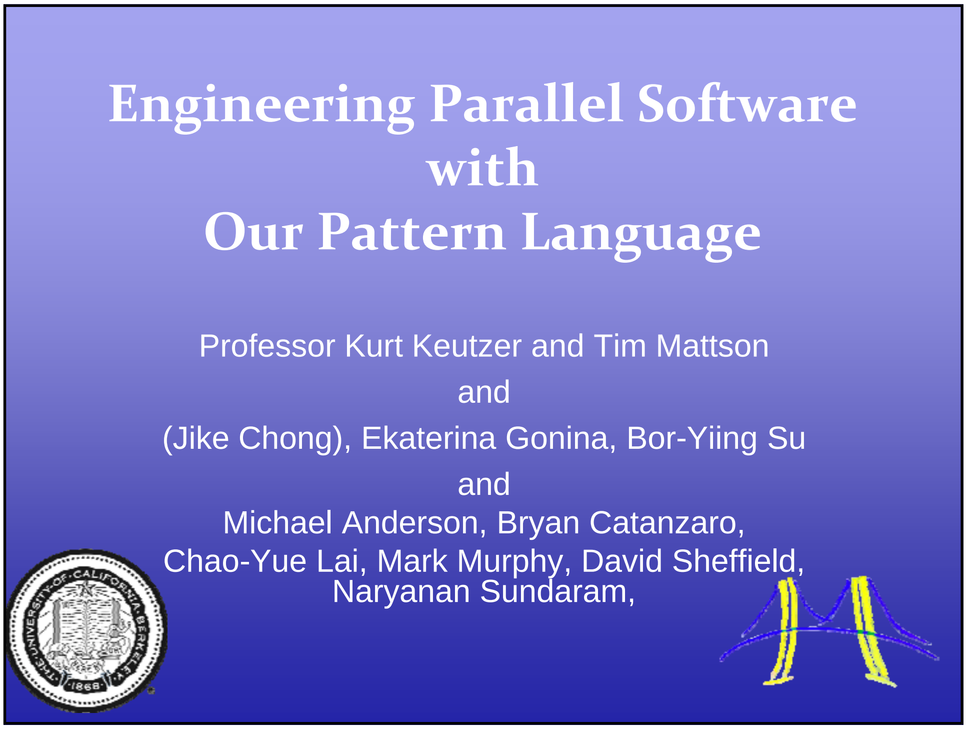
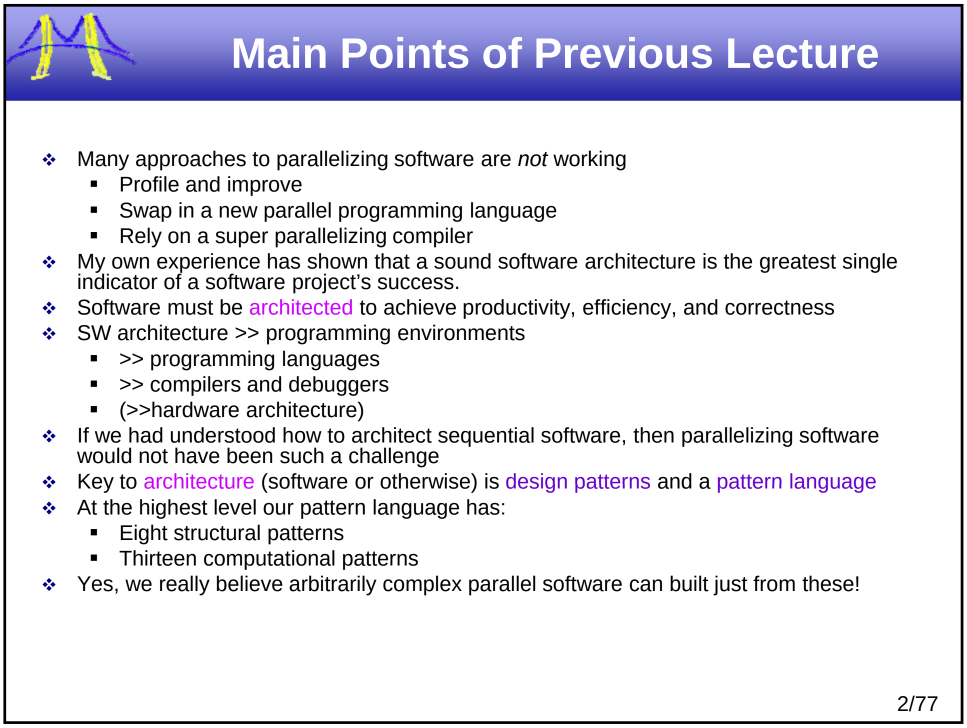


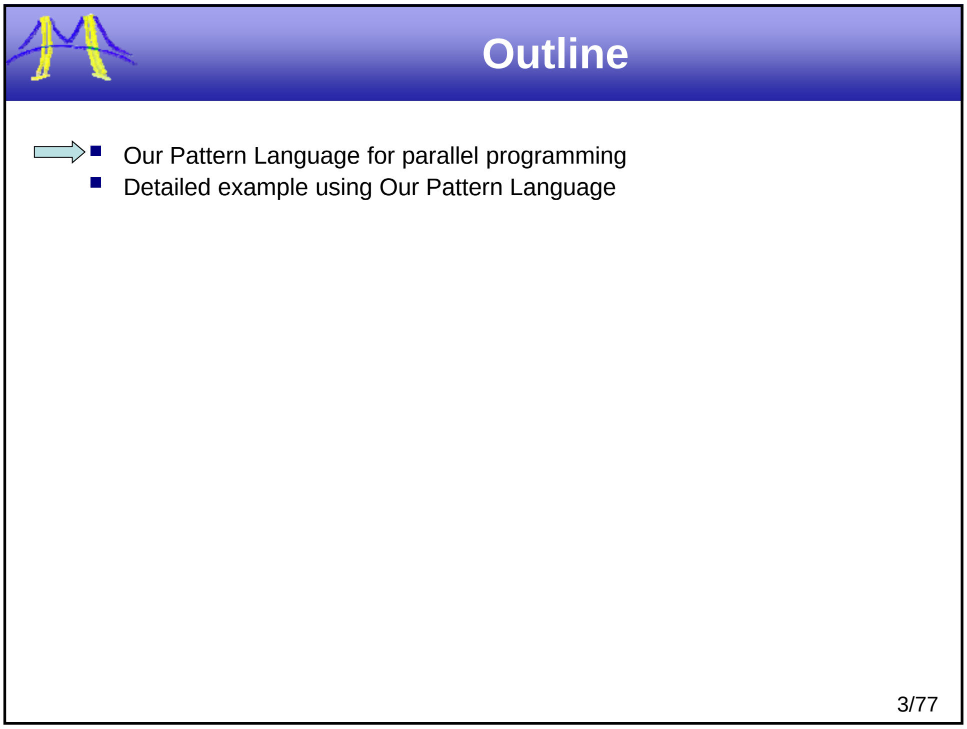
Engineering Parallel Software with Our Pattern Language

Professor Kurt Keutzer and Tim Mattson
and
(Jike Chong), Ekaterina Gonina, Bor-Yiing Su
and
Michael Anderson, Bryan Catanzaro,
Chao-Yue Lai, Mark Murphy, David Sheffield,
Naryanan Sundaram,



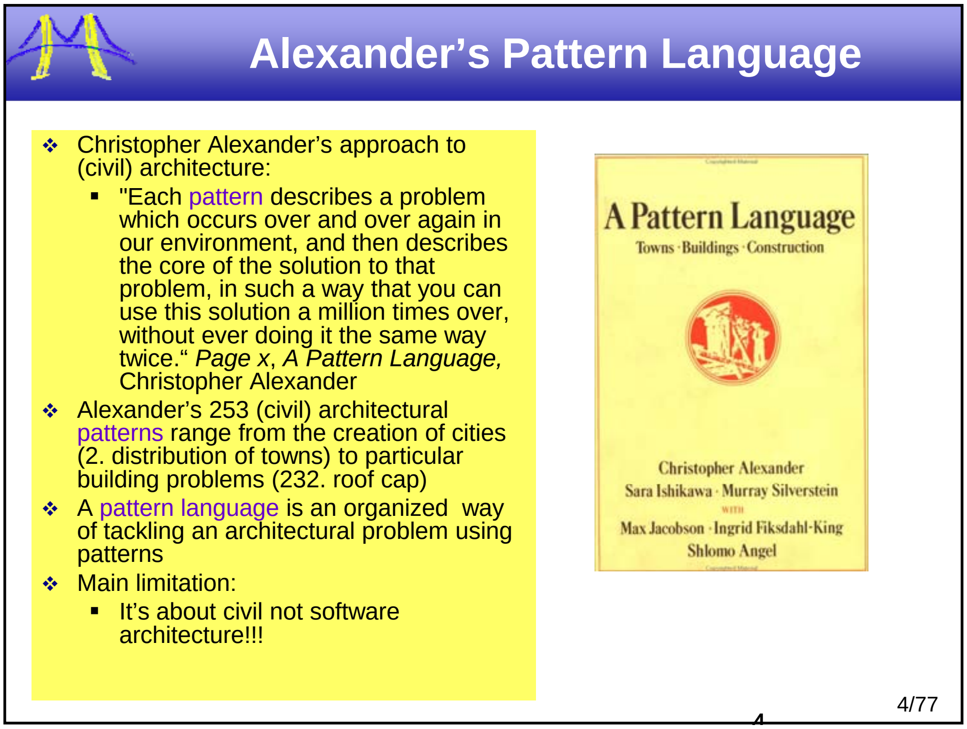
Main Points of Previous Lecture

- 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!!!

A Pattern Language

Towns · Buildings · Construction

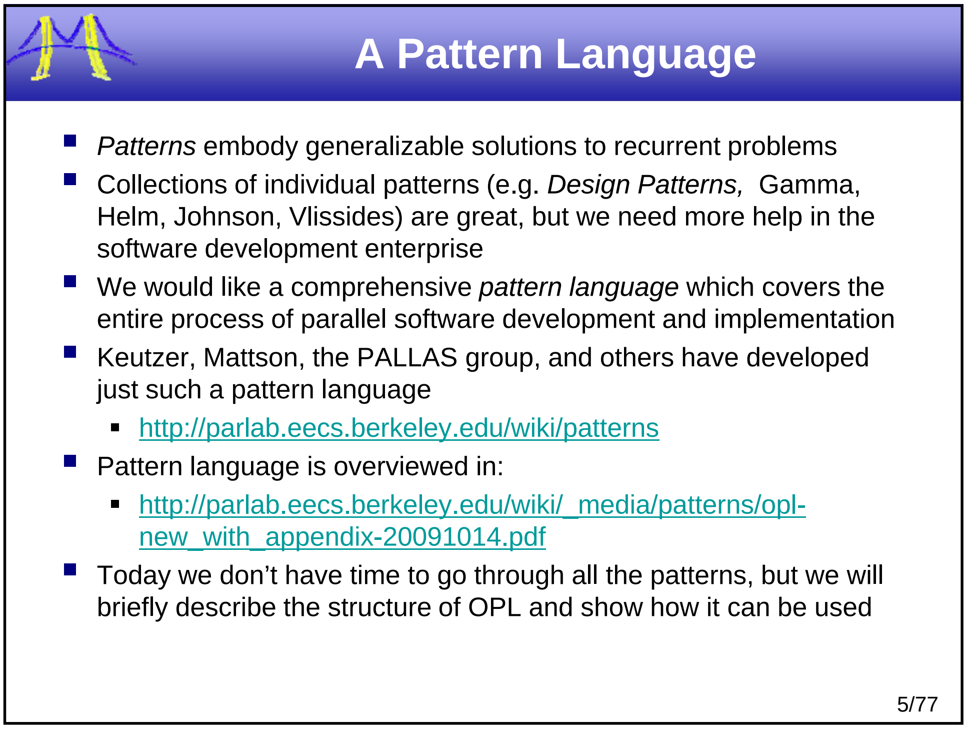
Christopher Alexander

Sara Ishikawa · Murray Silverstein

WITH

Max Jacobson · Ingrid Fiksdahl-King

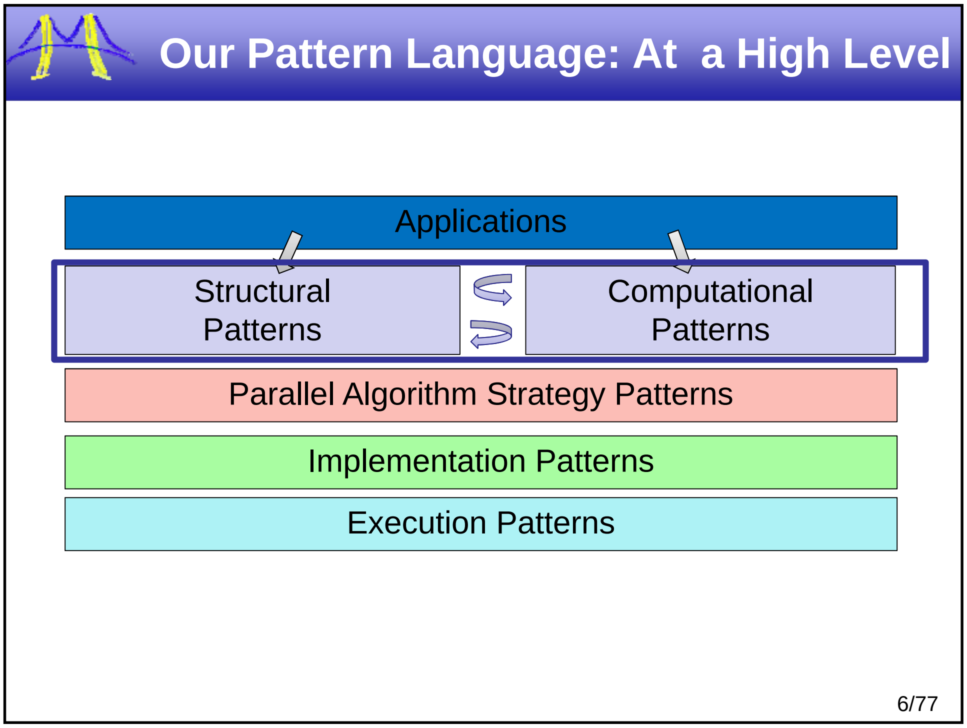
Shlomo Angel



A Pattern Language

- *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
- Pattern language is overviewed in:
  - http://parlab.eecs.berkeley.edu/wiki/_media/patterns/opl-new_with_appendix-20091014.pdf
- 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

