

SEJITS: embedded specializers to turn patterns-based designs into optimized parallel code

Tim Mattson

Intel Labs

timothy.g.mattson@intel.com



Disclaimer

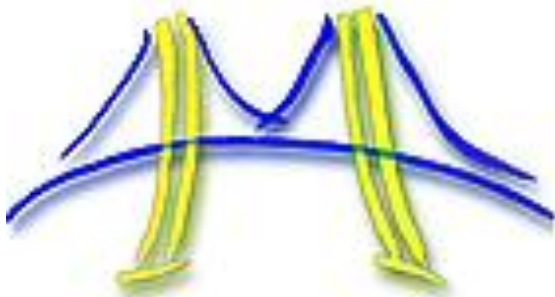


READ THIS ... it is very important

- The views expressed in this talk are those of the speaker and not his employer.
- This is an academic style talk and does not address details of any particular Intel product. You will learn nothing about Intel products from this presentation.
- This was a team effort, but if I say anything really stupid, it's my fault ... don't blame my collaborators.

Acknowledgements

- The work described in this talk comes from Intel's collaboration with the ParLab at UC Berkeley.
 - Patterns: Tim Mattson and Kurt Keutzer
 - SEJITS: Armando Fox and Shoaib Kamil (now at MIT)
 - FTDock: Henry Gabb (now at UIUC) and Tim Mattson
 - PyCASP: Katya Gonina and Kurt Keutzer



This is the logo for the ParLab. I use it here to note slides created in collaboration with ParLab researchers.

Outline

- ➔ • Solving the parallel programming problem
- Patterns and frameworks
- Making code written by productivity programmers run fast
 - FTDOCK: a simple example of SEJITS
 - Stencil: Complex software transformations with SEJITS
 - A few additional SEJITS specializers
 - PyCASP: From Pattern mining to extensible frameworks
- Next Steps and conclusions

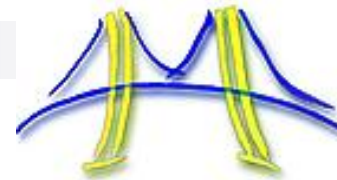
The many core challenge

- A harsh assessment ...
 - We have turned to multi-core chips not because of the success of our parallel software but because of our failure to continually increase CPU frequency.
- Result: a fundamental and dangerous mismatch
 - Parallel hardware is ubiquitous ... Parallel software is rare
- The Many Core challenge ...
 - Parallel software must become as common as parallel hardware

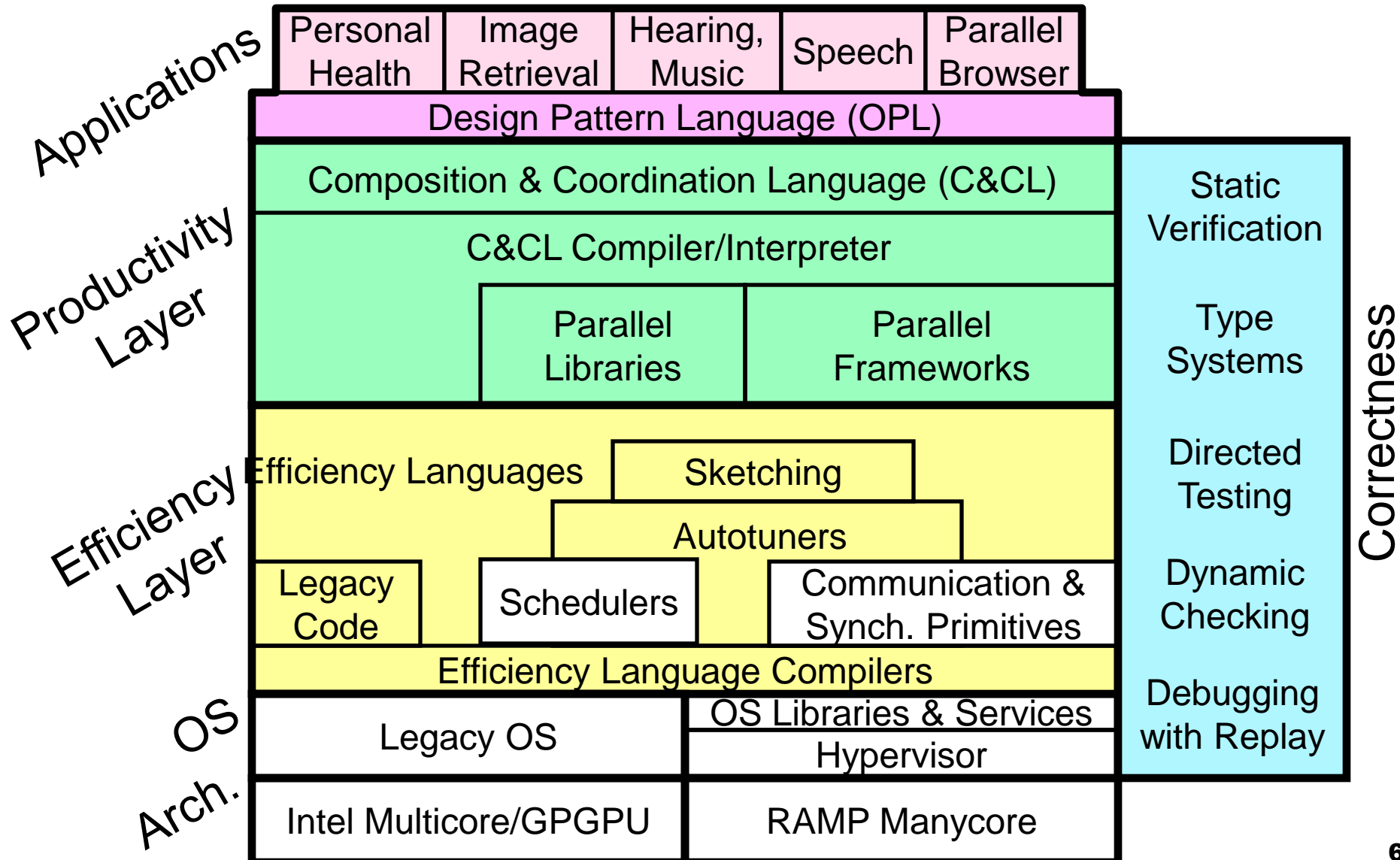
After ~30 years of parallel computing research, we know:
(1) automatic parallelism doesn't work
(2) an endless quest for the perfect parallel language is counterproductive ... "worse is better" (Richard Gabriel, 1991)

