Implementation strategy pattern
    - Execution patterns
  - Conclusion
Automatic Speech Recognition

- Key technology for enabling rich human-computer interaction
  - Increasingly important for intelligent devices without keyboards
- Interaction requires low latency responses
  - Only one of many components in exciting new real-time applications

- Main contributions:
  - A detailed analysis of the concurrency in large vocabulary continuous speech recognition (LVCSR) summarized in the pattern language
  - An implementation on GPU with 11x speedup over optimized C++ version
  - Achieving 3x better than real time performance with 50,000 word vocabulary

Continuous Speech Recognition

- Challenges:
  - Recognizing words from a large vocabulary arranged in exponentially many possible permutations
  - Inferring word boundaries from the context of neighboring words
  - Hidden Markov Model (HMM) is the most successful approach
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

Recognition Process
- Considers a sequence of features one at a time, for each feature:
  - Start with most-likely-sequences ending with previous feature
    - Corresponding to a set of active states
  - Consider the likelihood of current feature in its context
  - Compute most-likely-sequences ending with current feature
    - Corresponding to the next set of active states

Recognition is a process of graph traversal
Recognition Network

- Features from one frame
- Gaussian Mixture Model for One Phone State
  - Mixture Components
  - Computing distance to each mixture components
  - Computing weighted sum of all components
- HMM Acoustic Phone Model
- Pronunciation Model
  - HOP hh aa p
  - ON aa n
  - POP p aa p
- Bigram Language Model

Compiled HMM Recognition Network

Graph Traversal Characteristics

- Operations on a graph is driven by the graph structure
  - Irregular graph structures – challenging to statically load balancing
  - Irregular state access pattern – challenging to dynamically load balancing
  - Multiple states may share the same next state – write contentions in arc evaluation
  - Traversal produces memory accesses that spans the entire graph – poor spatial locality
  - Inner loops have high data access to computation ratio – limited by memory bandwidth
Parallel Platform Characteristics

- Multicore/manycore design philosophy
  - **Multicore**: Devote significant transistor resources to single thread performance
  - **Manycore**: Maximizing computation throughput at the expense of single thread performance

- Architecture Trend:
  - Increasing vector unit width
  - Increasing numbers of cores per die

- Application Implications:
  - Must increase data access regularity
  - Must optimize synchronization cost

Outline

- Speech Recognition Application
- **Software Architecture using Patterns**
  - Identify Structural Patterns
  - Identify Computational Patterns
- Parallelization: (for each module)
  - Algorithm strategy pattern
  - Implementation strategy pattern
  - Execution patterns
- Conclusion
OPL 2010

Applications

Structural Patterns
- Pipe-and-Filter
- Agent-and-Repository
- Process-Control
- Event-Based/Implicit-Invocation
- Puppeteer

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

Model-View-Controller
- Iterative-Refinement
- Map-Reduce
- Layered-Systems
- Arbitrary-Static-Task-Graph

Parallel Algorithm Strategy Patterns
- Task-Parallelism
  - Divide and Conquer
- Data-Parallelism
  - Pipeline
- Discrete-Event
  - Geometric-Decomposition
  - Speculation

Implementation Strategy Patterns
- SPMD
  - Data-Par/index-space
- Data-Par/index-space
  - Fork/Join Actors
- Loop-Par.
  - Task-Queue
- Shared-Queue
  - Shared-map
- Partitioned Graph
- Distributed-Array
  - Shared-Data
- Data structure

Parallel Execution Patterns
- MIMD
- SIMD
  - Thread-Pool
  - Task-Queue
  - Transactions

Concurrency Foundation constructs (not expressed as patterns)
- Thread creation/destruction
- Process creation/destruction
- Message-Passing
- Collective-Comm.
- Transactional memory
- Point-To-Point-Sync. (mutual exclusion)
- collective sync. (barrier)
- Memory sync/fence

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. Each time-step we need to identify the likely states in the recognition network given the observation acoustic signal. From the 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?

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
- \( P(x_t|s_t) \)
- \( m_{t-1}[s_{t-1}] \)
- \( P(s_t|s_{t-1}) \)
- \( m_t[s_t] \)

Markov Condition:

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

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

Digging Deeper – Active Set Computation Architecture

- In each iteration we need to:
  - Compute observation probabilities of transitions from current states
  - Traverse the likely non-epsilon arcs to reach the set of next active states
  - Traverse the likely epsilon arcs to reach the set of next active states
- What **structural pattern** is this?
In each iteration we need to:
- Compute observation probabilities of transitions from current states
- Traverse the likely non-epsilon arcs to reach the set of next active states
- Traverse the likely epsilon arcs to reach the set of next active states

What **Structural pattern** is this?

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

What **Structural pattern** is this?
### Map-Reduce Structural Pattern

- Map each mixture probability computation
- Reduce the result - accumulate the total probability for that state

**Gaussian Mixture Model for One Phone State**

**Phase 2: Graph Traversal**

- Now that we know the transition probabilities from current set of states, we need to compute the next set of active states (follow the likely transitions)
- Each transition is independent
- Multiple transitions might end in the same state
- The end result needs to be a set of most probable states from all transitions
- What **Structural pattern** is this?