5/77



Our Pattern Language: At a High Level

Applications

Structural Patterns

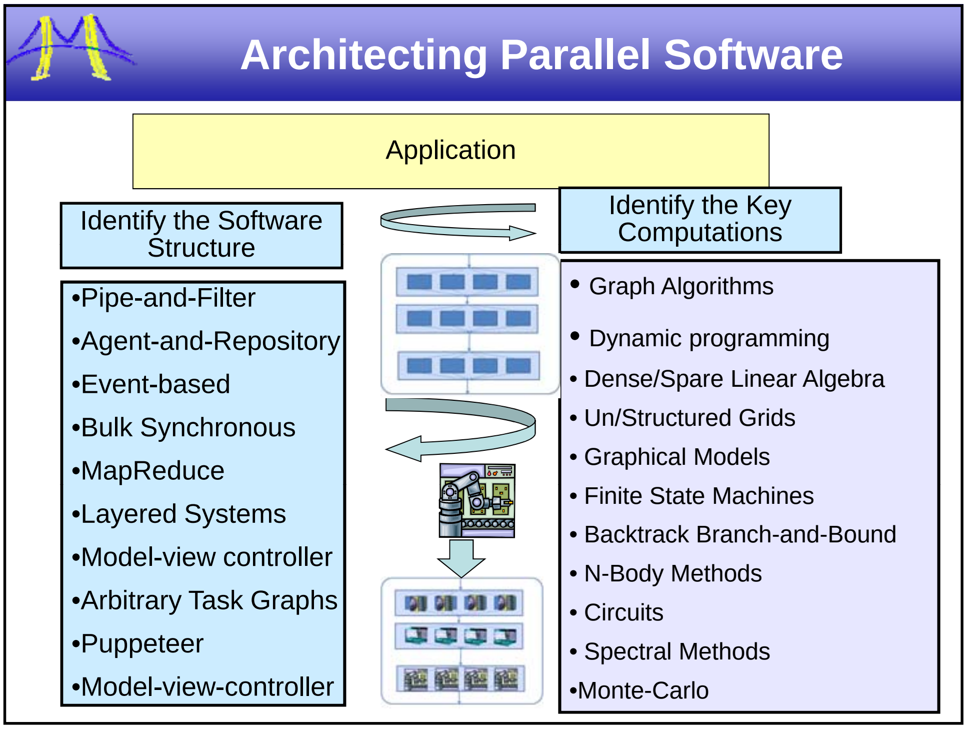
Computational Patterns

Parallel Algorithm Strategy Patterns

Implementation Patterns

Execution Patterns

6/77



Architecting Parallel Software

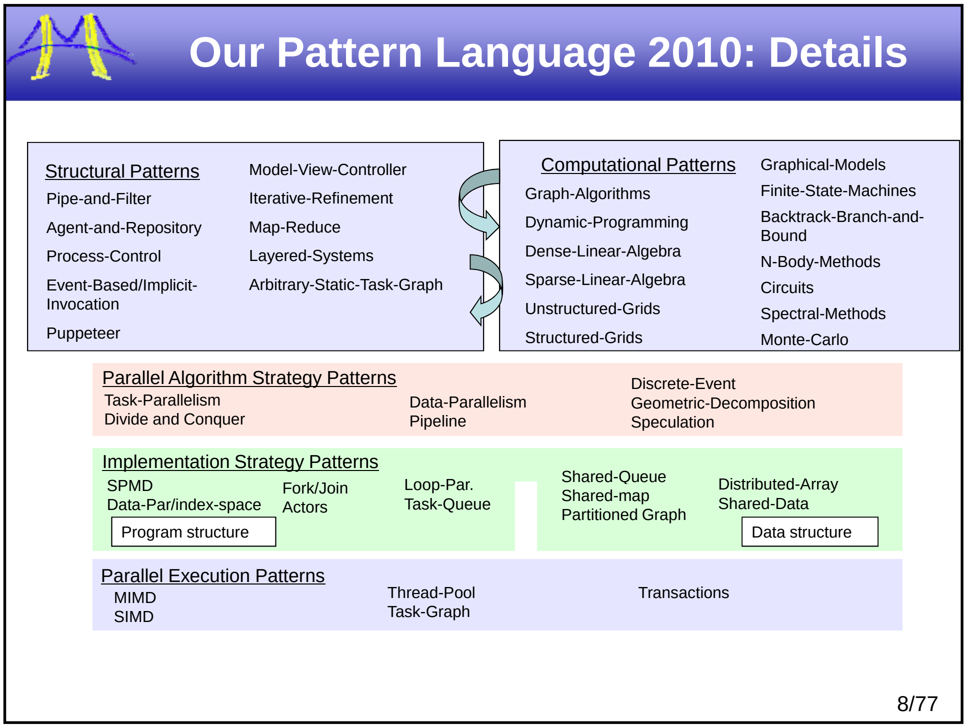
Application

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/Spare 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

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

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
- Fork/Join
- Actors
- Loop-Par.
- Task-Queue
- Shared-Queue
- Shared-map
- Partitioned Graph
- Distributed-Array
- Shared-Data

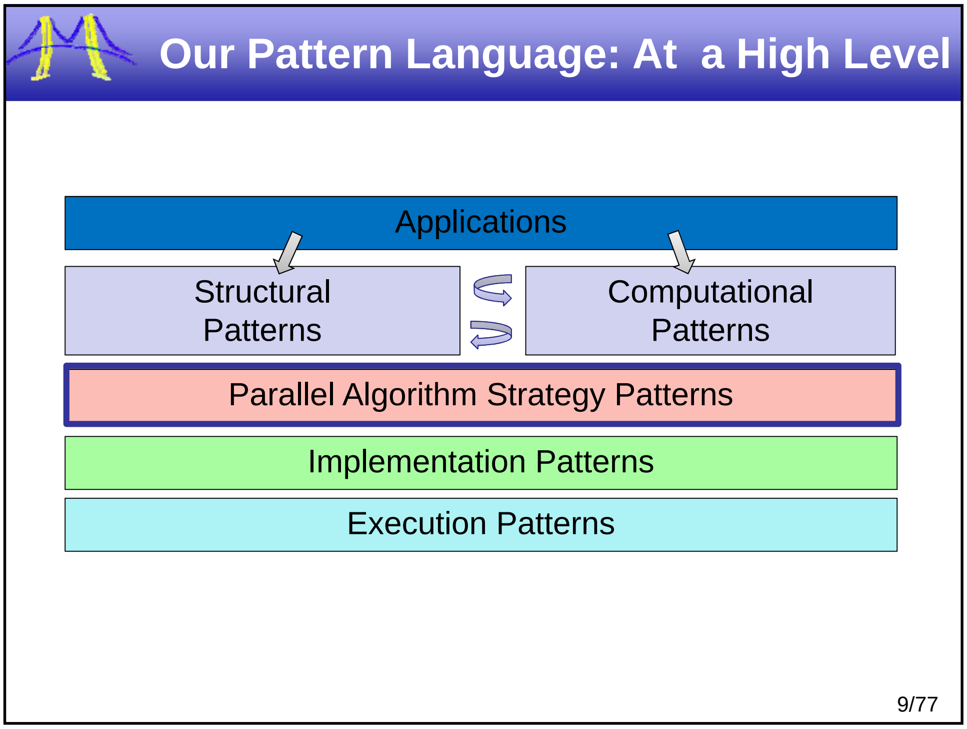
Program structure

Data structure

Parallel Execution Patterns

- MIMD
- SIMD
- Thread-Pool
- Task-Graph
- Transactions

8/77



Our Pattern Language: At a High Level

Applications

Structural Patterns

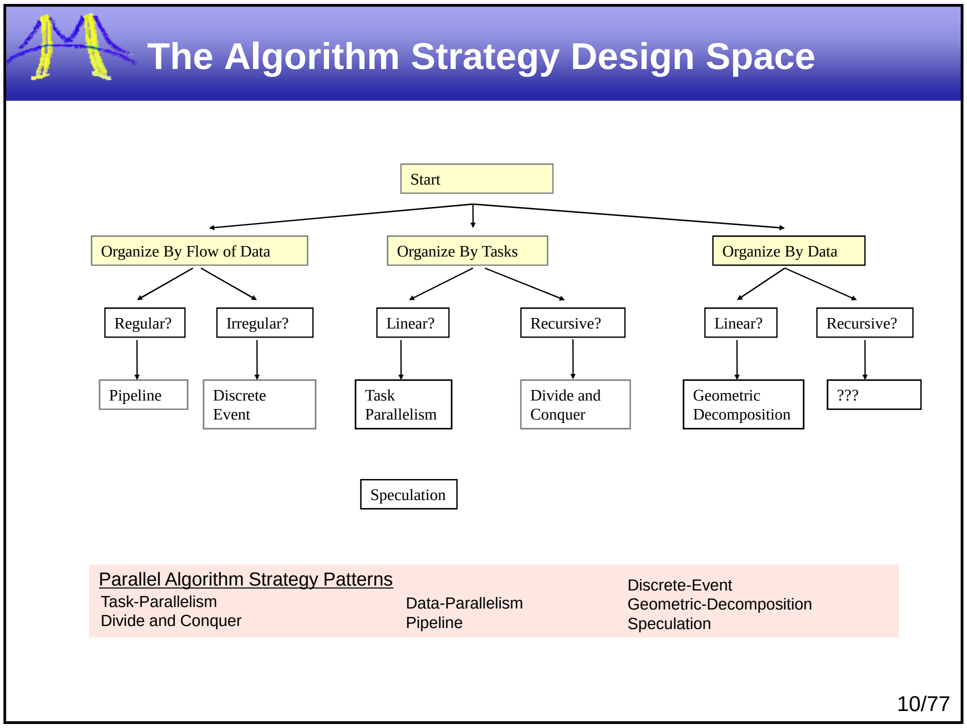
Computational Patterns

Parallel Algorithm Strategy Patterns

Implementation Patterns

Execution Patterns

9/77



The Algorithm Strategy Design Space

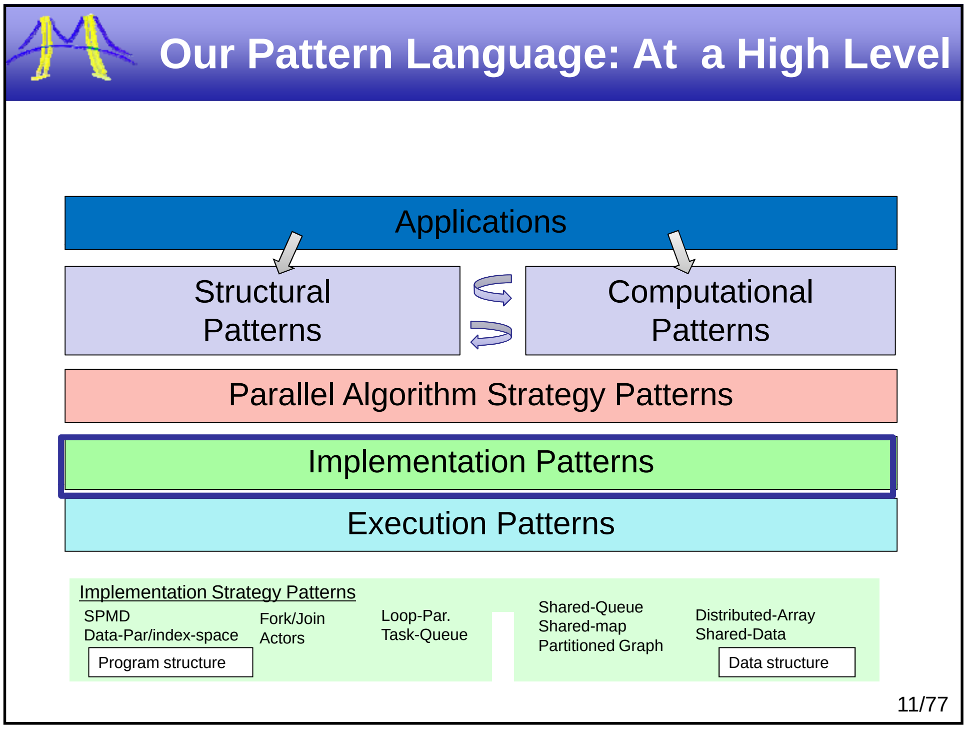
Parallel Algorithm Strategy Patterns

Task-Parallelism
Divide and Conquer

Data-Parallelism
Pipeline

Discrete-Event
Geometric-Decomposition
Speculation

10/77



Our Pattern Language: At a High Level

Applications

Structural Patterns

Computational Patterns

Parallel Algorithm Strategy Patterns

Implementation Patterns

Execution Patterns

Implementation Strategy Patterns

SPMD
Data-Par/index-space

Fork/Join
Actors

Loop-Par.
Task-Queue

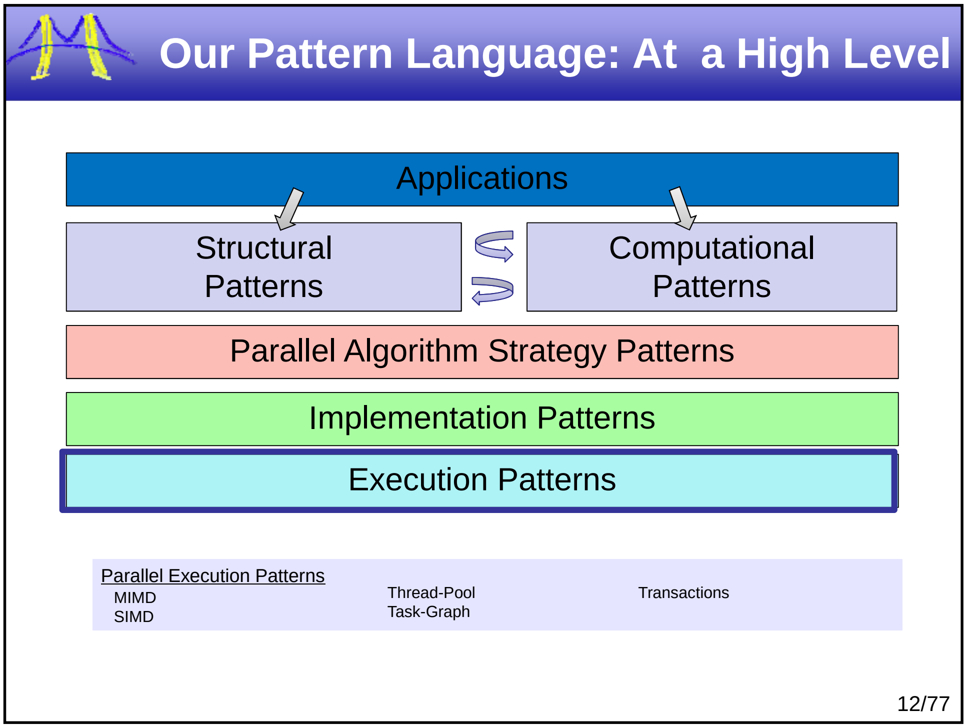
Shared-Queue
Shared-map
Partitioned Graph