So how can we address the many core challenge?

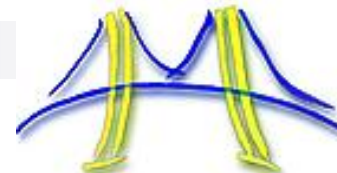
Par Lab Research Overview



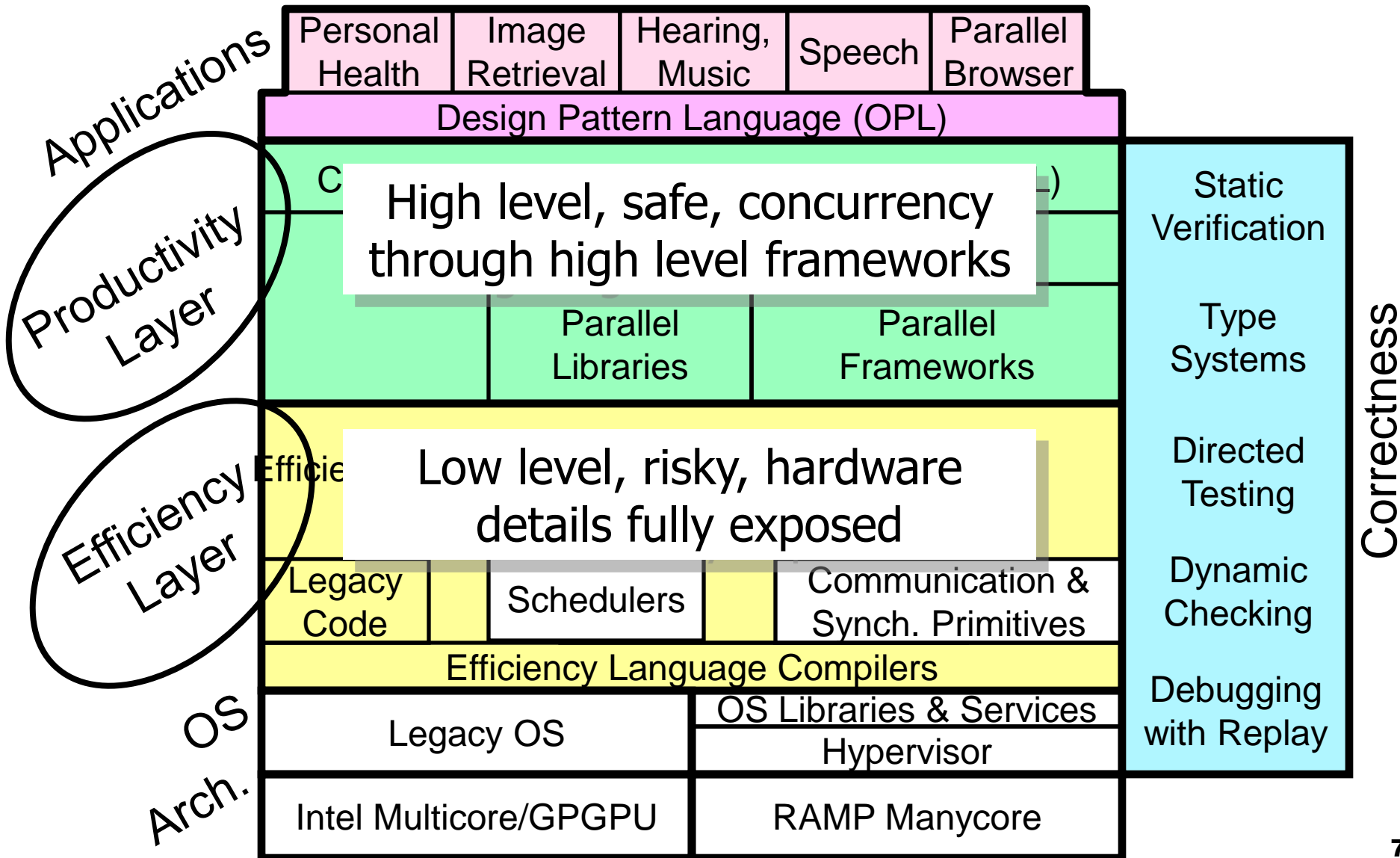
Easy to write correct software that runs efficiently on manycore