![Viterbi Algorithm](image-url)
Map-Reduce Structural Pattern - again

- Map each mixture probability computation
- Reduce the result - accumulate the total probability for that state

\[ m[i][s] = \max_{k-1} m[i][s_{k-1}] \cdot P(s_i|s_{k-1}) \cdot P(x_i|s_i) \]

Viterbi Algorithm

Inference Engine

- Phase 1: Observation Probability Computation
- Phase 2: Graph Traversal

Structural Patterns

- Model-View-Controller
- Iterative Refinement
- Map-Reduce
- Layered Systems
- Arbitrary-Static-Task-Graph
- Puppeteer

High Level Structure of Engine
### Outline

- Speech Recognition Application
- Software Architecture using Patterns
  - Identify Structural Patterns
  - **Identify Computational Patterns**
- Parallelization: (for each module)
  - Algorithm strategy pattern
  - Implementation strategy pattern
  - Execution patterns
- Conclusion

### What about Computation?

- Active Set Computation:
  - Phase 1: Compute observation probability of transitions given current set of states
  - Phase 2: Traverse arcs to determine next set of most likely active states
- What **Computational Patterns** are these?

<table>
<thead>
<tr>
<th>Computational Patterns</th>
<th>Graphical-Models</th>
</tr>
</thead>
<tbody>
<tr>
<td>Graph-Algorithm</td>
<td>Finite-State-Machines</td>
</tr>
<tr>
<td>Dynamic-Programming</td>
<td>Backtrack-Branch-and-Bound</td>
</tr>
<tr>
<td>Dense-Linear-Algebra</td>
<td>N-Body-Methods</td>
</tr>
<tr>
<td>Sparse-Linear-Algebra</td>
<td>Circuits</td>
</tr>
<tr>
<td>Unstructured-Grids</td>
<td>Spectral-Methods</td>
</tr>
<tr>
<td>Structured-Grids</td>
<td>Monte-Carlo</td>
</tr>
</tbody>
</table>
Outline

- Speech Recognition Application
- Software Architecture using Patterns
  - Identify Structural Patterns
  - Identify Computational Patterns

**Parallelization: (for each module)**
- Algorithm strategy pattern
- Implementation strategy pattern
- Execution patterns
- Conclusion

Now to Parallelism – Inference Engine

- Structural Patterns: Iterative Refinement
- Computational Patterns: Dynamic Programming
- Inference engine and Active Set computation is sequential
  - Let's look at Phase 1 and Phase 2

- What Parallel Algorithm Strategy can we use?

<table>
<thead>
<tr>
<th>Parallel Algorithm Strategy Patterns</th>
<th>Task-Parallelism</th>
<th>Data-Parallelism</th>
<th>Discrete-Event</th>
</tr>
</thead>
<tbody>
<tr>
<td>Divide and Conquer</td>
<td>Pipeline</td>
<td>Geometric-Decomposition</td>
<td></td>
</tr>
<tr>
<td>Speculation</td>
<td></td>
<td>Speculation</td>
<td></td>
</tr>
</tbody>
</table>
Now to Parallelism – Inference Engine

- Phase 1: Observation Probability Computation
  - Structural: MapReduce
  - Computational: Graphical Models
- Compute cluster Gaussian Mixture Probabilities for each transition label
- Hint:
  - Look at data dependencies (or lack thereof)

Parallel Algorithm Strategy Patterns

<table>
<thead>
<tr>
<th>Task-Parallelism</th>
<th>Data-Parallelism</th>
<th>Discrete-Event</th>
</tr>
</thead>
<tbody>
<tr>
<td>Divide and Conquer</td>
<td>Pipeline</td>
<td>Geometric-Decomposition</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Speculation</td>
</tr>
</tbody>
</table>

(map of Viterbi Algorithm and states)

Parallel Algorithm Strategy Patterns

<table>
<thead>
<tr>
<th>Task-Parallelism</th>
<th>Data-Parallelism</th>
<th>Discrete-Event</th>
</tr>
</thead>
<tbody>
<tr>
<td>Divide and Conquer</td>
<td>Pipeline</td>
<td>Geometric-Decomposition</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Speculation</td>
</tr>
</tbody>
</table>
Gaussian Mixture Model is shared among all computations – read only
### Observation Probability Pattern

#### Structural Patterns
- Pipe-and-Filter
- Agent-and-Repository
- Process-Control
- Event-Based/Implicit-Invocation
- Pupeteer

#### Computational Patterns
- Graph-Algorithms
- Dynamic-Programming
- Dense-Linear-Algebra
- Sparse-Linear-Algebra
- Unstructured-Grids
- Structured-Grids

#### Parallel Algorithm Strategy Patterns
- Task-Parallelism
- Divide and Conquer
- Data-Parallelism
- Pipeline

#### Implementation Strategy Patterns
- SPMD
- Data-Parallelism
- Fork/Join
- Actor

#### Parallel Execution Patterns
- MIMD
- SIMD
- Loop-Par.
- Task-Queue

#### Program structure?

---

**Data parallel – a single instruction stream is applied to multiple data elements**
Execution? – a single instruction stream is applied to multiple data elements
Phase 2: Graph Traversal
- Structural: MapReduce
- Computational: Graph algorithms/graph traversal
- The recognition network is a finite state transducer, represented as a weighted and labeled graph
- Decoding on this graph is Breadth-First Traversal
- What Parallel Algorithmic Strategy can we use?