Distributed-Array
Shared-Data

Program structure

Data structure

11/77



Our Pattern Language: At a High Level

Applications

Structural Patterns

Computational Patterns

Parallel Algorithm Strategy Patterns

Implementation Patterns

Execution Patterns

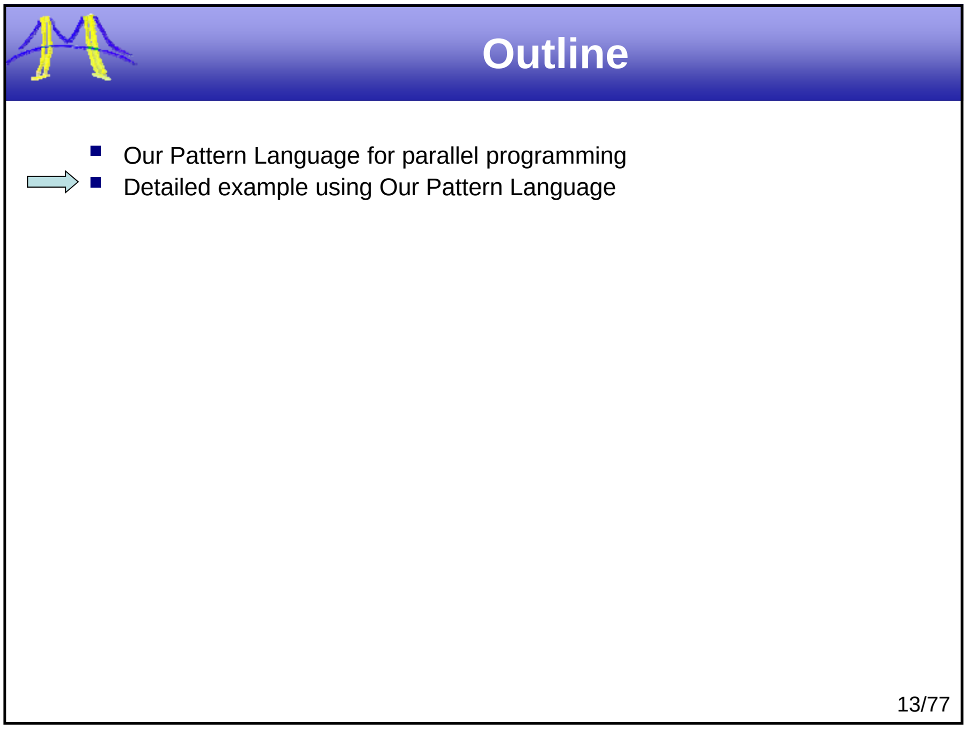
Parallel Execution Patterns

MIMD
SIMD

Thread-Pool
Task-Graph

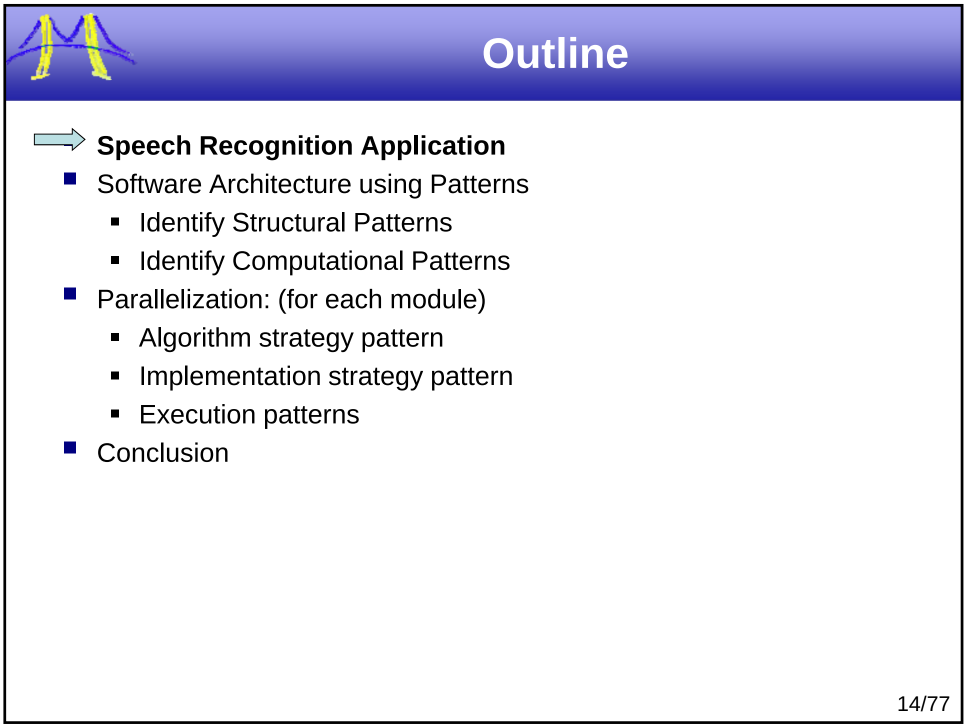
Transactions

12/77



Outline

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



Outline

⇨ **Speech Recognition Application**

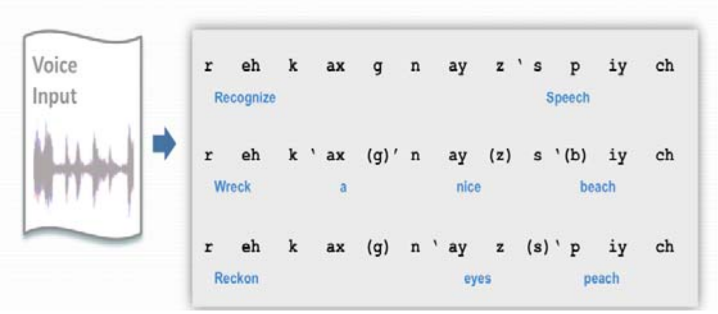
- Software Architecture using Patterns
  - Identify Structural Patterns
  - Identify Computational Patterns
- Parallelization: (for each module)
  - Algorithm strategy pattern
  - Implementation strategy pattern
  - Execution patterns
- Conclusion

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

15/77

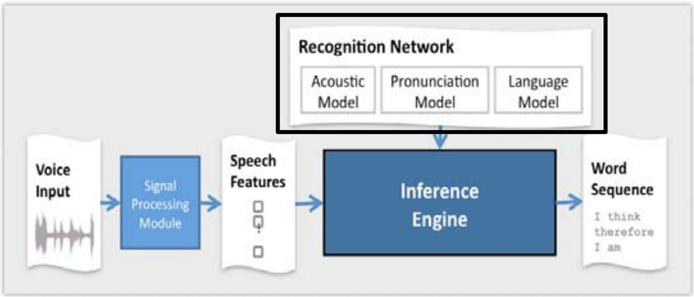
Continuous Speech Recognition


Voice
Input
r eh k ax g n ay z ' s p iy ch
Recognize
Speech
r eh k ' ax (g) ' n ay (z) s ' (b) iy ch
Wreck
a
nice
beach
r eh k ax (g) n ' ay z (s) ' p iy ch
Reckon
eyes
peach

- 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

16/77

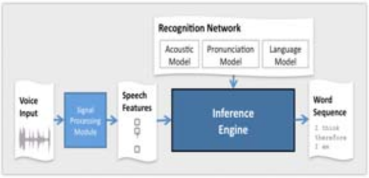
CSR System Structure

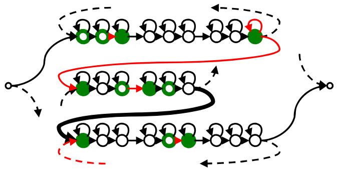


- Inference engine based system
  - Used in Sphinx (CMU, USA), HTK (Cambridge, UK), and Julius (CSRC, Japan)
- Modular and flexible setup
  - Shown to be effective for Arabic, English, Japanese, and Mandarin

Recognition Process

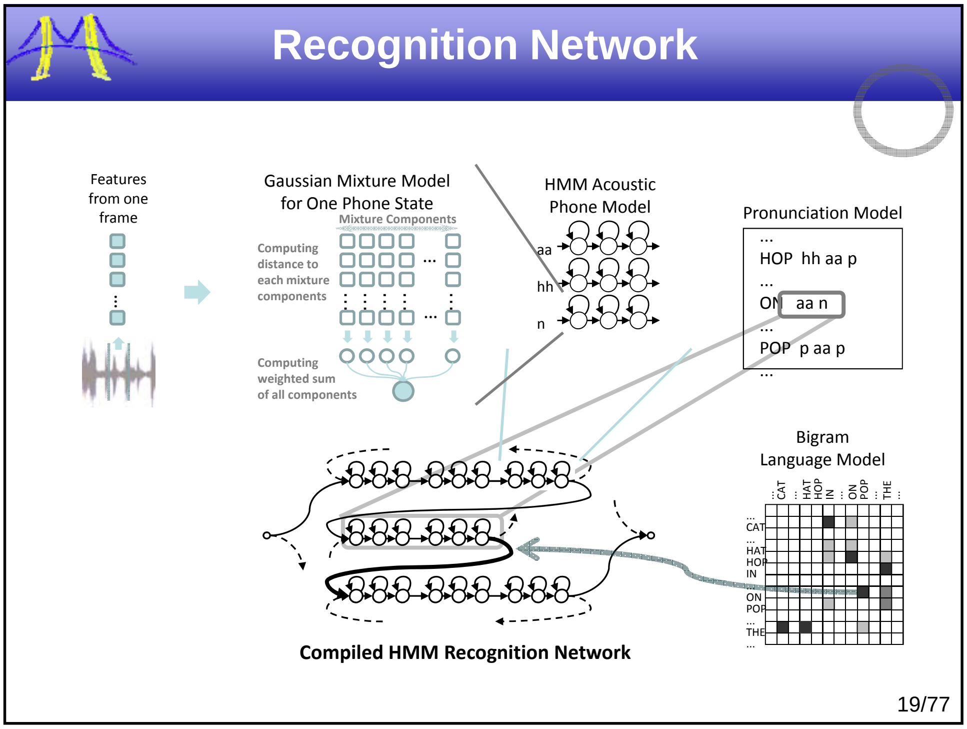
- Considers a sequence of features one at a time, for each feature:
  - Start with most-likely-sequences ending with previous feature
    - Corresponding to a set of *active states*
  - Consider the likelihood of current feature in its context
  - Compute most-likely-sequences ending with current feature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





Recognition Network

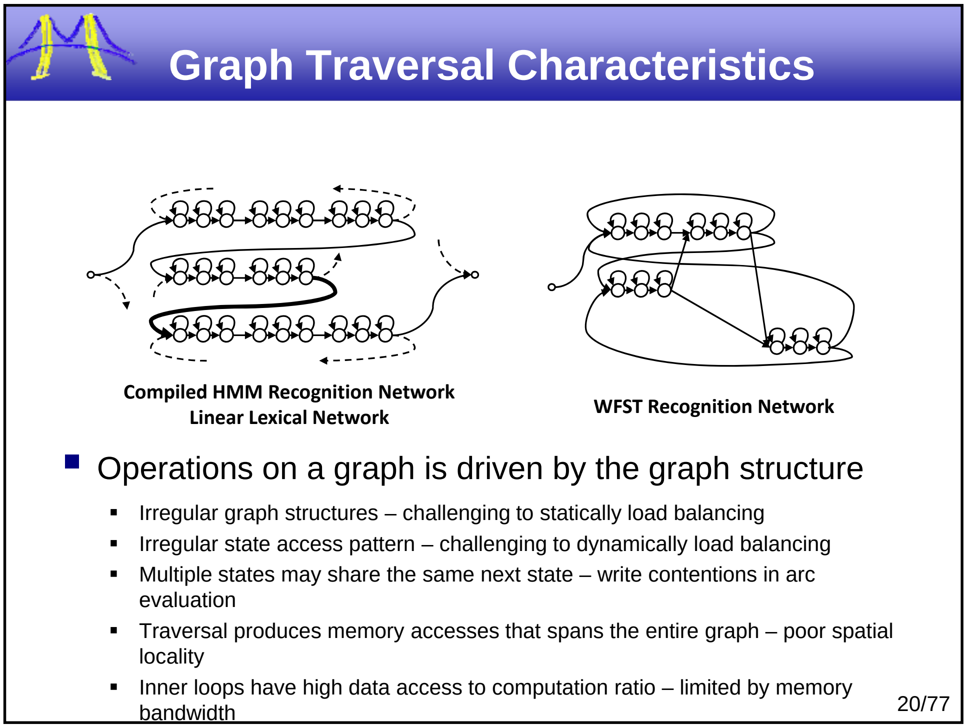
Recognition is a process of graph traversal



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
aa
hh
n
Pronunciation Model
...
HOP hh aa p
...
ON aa n
...
POP p aa p
...
Bigram Language Model
... CAT ... HAT HOP IN ... ON POP ... THE ...
... CAT ... HAT HOP IN ... ON POP ... THE ...
Compiled HMM Recognition Network