Par Lab Research Overview



Easy to write correct software that runs efficiently on manycore

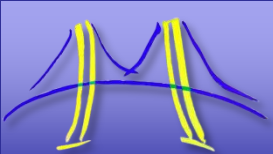


The BIG Vision statement

- We can solve the parallel programming crisis with a programming infrastructure that supports a “Separation of Concerns”:
 - Domain Specialist programmers (e.g. physicists) work at the productivity layer. They “express the concurrency” in their algorithms but pay little attention to “how that concurrency” is exploited on a given platform.
 - Computer scientists work at the efficiency layer to manage how concurrency is exploited and mapped onto the target platform.
- Most programmers (~95%) do not have the time or interest to master the low level details of every platform they use.
- We must create a way for a small number (~5%) of motivated “efficiency layer programmers” to support the domain specialist or “productivity layer programmers”.

Outline

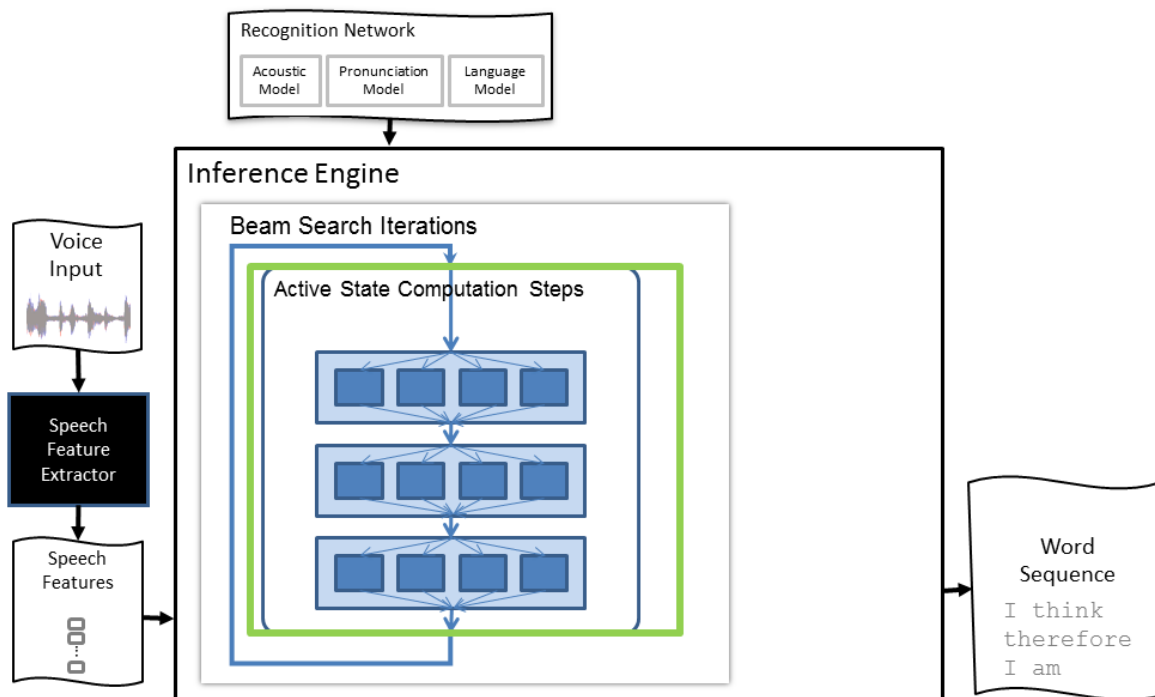
- Solving the parallel programming problem
- ➔ • Patterns and frameworks
- Making code written by productivity programmers run fast
 - FTDOCK: a simple example of SEJITS
 - Stencil: Complex software transformations with SEJITS
 - A few additional SEJITS specializers
 - PyCASP: From Pattern mining to extensible frameworks
- Next Steps and conclusions



Architecting Parallel Software

Keutzer and Mattson, UCB CS164 Spring'2012

- We believe the solution to parallel programming starts with developing a good software architecture



- Example: SW Architecture of Large-Vocabulary Continuous Speech Recognition

How can we systematically describe software architectures?