Hint:
- What are the operands?
- What are the dependences?

Data Parallelism!
Each next state computation can be computed independently.
**Active Set Computation**

**Structural Patterns**
- Pipe-and-Filter
- Agent-and-Repository
- Process-Control
- Event-Based/Implicit-Invocation
- Puppeteer

**Model-View-Controller**
- Iterative-Refinement
- Layered-Systems
- Arbitrary-Static-Task-Graph

**Computational Patterns**
- Graph-Algorithms
- Dynamic-Programming
- Dense-Linear-Algebra
- Sparse-Linear-Algebra
- Unstructured-Grids
- Structured-Grids

**Computational Patterns**
- Finite-State-Machines
- Backtrack-Branch-and-Bound
- N-Body-Methods
- Circuits
- Spectral-Methods
- Monte-Carlo

**Parallel Algorithm Strategy Patterns**
- Task-Parallelism
- Divide and Conquer
- Data-Parallelism
- Pipeline

**Implementation Strategy Patterns**
- SPMD
  - Fork/Join
  - Task-Queue
- Loop-Par
  - Task-Queue

**Parallel Execution Patterns**
- Thread-Pool
- Task-Graph
- Transactions

**Shared Queue Implementation**

- In each iteration:
  - Arc transitions from the current set of states are traversed to construct a set of next active states
  - Each thread enqueues one or more next active states to a **global** queue of next active states
  - This approach causes significant **contention** at the queue head pointer
    - In our application - thousands of threads are accessing one memory location
    - Causes serialization of thread memory accesses
    - Limits scalability

**SIMD**

```
Each thread enqueues one or more next active states to a global queue of next active states
```

```
This approach causes significant contention at the queue head pointer
```

```
Limits scalability
```
Shared Queue Implementation

- Solution to the queue head pointer contention -> distributed queue
  1. Each thread in a thread block writes to a local queue of next active states
  2. Local queues are merged into a global queue

- Contention on global queue pointer is reduced from #threads to #blocks
- Significantly improves scalability

Speech Reference Implementation

Summary

- A good architect needs to understand:
  - Structural patterns
  - Computational patterns
  - Refinement through Our Pattern Language
- Graph algorithms and graphical models are critical to many applications
- Graph algorithms are especially difficult to parallelize and library support is inadequate
- There will be at least a decade of hand-crafted solutions
- We achieved good results on parallelizing large-vocabulary automatic speech recognition

Extras
**Example: Breadth First Search**

```cpp
template<class IncidenceGraph, class Buffer, class BFSVisitor, class ColorMap>

breadth_first_search (const IncidenceGraph& g,  
                        typedef graph_traits<IncidenceGraph>::vertex_descriptor s,  
                        Buffer& Q, BFSVisitor vis, ColorMap color)
{
    for (color[s] = Color::gray; Q.push(s);)
        while (!Q.empty()) {
            for (size_t i = 0; i < g.out_edges(s).size(); i++) {
                color[g.target(i)] = Color::gray;
            }
            color[s] = Color::white;
            while (!Q.empty()) {
                color[Q.front()] = Color::gray;
                Q.pop();
            }
            color[s] = Color::black;
            vis.discover_vertex(s, g);
            for (size_t i = 0; i < g.out_edges(s).size(); i++) {
                vis.examine_edge(i, g);
            }
            vis.finish_vertex(s, g);
        }
```

**Example: Distributed BFS**

- **Distributed Graph**
- **Distributed Queues**
- **Distributed Visitors**
- **Distributed Property Map**
Example: Distributed BFS

- Distributed Queues
  - `pop()` from local queues
  - `push()` sends message to the vertex owner
  - `empty()` exhaust local queue and synchronize with other processors to determine termination condition
  - Messages are only received after all processors have completed operation at one level

(b) Distributed adjacency list representation

Example: Distributed BFS

- Distributed Visitors
  - Owner-compute scheme, so distributed version is the same as sequential visitor

- Distributed Property Map
  - Local Property Map
    - Store local properties for local vertices and edges
  - Ghost Cells
    - Store properties for ghost cells
  - Process Group
    - Communication medium
  - Data Race Resolver
    - Decides among various `put()` messages sent to the Distributed Property Map

(a) Distributed graph

(only in distributed environment; more complex in non-distributed)}
Performance achieved

Graph traversal algorithm characteristic:
1. Input data-driven computation
2. Unstructured problems
3. Poor data locality
4. High data access to computation ratio

High demand for low memory latency and poor data locality makes it challenging for PEs without fine-grain multiprocessing

Pointer chasing mainly involves integer operations
- Niagara type platforms seems suitable for this application domain
- What about GPGPU?

What how well would our scheduling algorithms map to Parallel BGL?