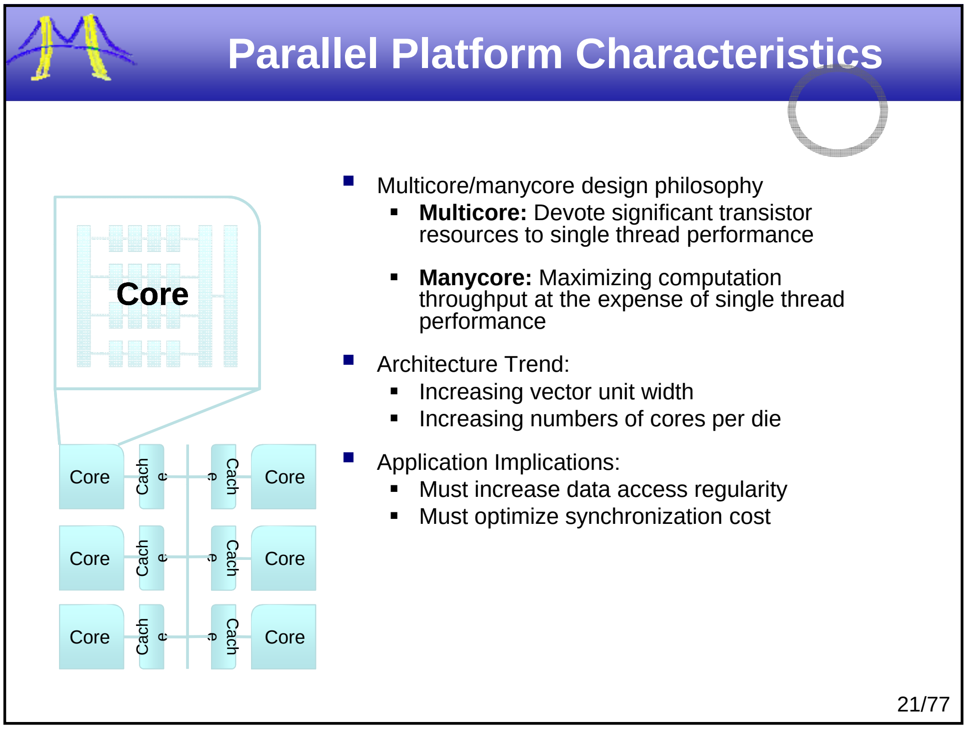
19/77



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

20/77

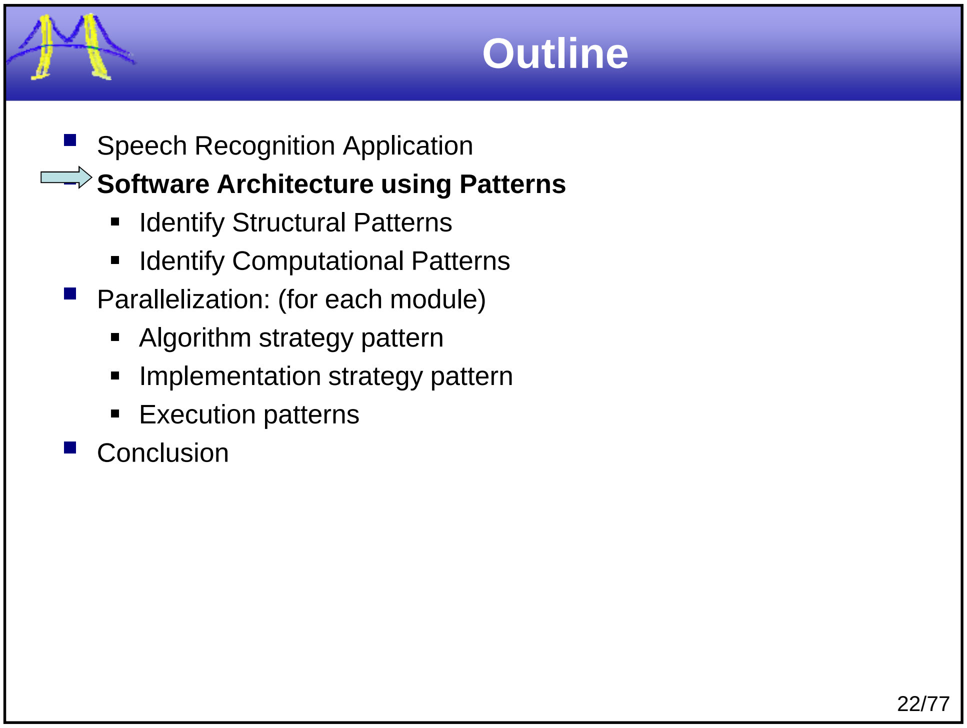


Parallel Platform Characteristics

Core
Core
Cach e
Cach e
Core
Core
Cach e
Cach e
Core
Core
Cach e
Cach e
Core

- 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

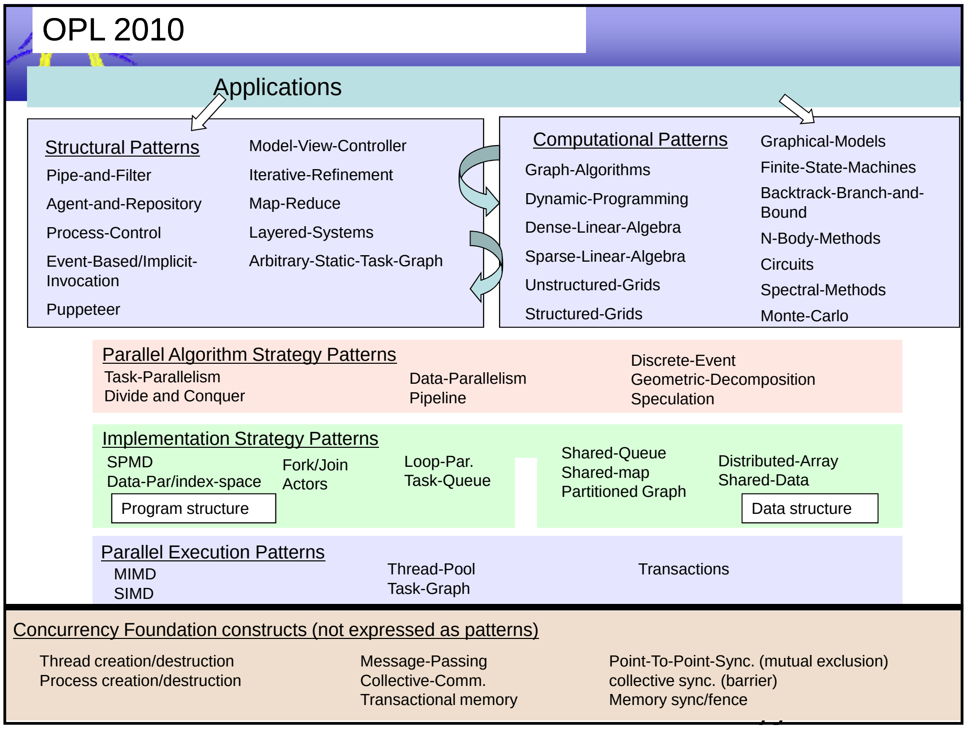
21/77



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

22/77



OPL 2010

Applications

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

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

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

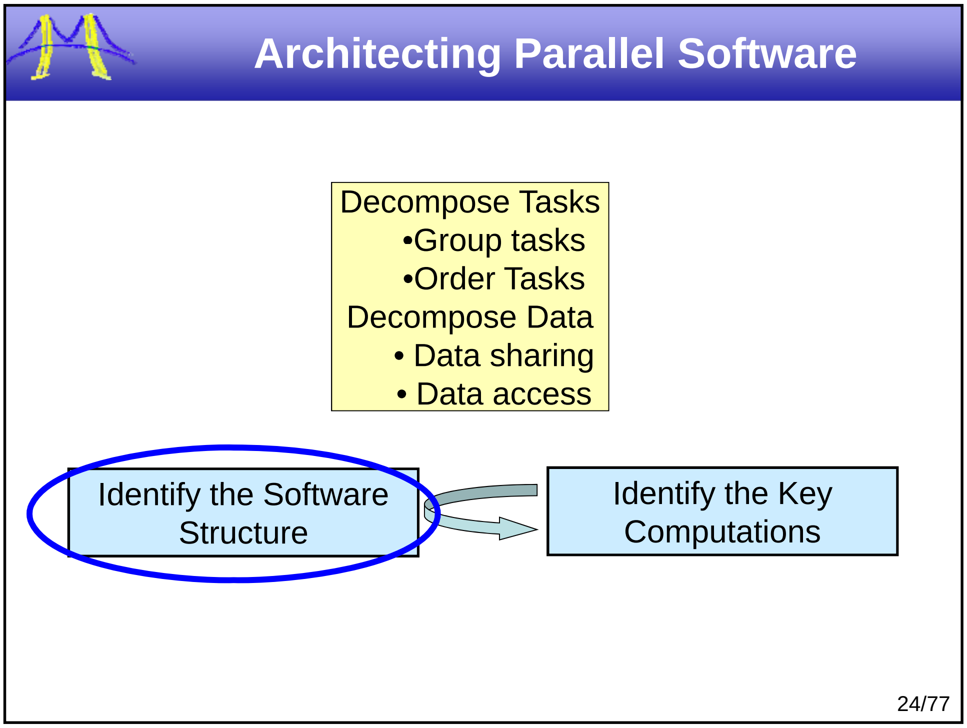
Implementation Strategy Patterns
- SPMD
- Data-Par/index-space
- Fork/Join
- Actors
- Loop-Par.
- Task-Queue
- Shared-Queue
- Shared-map
- Partitioned Graph
- Distributed-Array
- Shared-Data

Program structure

Data structure

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

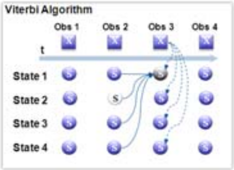
24/77

Inference Engine Architecture

- 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 a the 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?

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



Key computation: HMM Inference Algorithm

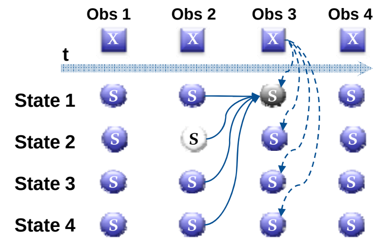
An instance of: **Graphical Models**

Implemented with: **Dynamic Programming**

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

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

Viterbi Algorithm



Legends:

S A State

X An Observation

$P(x_t|s_t)$

$m[t\text{-}1][s_{t\text{-}1}]$

$P(s_t|s_{t\text{-}1})$

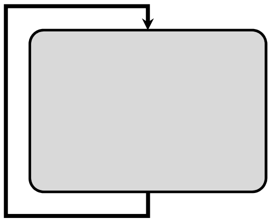
$m[t][s_t]$

Markov Condition:

$$m[t][s_t] \doteq \max_{s_0,\ldots,s_{t-1}} P(x_0,\ldots,x_t,s_0,\ldots,s_{t-1},s_t)$$

J. Chong, Y. Yi, A. Faria, N.R. Satish and K. Keutzer, "**Data-Parallel Large Vocabulary Continuous Speech Recognition on Graphics Processors**", Emerging Applications and Manycore Arch. 2008, pp. 23-35, June 2008

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

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

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?