Our Pattern Language (OPL 2012)

Applications

Structural Patterns

Pipe-and-Filter
Agent-and-Repository
Process-Control
Event-Based/Implicit-Invocation
Arbitrary-Static-Task-Graph

Model-View-Controller
Iterative-Refinement
Map-Reduce
Layered-Systems
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

Finding Concurrency Patterns

Task Decomposition
Data Decomposition

←→

Ordered task groups
Data sharing

↓

Design Evaluation

Parallel Algorithm Strategy Patterns

Task-Parallelism
Divide and Conquer

Data-Parallelism
Pipeline

Discrete-Event
Geometric-Decomposition
Speculation

Implementation Strategy Patterns

SPMD
Kernel-Par.

Fork/Join
Actors
Vector-Par

Loop-Par.
Workpile

Shared-Queue
Shared-Map
Parallel Graph Traversal

Distributed-Array
Shared-Data

Program structure

Algorithms and Data structure

Parallel Execution Patterns

Coordinating Processes
Stream processing

Shared Address Space Threads
Task Driven Execution

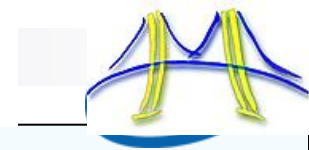
Concurrency Foundation constructs (not expressed as patterns)

Thread/proc management

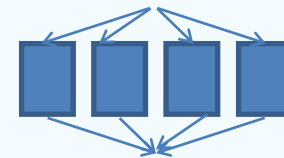
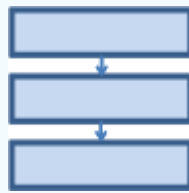
Communication

Synchronization

Pattern examples



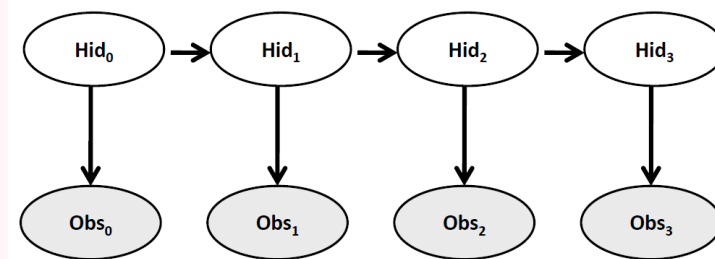
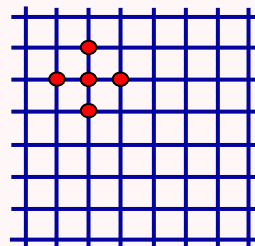
Structural Patterns	Model-View-Controller Iterative-Refinement Map-Reduce Layered-Systems Puppeteer	Computational Patterns	Unstructured-Grids Structured-Grids Graphical-Models Finite-State-Machines Backtrack-Branch-and-Bound N-Body-Methods Circuits Spectral-Methods Monte-Carlo
Pipe and Filter Agent and Repository Process Control Event-Based-Implicit-Invocation Arbitrary-Static-Task-Graph		Graph-Algorithms Dynamic-Programming Dense-Linear-Algebra Sparse-Linear-Algebra	
Finding Concurrency Patterns		Ordered task groups	
Task Decomposition Data Decomposition		Data sharing	
Design Evaluation			
Parallel Algorithm Strategy Patterns		Discrete-Event Geometric-Composition Speculation	
Task-Parallelism Divide-and-Conquer	Data-Parallelism Pipeline		
Implementation Strategy Patterns		Shared-Queue Shared-Map Parallel-Graph-Traversal	Distributed-Array Shared-Data
SPMD Kernel-Par. Program-structure	Fork/Join Actors Vector-Par.	Loop-Par. Workpile	Algorithms and Data structure
Parallel Execution Patterns		Shared-Address-Space-Threads Task-Driven-Execution	
Coordinating-Processes Stream-processing			



- Pipe-and-Filter
- Iterative refinement
- MapReduce

Structural Patterns: Define the software structure .. *Not* what is computed

Structural Patterns	Model-View-Controller Iterative-Refinement Map-Reduce Layered-Systems Puppeteer	Computational Patterns	Unstructured-Grids Structured-Grids Graphical-Models Finite-State-Machines Backtrack-Branch-and-Bound N-Body-Methods Circuits Spectral-Methods Monte-Carlo
Pipe and Filter Agent and Repository Process Control Event-Based-Implicit-Invocation Arbitrary-Static-Task-Graph		Graph-Algorithms Dynamic-Programming Dense-Linear-Algebra Sparse-Linear-Algebra	
Finding Concurrency Patterns		Ordered task groups	
Task Decomposition Data Decomposition		Data sharing	
Design Evaluation			
Parallel Algorithm Strategy Patterns		Discrete-Event Geometric-Composition Speculation	
Task-Parallelism Divide-and-Conquer	Data-Parallelism Pipeline		
Implementation Strategy Patterns		Shared-Queue Shared-Map Parallel-Graph-Traversal	Distributed-Array Shared-Data
SPMD Kernel-Par. Program-structure	Fork/Join Actors Vector-Par.	Loop-Par. Workpile	Algorithms and Data structure
Parallel Execution Patterns		Shared-Address-Space-Threads Task-Driven-Execution	
Coordinating-Processes Stream-processing			



• Structured mesh

• Graphical Models

Computational Patterns: Define the computations “inside the boxes”

Structural Patterns	Model-View-Controller Iterative-Refinement Map-Reduce Layered-Systems Puppeteer	Computational Patterns	Unstructured-Grids Structured-Grids Graphical-Models Finite-State-Machines Backtrack-Branch-and-Bound N-Body-Methods Circuits Spectral-Methods Monte-Carlo
Pipe and Filter Agent and Repository Process Control Event-Based-Implicit-Invocation Arbitrary-Static-Task-Graph		Graph-Algorithms Dynamic-Programming Dense-Linear-Algebra Sparse-Linear-Algebra	
Finding Concurrency Patterns		Ordered task groups	
Task Decomposition Data Decomposition		Data sharing	
Design Evaluation			
Parallel Algorithm Strategy Patterns		Discrete-Event Geometric-Composition Speculation	
Task-Parallelism Divide-and-Conquer	Data-Parallelism Pipeline		
Implementation Strategy Patterns		Shared-Queue Shared-Map Parallel-Graph-Traversal	Distributed-Array Shared-Data
SPMD Kernel-Par. Program-structure	Fork/Join Actors Vector-Par.	Loop-Par. Workpile	Algorithms and Data structure
Parallel Execution Patterns		Shared-Address-Space-Threads Task-Driven-Execution	
Coordinating-Processes Stream-processing			

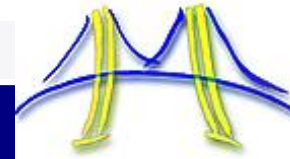
• Fork-join

• SPMD

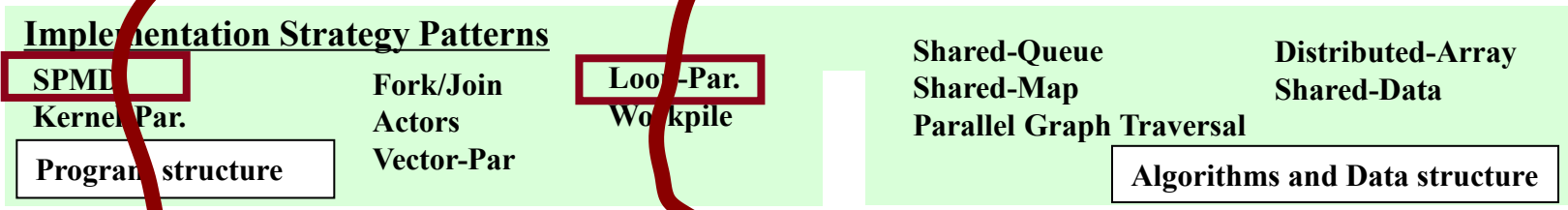
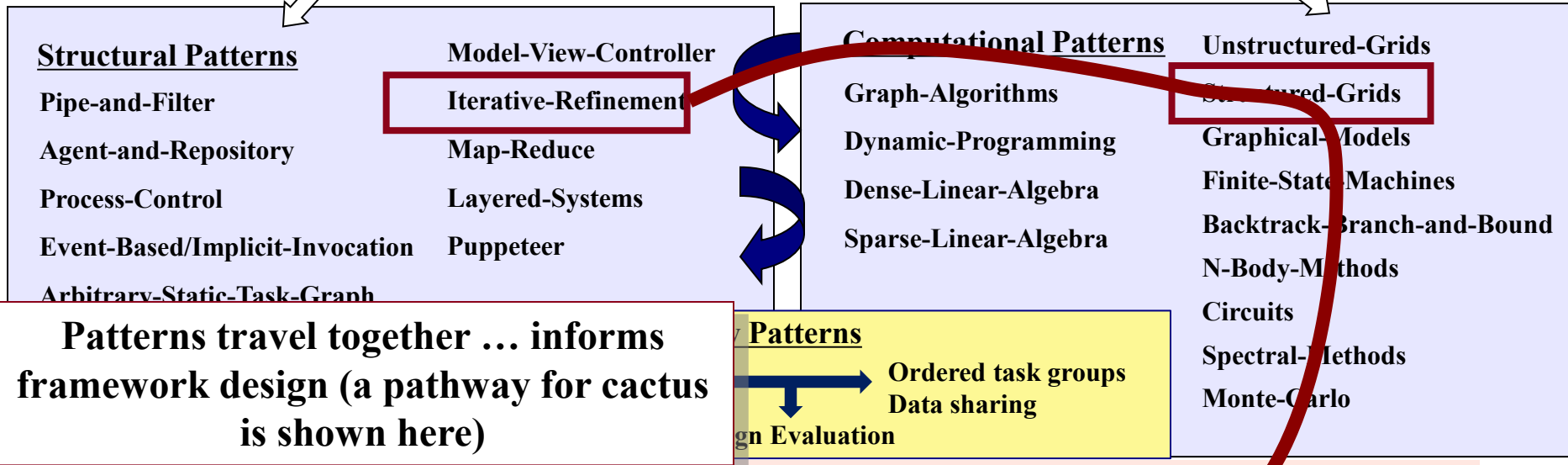
• Data parallel