Structural Patterns

Pipe-and-Filter

Agent-and-Repository

Process-Control

Event-Based/Implicit-Invocation

Puppeteer

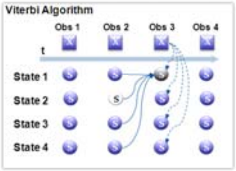
Model-View-Controller

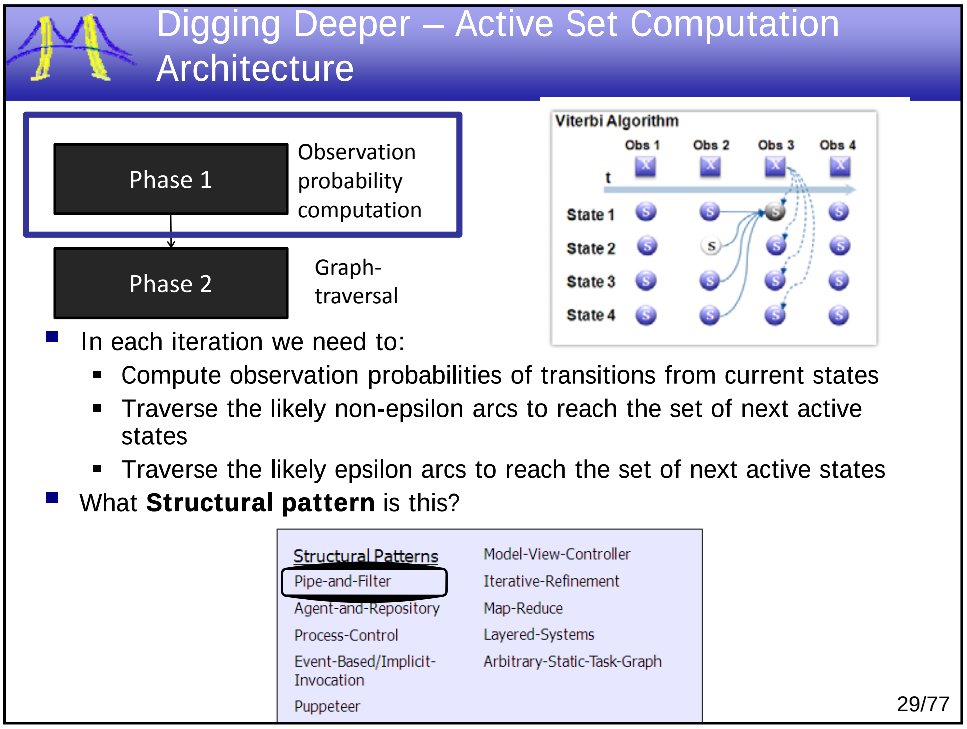
Iterative-Refinement

Map-Reduce

Layered-Systems

Arbitrary-Static-Task-Graph





Digging Deeper – Active Set Computation Architecture

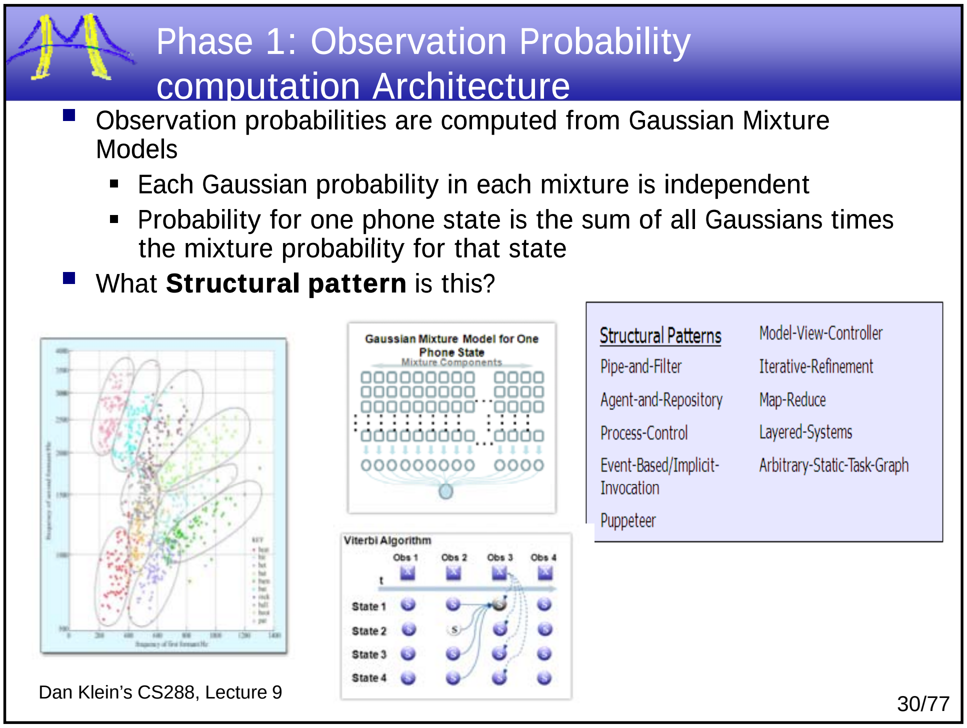
Phase 1 — Observation probability computation

Phase 2 — Graph-traversal

- 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?

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

29/77



Phase 1: Observation Probability computation Architecture

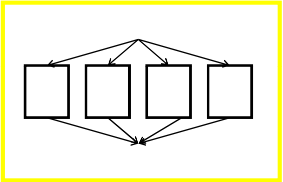
- Observation probabilities are computed from Gaussian Mixture Models
  - Each Gaussian probability in each mixture is independent
  - Probability for one phone state is the sum of all Gaussians times the mixture probability for that state
- What **Structural pattern** is this?

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

Dan Klein's CS288, Lecture 9

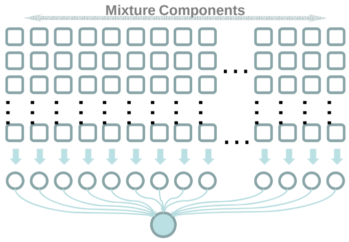
30/77

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



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

31/77

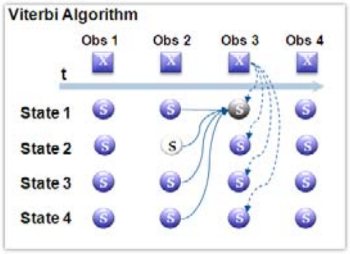
Phase 2: Graph Traversal

- 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?

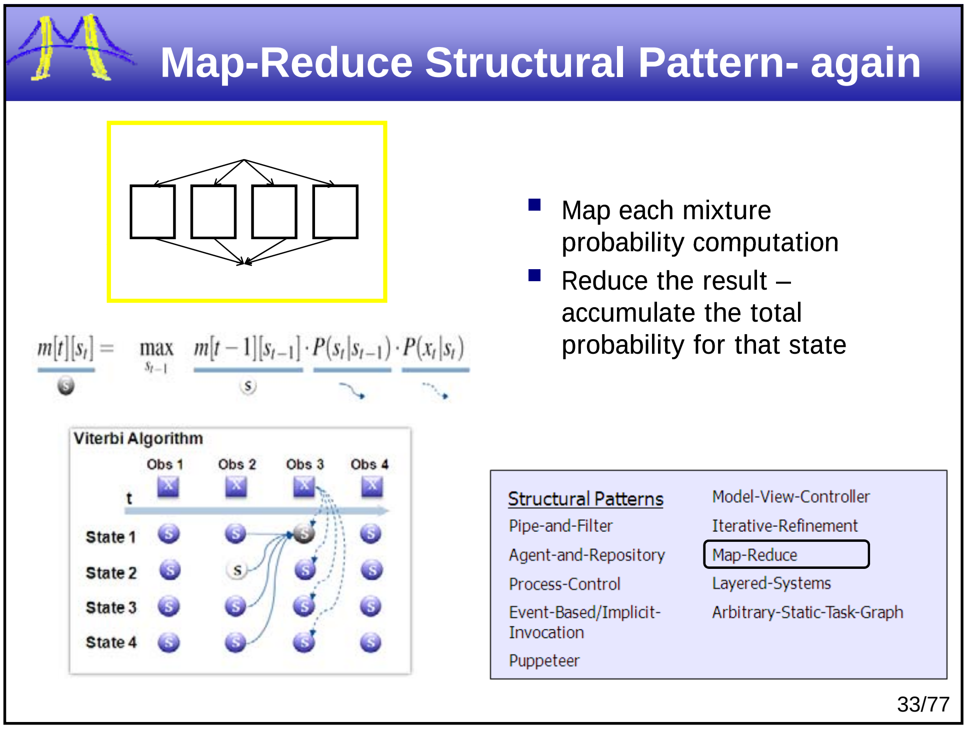
$$m[t][s_t] = \max_{s_{t-1}} m[t-1][s_{t-1}] \cdot P(s_t|s_{t-1}) \cdot P(x_t|s_t)$$

Structural Patterns

- 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



32/77



Map-Reduce Structural Pattern- again

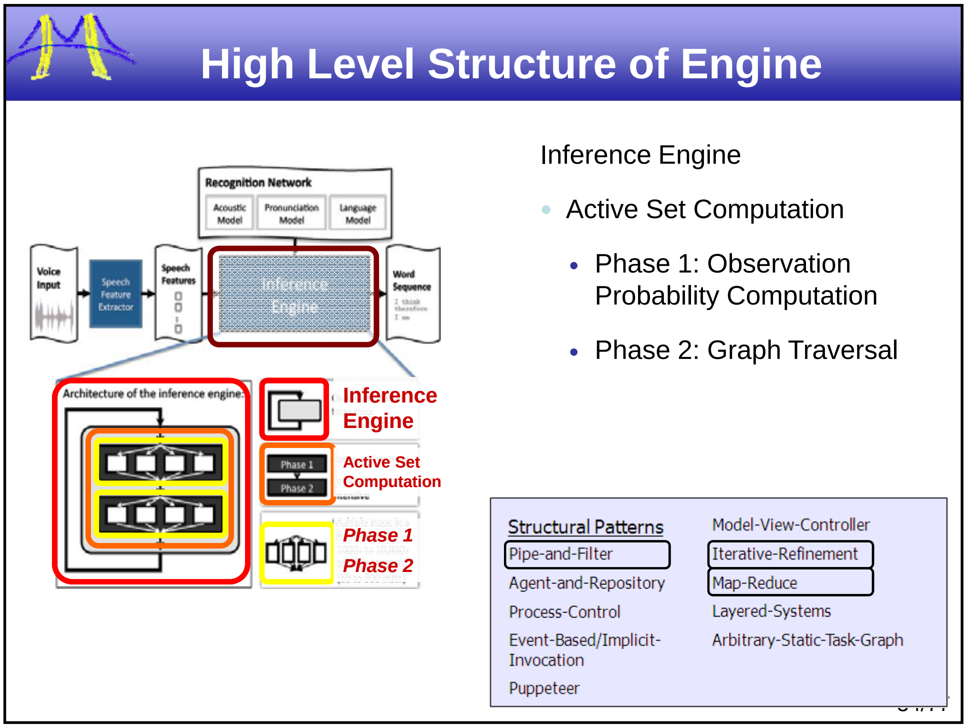
- Map each mixture probability computation
- Reduce the result – accumulate the total probability for that state

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

Structural Patterns
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

33/77



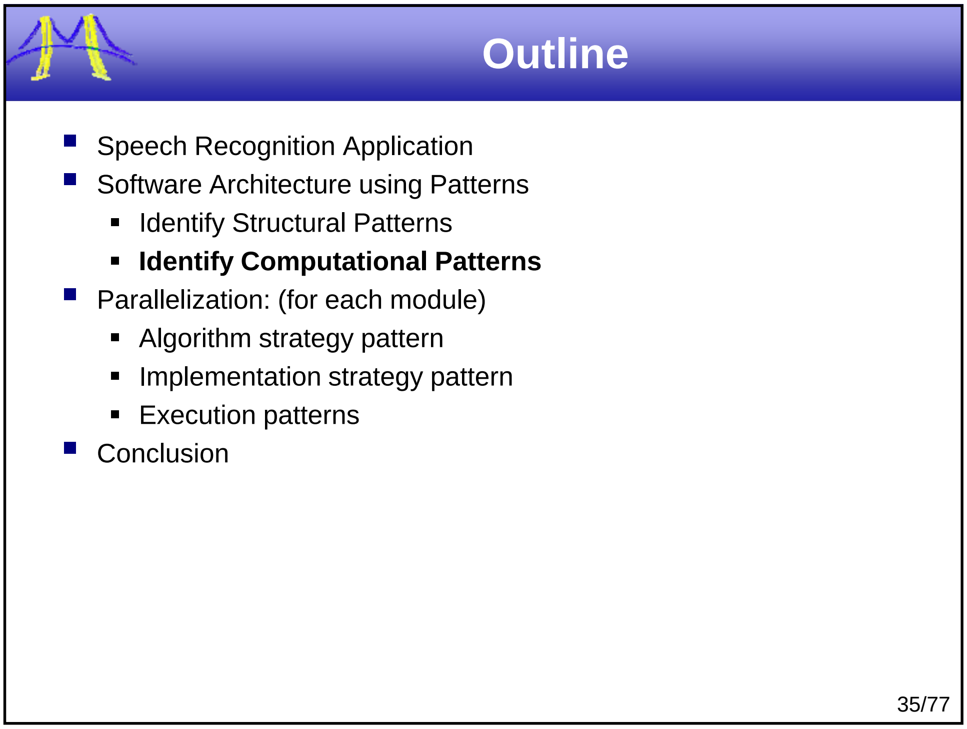
High Level Structure of Engine

Inference Engine

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

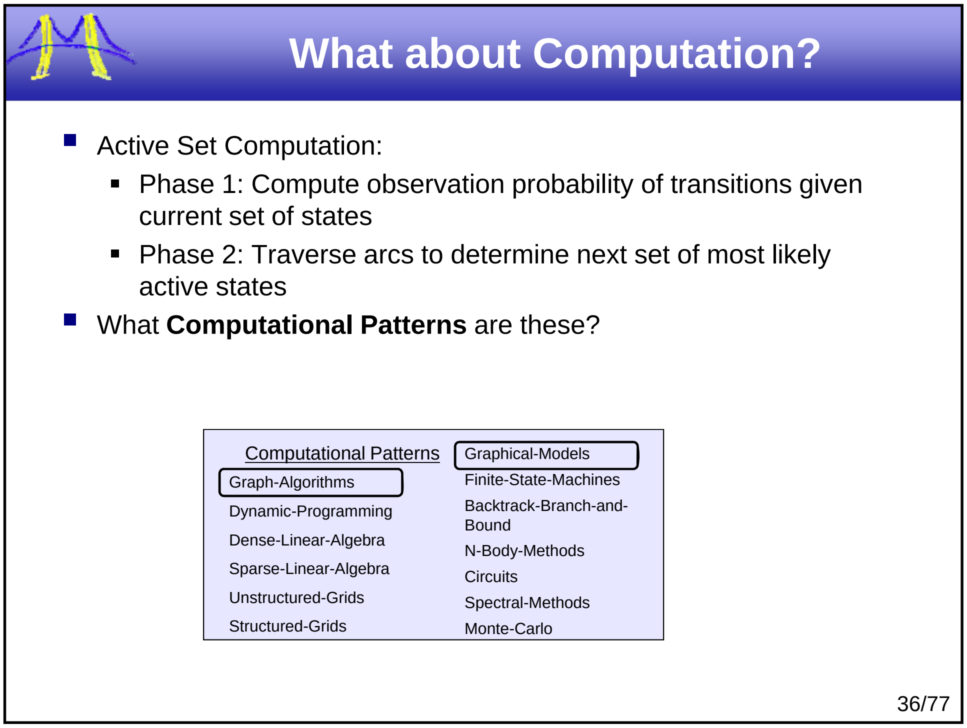
Structural Patterns
Pipe-and-Filter
Agent-and-Repository
Process-Control
Event-Based/Implicit-Invocation
Puppeteer

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



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?

Computational Patterns

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

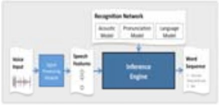
Graphical-Models
Finite-State-Machines
Backtrack-Branch-and-Bound
N-Body-Methods
Circuits
Spectral-Methods
Monte-Carlo

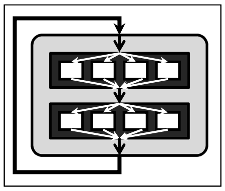
Outline

- 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?



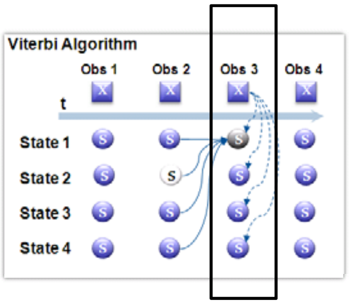


<u>Parallel Algorithm Strategy Patterns</u>

Task-Parallelism Divide and Conquer	Data-Parallelism Pipeline	Discrete-Event Geometric-Decomposition Speculation

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 there-of)



Parallel Algorithm Strategy Patterns

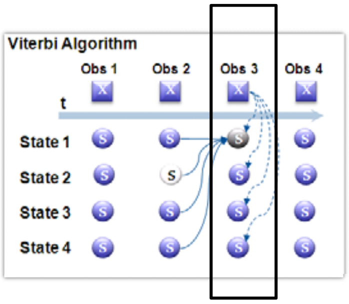
Task-Parallelism
Divide and Conquer

Data-Parallelism
Pipeline

Discrete-Event
Geometric-Decomposition
Speculation

Now to Parallelism – Inference Engine

- Phase 1: Observation Probability Computation
  - Structural: MapReduce
  - Computational: Graphical Models
- Compute cluster Gaussian Mixture Probabilities for each transition label
- Map reduce necessarily implies data-parallelism

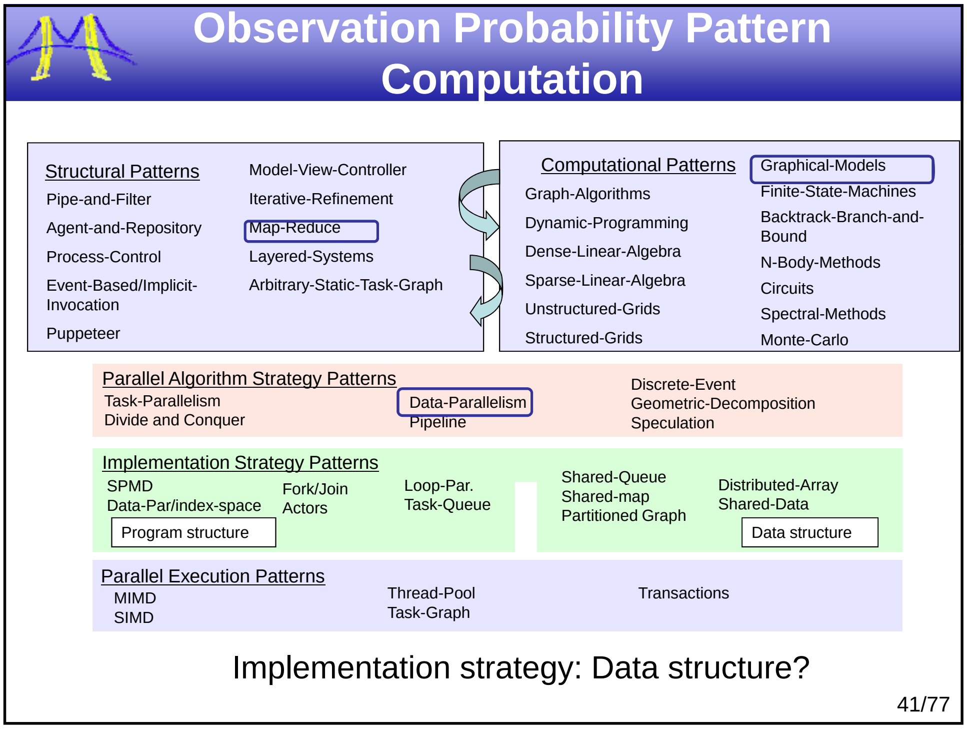


Parallel Algorithm Strategy Patterns

Task-Parallelism
Divide and Conquer

Data-Parallelism
Pipeline

Discrete-Event
Geometric-Decomposition
Speculation



Observation Probability Pattern Computation

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

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

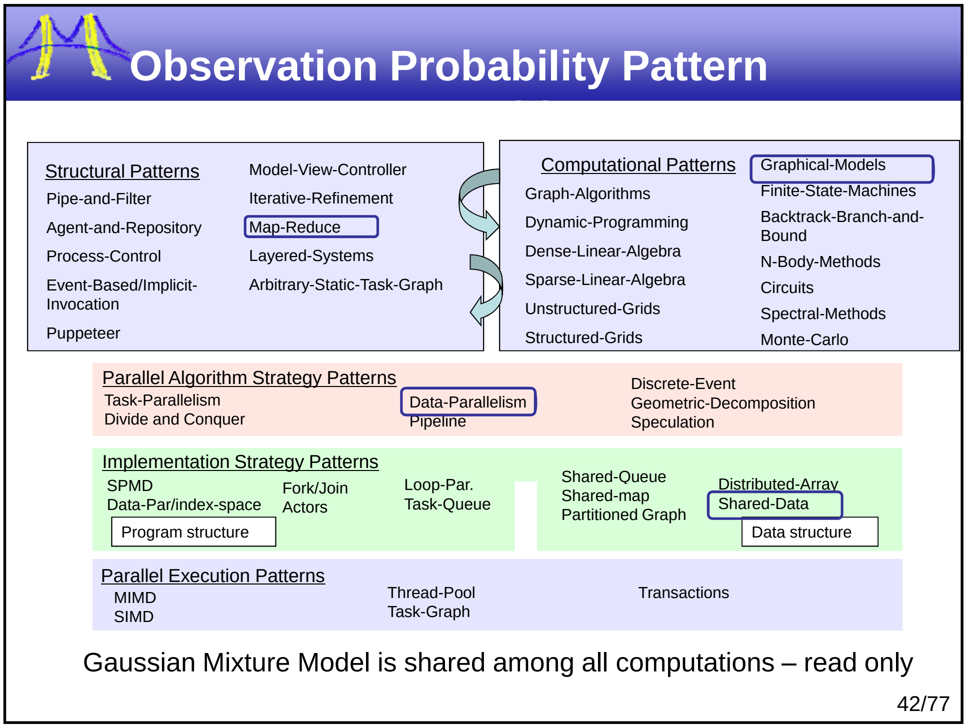
Parallel Algorithm Strategy Patterns
Task-Parallelism
Divide and Conquer
Data-Parallelism
Pipeline
Discrete-Event
Geometric-Decomposition
Speculation

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

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

Implementation strategy: Data structure?

41/77



Observation Probability Pattern

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

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

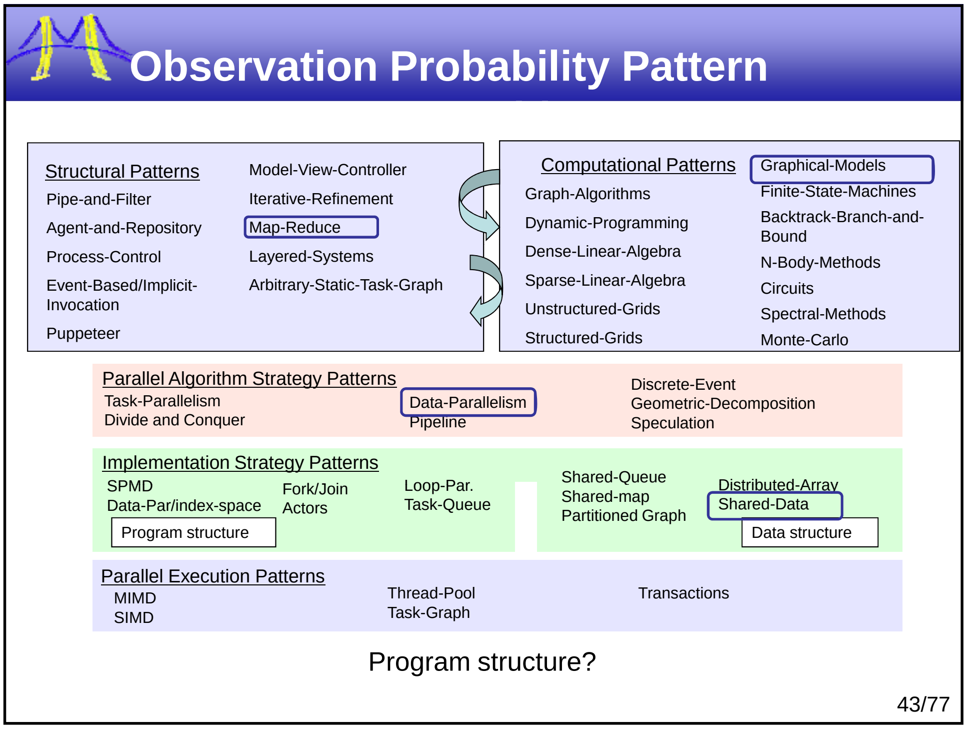
Parallel Algorithm Strategy Patterns
Task-Parallelism
Divide and Conquer
Data-Parallelism
Pipeline
Discrete-Event
Geometric-Decomposition
Speculation

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

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

Gaussian Mixture Model is shared among all computations – read only

42/77



Observation Probability Pattern

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

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

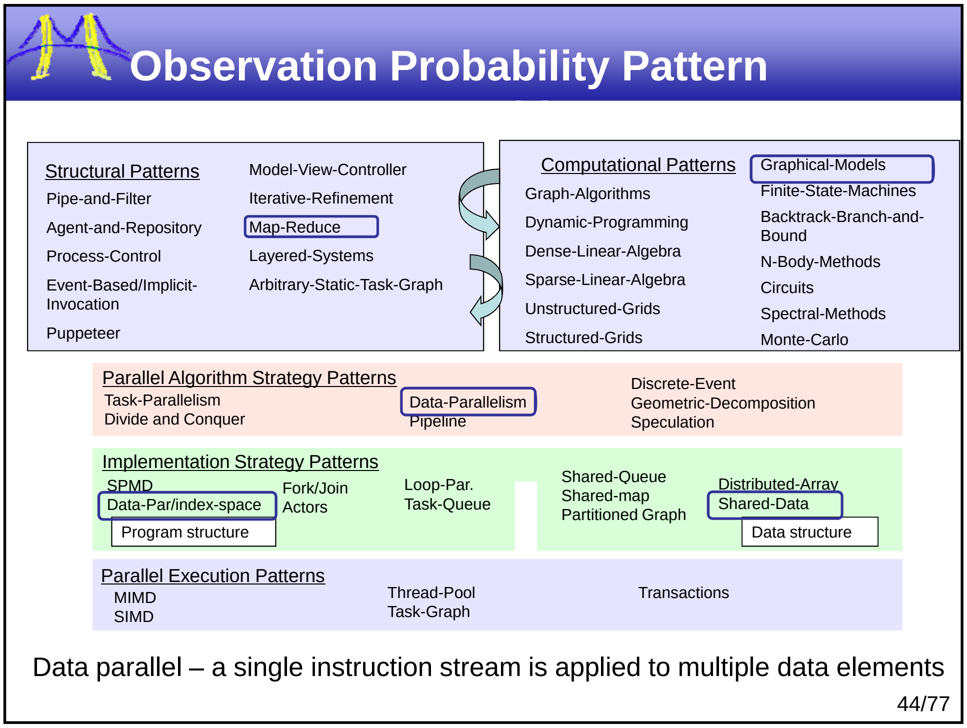
Parallel Algorithm Strategy Patterns
Task-Parallelism
Divide and Conquer
Data-Parallelism
Pipeline
Discrete-Event
Geometric-Decomposition
Speculation

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

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

Program structure?

43/77



Observation Probability Pattern

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

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

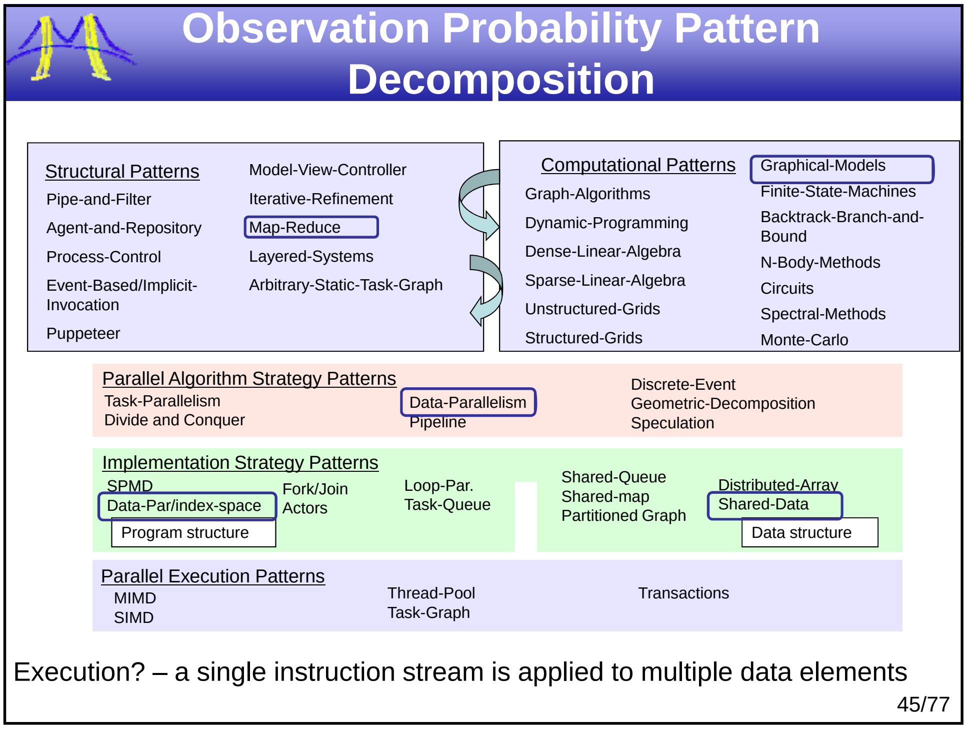
Parallel Algorithm Strategy Patterns
Task-Parallelism
Divide and Conquer
Data-Parallelism
Pipeline
Discrete-Event
Geometric-Decomposition
Speculation

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

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

Data parallel – a single instruction stream is applied to multiple data elements

44/77



Observation Probability Pattern Decomposition

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

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

Parallel Algorithm Strategy Patterns

Task-Parallelism
Divide and Conquer
Data-Parallelism
Pipeline
Discrete-Event
Geometric-Decomposition
Speculation