Parallel Patterns: Defines parallel algorithms

OPL Pattern Language

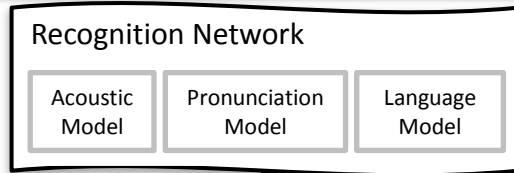
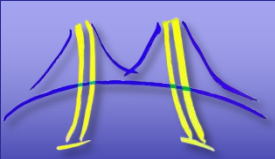


Applications

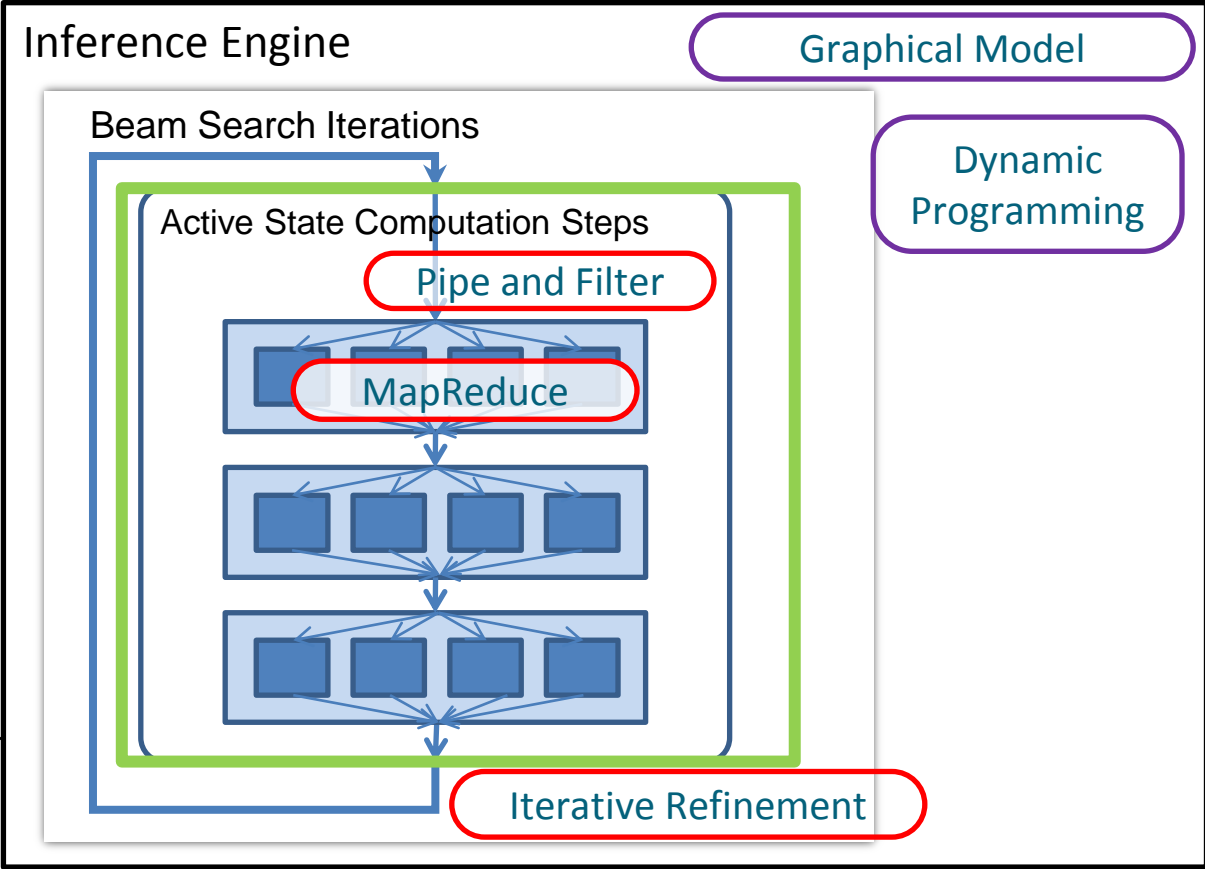


Distributed memory cluster and MPP computers

Multiprocessors (SMP and NUMA)