Implementation Strategy Patterns

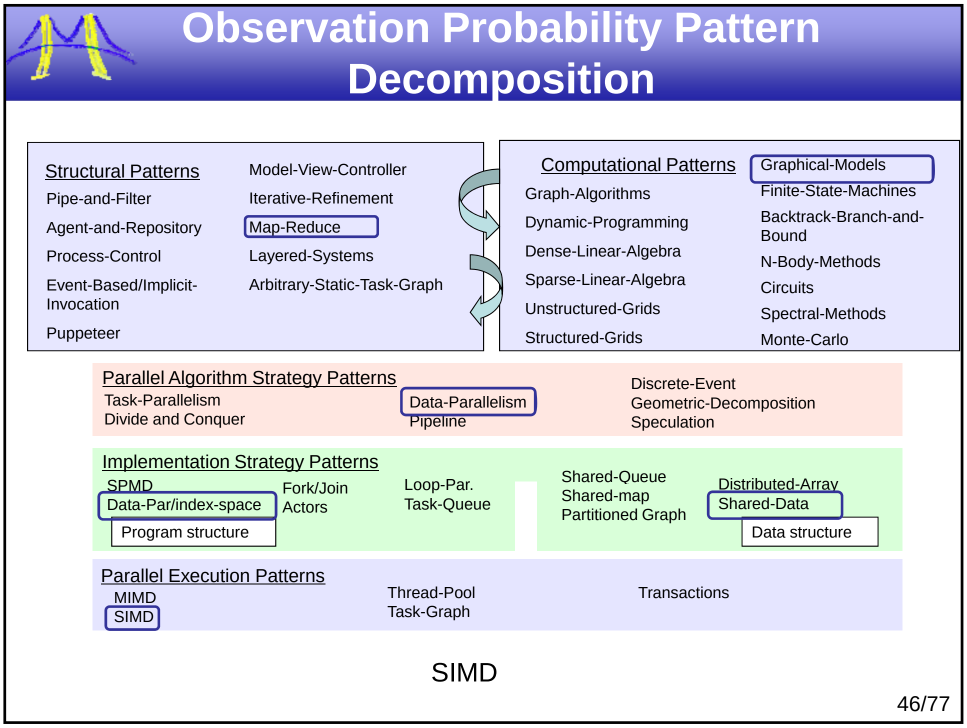
SPMD
Data-Par/index-space
Fork/Join
Actors
Loop-Par.
Task-Queue
Program structure
Shared-Queue
Shared-map
Partitioned Graph
Distributed-Array
Shared-Data
Data structure

Parallel Execution Patterns

MIMD
SIMD
Thread-Pool
Task-Graph
Transactions

Execution? – a single instruction stream is applied to multiple data elements

45/77



Observation Probability Pattern Decomposition

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

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

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
Fork/Join
Actors
Loop-Par.
Task-Queue
Program structure
Shared-Queue
Shared-map
Partitioned Graph
Distributed-Array
Shared-Data
Data structure

Parallel Execution Patterns

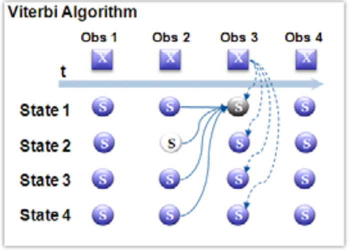
MIMD
SIMD
Thread-Pool
Task-Graph
Transactions

SIMD

46/77

Now to Parallelism Pt2 – Inference Engine

- 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?



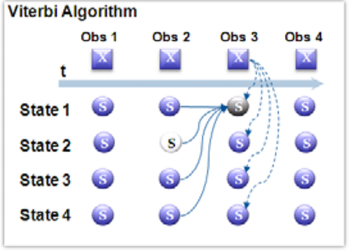
- Hint:
  - What are the operands?
  - What are the dependences?

Parallel Algorithm Strategy Patterns

Task-Parallelism Divide and Conquer	Data-Parallelism Pipeline	Discrete-Event Geometric-Decomposition Speculation

Now to Parallelism – Inference Engine

- Phase 2: Graph Traversal
  - Structural: MapReduce
  - Computational: 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

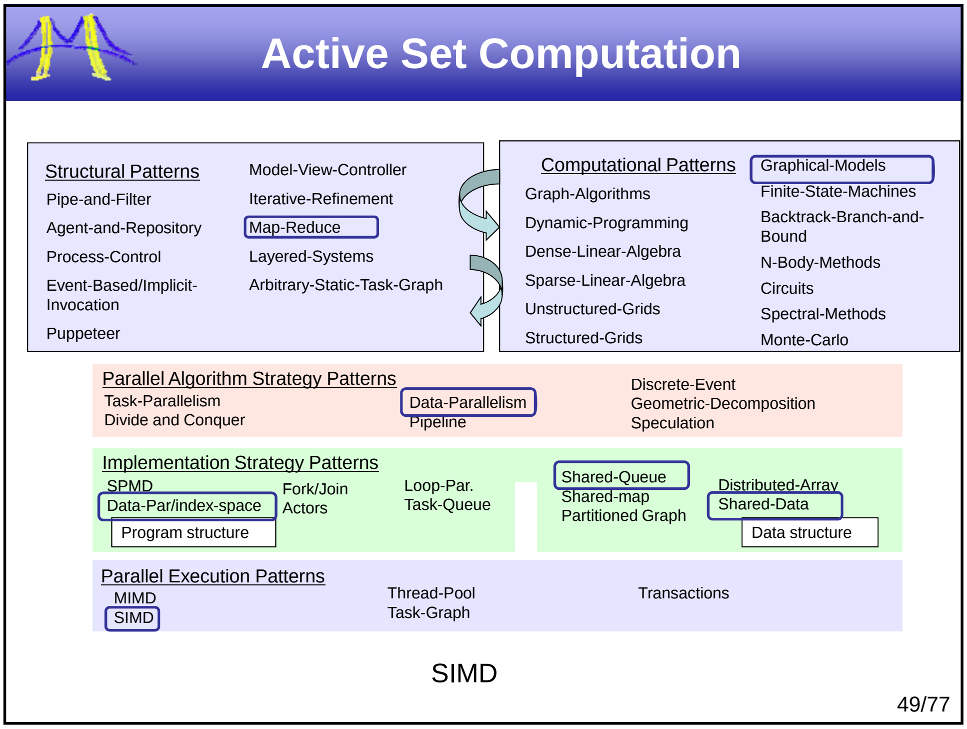


Data Parallelism!

Each next state computation can be computed independently.

Parallel Algorithm Strategy Patterns

Task-Parallelism Divide and Conquer	Data-Parallelism Pipeline	Discrete-Event Geometric-Decomposition Speculation



Active Set Computation

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

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

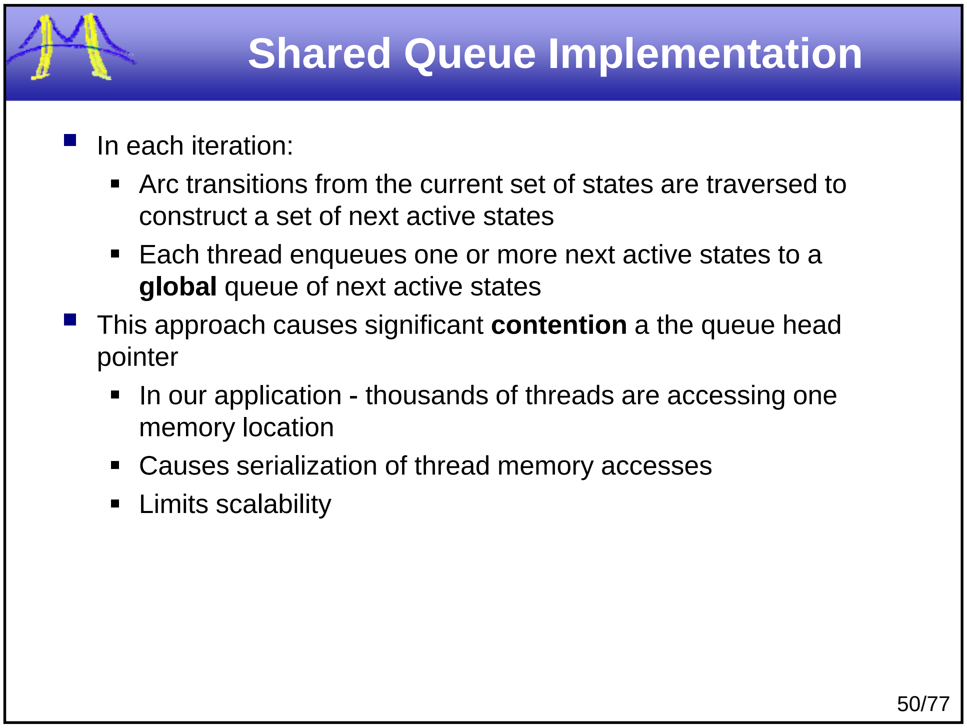
Parallel Algorithm Strategy Patterns
Task-Parallelism
Divide and Conquer
Data-Parallelism
Pipeline
Discrete-Event
Geometric-Decomposition
Speculation

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

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

SIMD

49/77



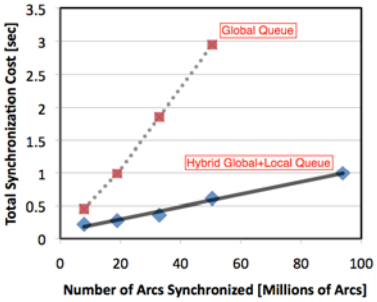
Shared Queue Implementation

- 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** a the queue head pointer
  - In our application - thousands of threads are accessing one memory location
  - Causes serialization of thread memory accesses
  - Limits scalability

50/77

Shared Queue Implementation

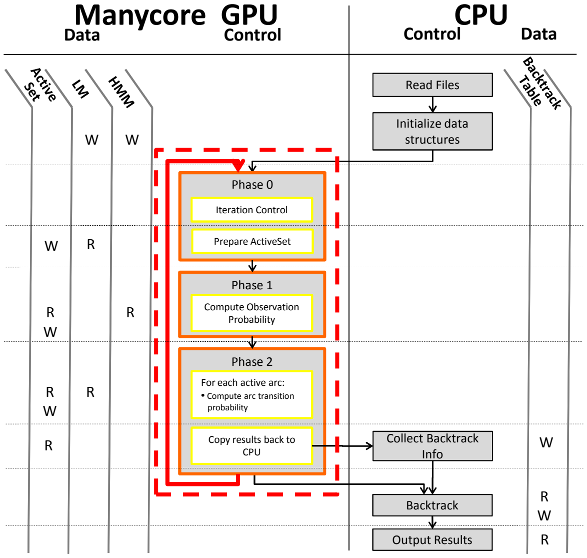
- Solution to the queue head pointer contention -> distributed queue
  1. Each thread in a thread block writes to a local queue of next active states
  2. Local queues are merged into a global queue



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

Speech Reference Implementation

Jike Chong, Ekaterina Gonina, Youngmin Yi, Kurt Keutzer, “**A Fully Data Parallel WFST-based Large Vocabulary Continuous Speech Recognition on a Graphics Processing Unit**”, Proceeding of the 10th Annual Conference of the International Speech Communication Association (InterSpeech), page 1183 – 1186, September, 2009.



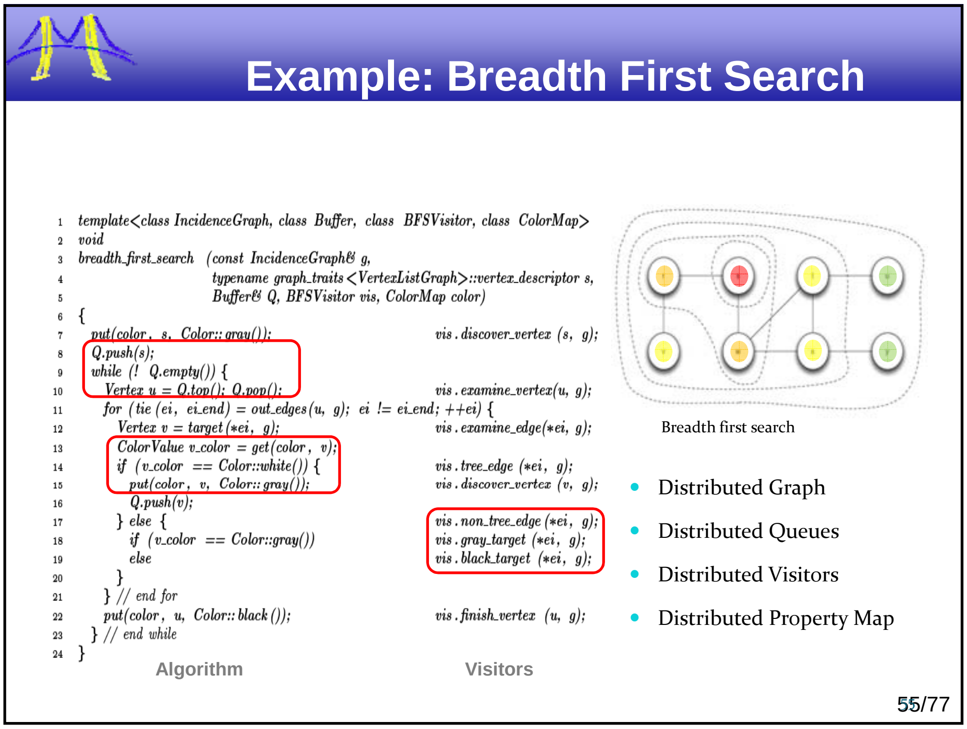
Kisun You, Jike Chong, Youngmin Yi, Ekaterina Gonina, Christopher Hughes, Yen-Kuang Chen, Wonyong Sung, Kurt Keutzer, “**Parallel Scalability in Speech Recognition: Inference engine in large vocabulary continuous speech recognition**”, IEEE Signal Processing Magazine, vol. 26, no. 6, pp. 124-135, November 2009.

Jike Chong, Ekaterina Gonina, Kisun You, Kurt Keutzer, “**Scalable Parallelization of Automatic Speech Recognition**”, Invited book chapter in Scaling Up Machine Learning, an upcoming 2010 Cambridge University Press book.

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

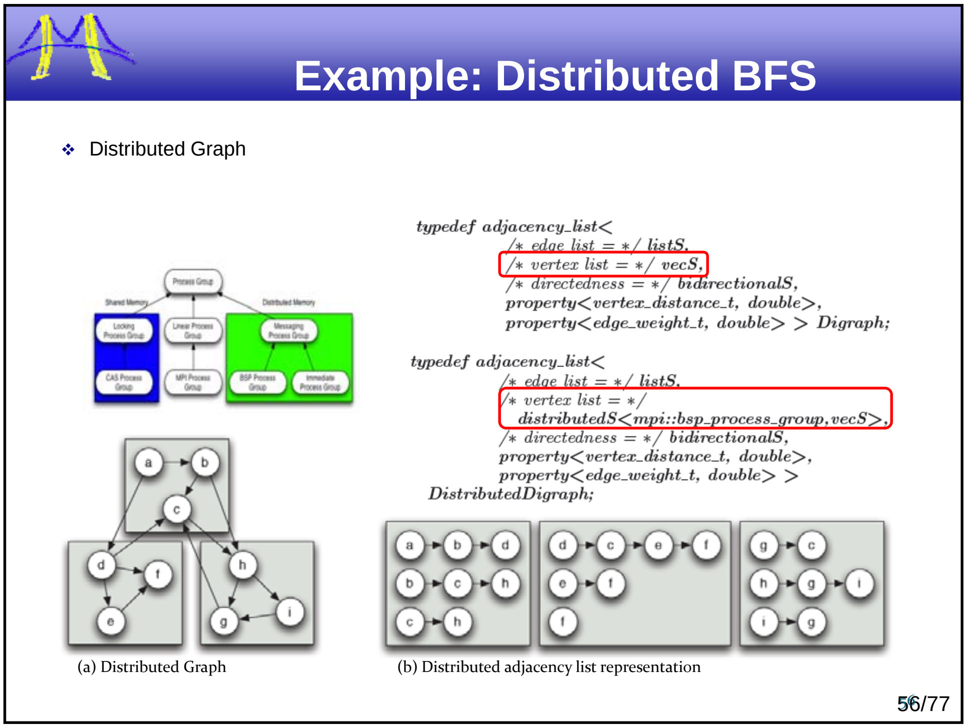
```
template<class IncidenceGraph, class Buffer, class BFSVisitor, class ColorMap>
void
breadth_first_search (const IncidenceGraph& g,
                      typename graph_traits<VertexListGraph>::vertex_descriptor s,
                      Buffer& Q, BFSVisitor vis, ColorMap color)
{
  put(color, s, Color::gray());                             vis.discover_vertex (s, g);
  Q.push(s);
  while (! Q.empty()) {
    Vertex u = Q.top(); Q.pop();                            vis.examine_vertex(u, g);
    for (tie(ei, ei_end) = out_edges(u, g); ei != ei_end; ++ei) {
      Vertex v = target(*ei, g);                            vis.examine_edge(*ei, g);
      ColorValue v_color = get(color, v);
      if (v_color == Color::white()) {                      vis.tree_edge (*ei, g);
        put(color, v, Color::gray());                       vis.discover_vertex (v, g);
        Q.push(v);
      } else {                                              vis.non_tree_edge(*ei, g);
        if (v_color == Color::gray())                       vis.gray_target (*ei, g);
        else                                                vis.black_target (*ei, g);
      }
    } // end for
    put(color, u, Color::black());                          vis.finish_vertex (u, g);
  } // end while
}
```

Algorithm | Visitors

Breadth first search

- Distributed Graph
- Distributed Queues
- Distributed Visitors
- Distributed Property Map

55/77



Example: Distributed BFS

- Distributed Graph

Process Group
Shared Memory
Distributed Memory
Locking Process Group
Linear Process Group
Messaging Process Group
CAS Process Group
MPI Process Group
BSP Process Group
Immediate Process Group
a
b
c
d
e
f
g
h
i

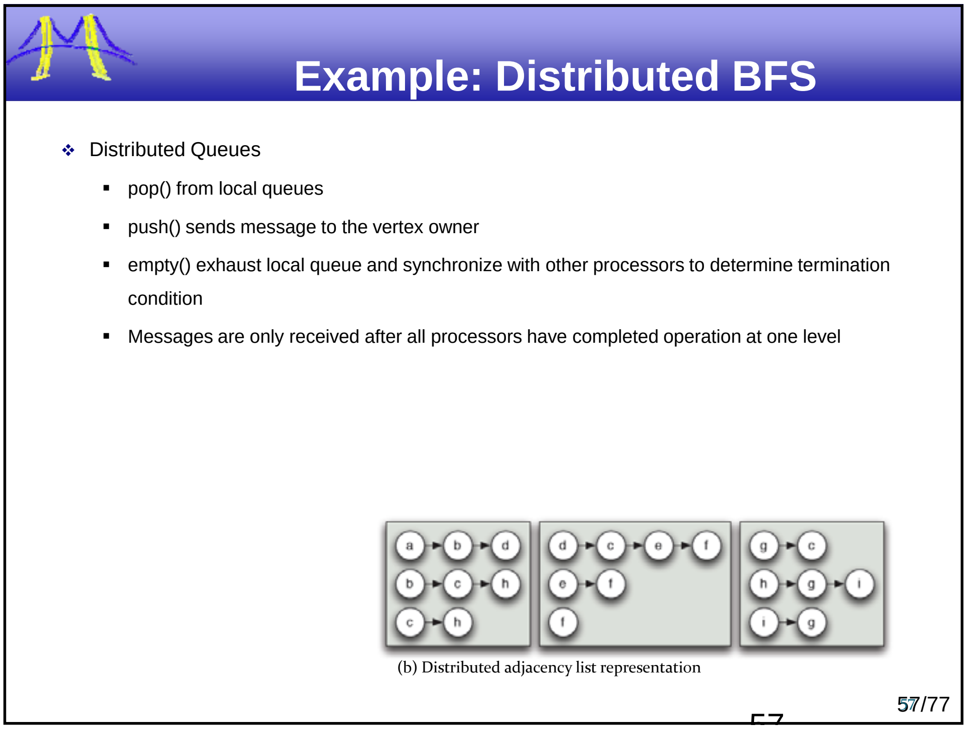
```
typedef adjacency_list<
        /* edge list = */ listS,
        /* vertex list = */ vecS,
        /* directedness = */ bidirectionalS,
        property<vertex_distance_t, double>,
        property<edge_weight_t, double> > Digraph;

typedef adjacency_list<
        /* edge list = */ listS,
        /* vertex list = */
          distributedS<mpi::bsp_process_group,vecS>,
        /* directedness = */ bidirectionalS,
        property<vertex_distance_t, double>,
        property<edge_weight_t, double> >
  DistributedDigraph;
```

(a) Distributed Graph

(b) Distributed adjacency list representation

56/77



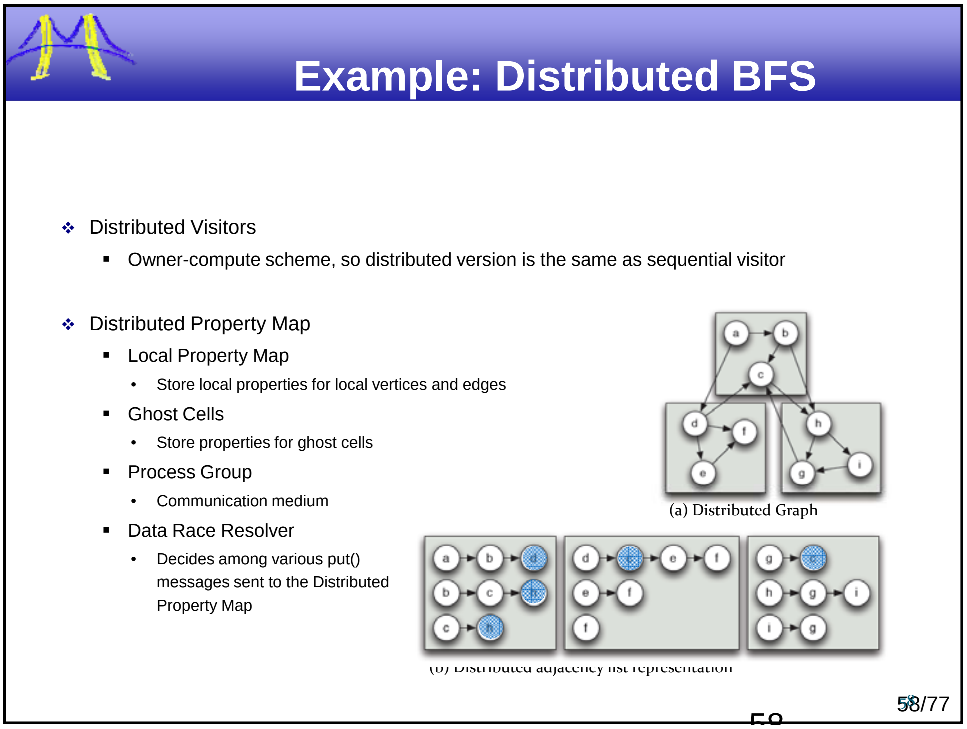
Example: Distributed BFS

- 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

a b d
b c h
c h
d c e f
e f
f
g c
h g i
i g

(b) Distributed adjacency list representation

57/77



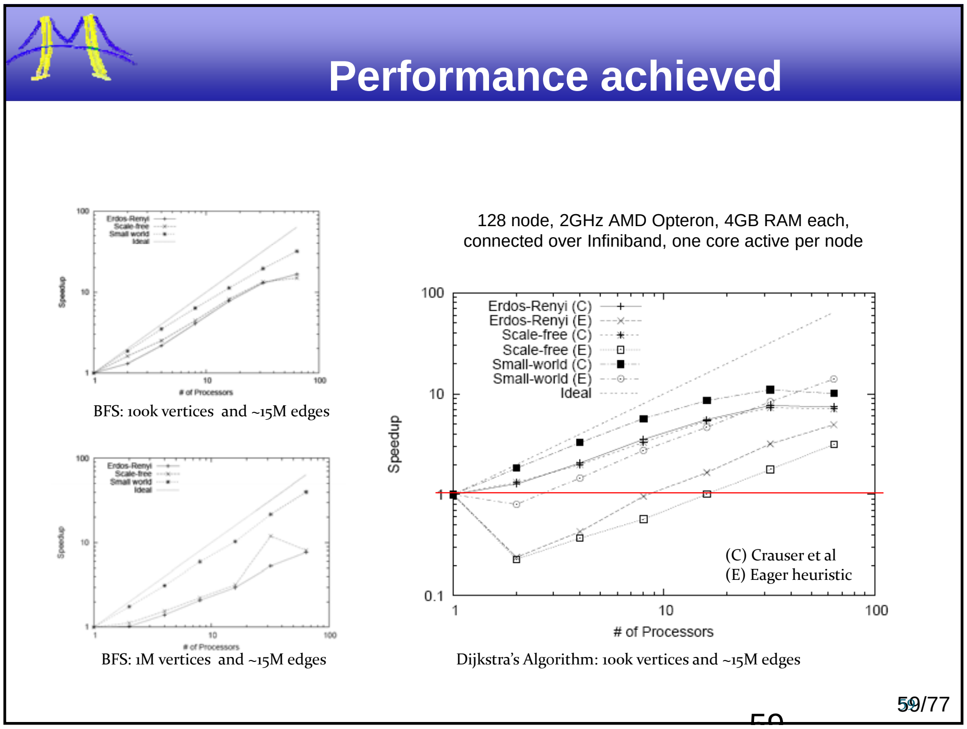
Example: Distributed BFS

- 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

(b) Distributed adjacency list representation

58/77



Performance achieved

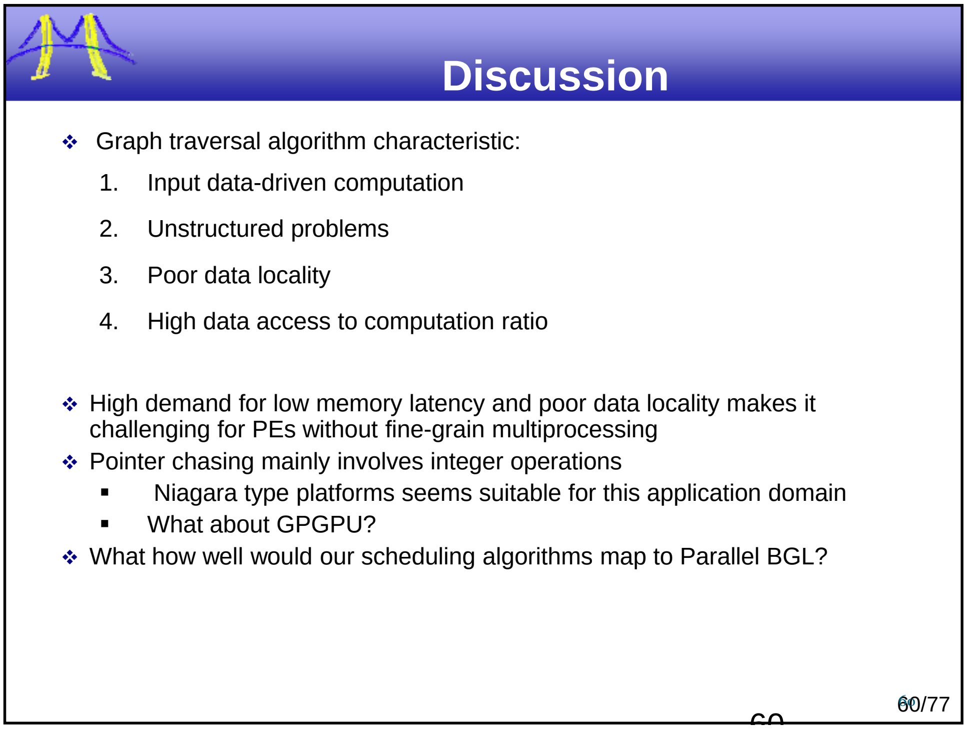
BFS: 100k vertices and ~15M edges

BFS: 1M vertices and ~15M edges

128 node, 2GHz AMD Opteron, 4GB RAM each, connected over Infiniband, one core active per node

Dijkstra's Algorithm: 100k vertices and ~15M edges

59/77



Discussion

- ❖ 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?

60/77