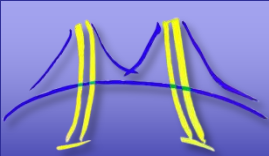


Pipe-and-filter

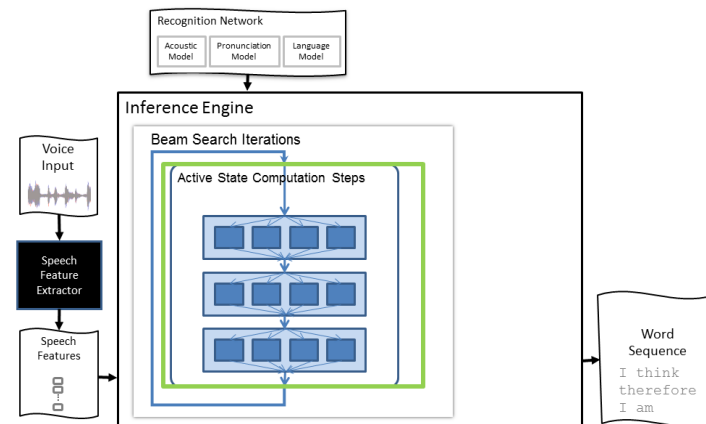


Word Sequence

I think
therefore
I am



- Architecture expressed as a composition of design patterns. Implemented as a C++ Framework.
 - Input: Speech audio waveform
 - Output: Recognized word sequences



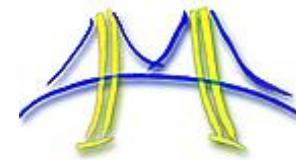
- They “hand tuned” their framework. Achieved impressive results:
 - Achieved 11x speedup (CUDA/GPU) over sequential version (CPU)
 - Allows 3.5x faster than real time recognition.

... But this required C++ experts to map their code onto the details of the target platform. To achieve our goals, we need “productivity” programmers working with a high level language to generate high performance solutions.

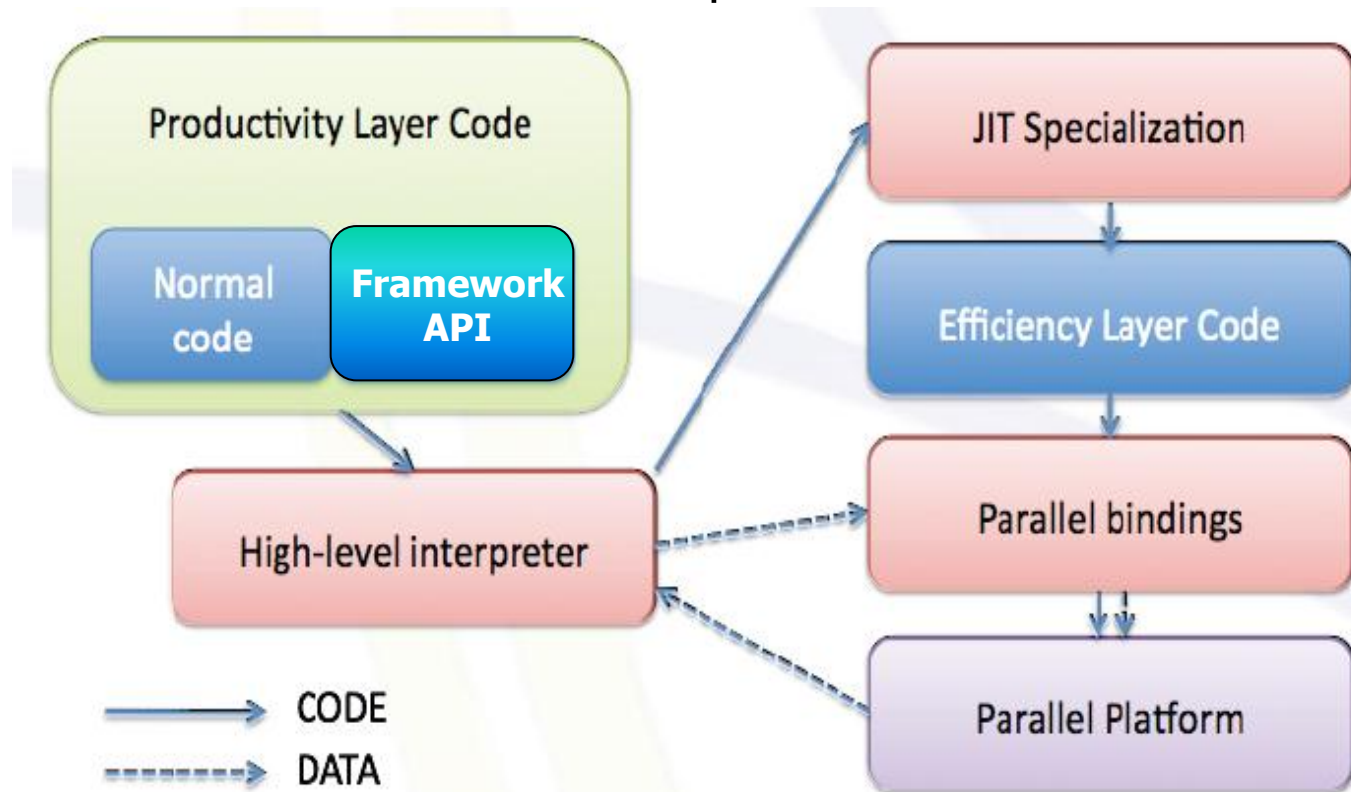
Outline

- Solving the parallel programming problem
- Patterns and frameworks
- ➔ • Making code written by productivity programmers run fast
 - FTDOCK: a simple example of SEJITS
 - Stencil: Complex software transformations with SEJITS
 - A few additional SEJITS specializers
 - PyCASP: From Pattern mining to extensible frameworks
- Next Steps and conclusions

How do we squeeze high performance from framework-based applications?

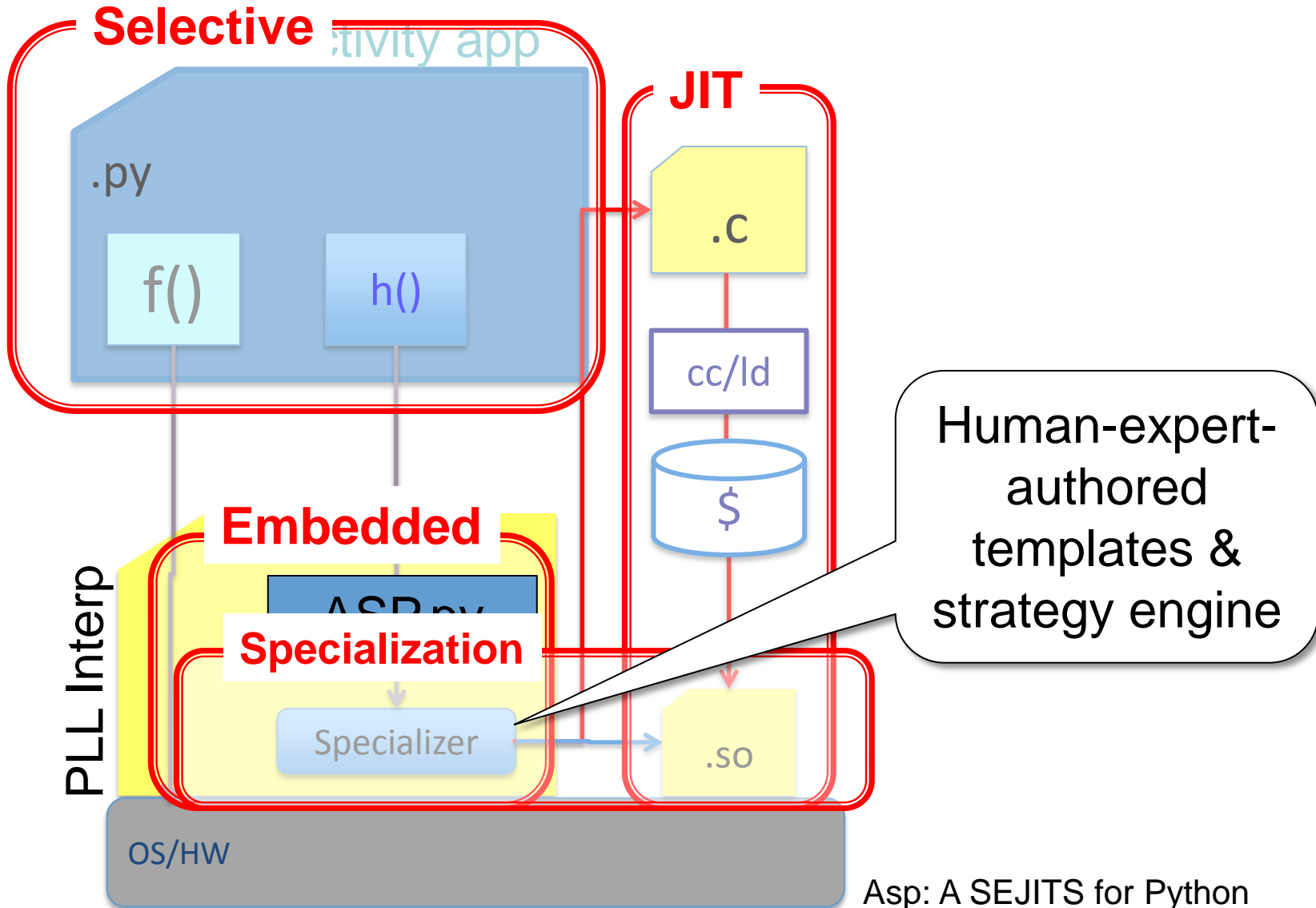
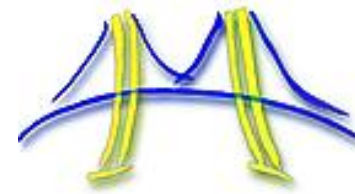


- SEJITS: Scalable, embedded, just in time specialization
 - Code with a high level language (e.g. Python or Ruby) that is mapped onto a low level, efficiency language (e.g. OpenMP/C or CUDA).
 - SEJITS system to embed optimized kernels specialized at runtime to flatten abstraction overhead and map onto hardware features.

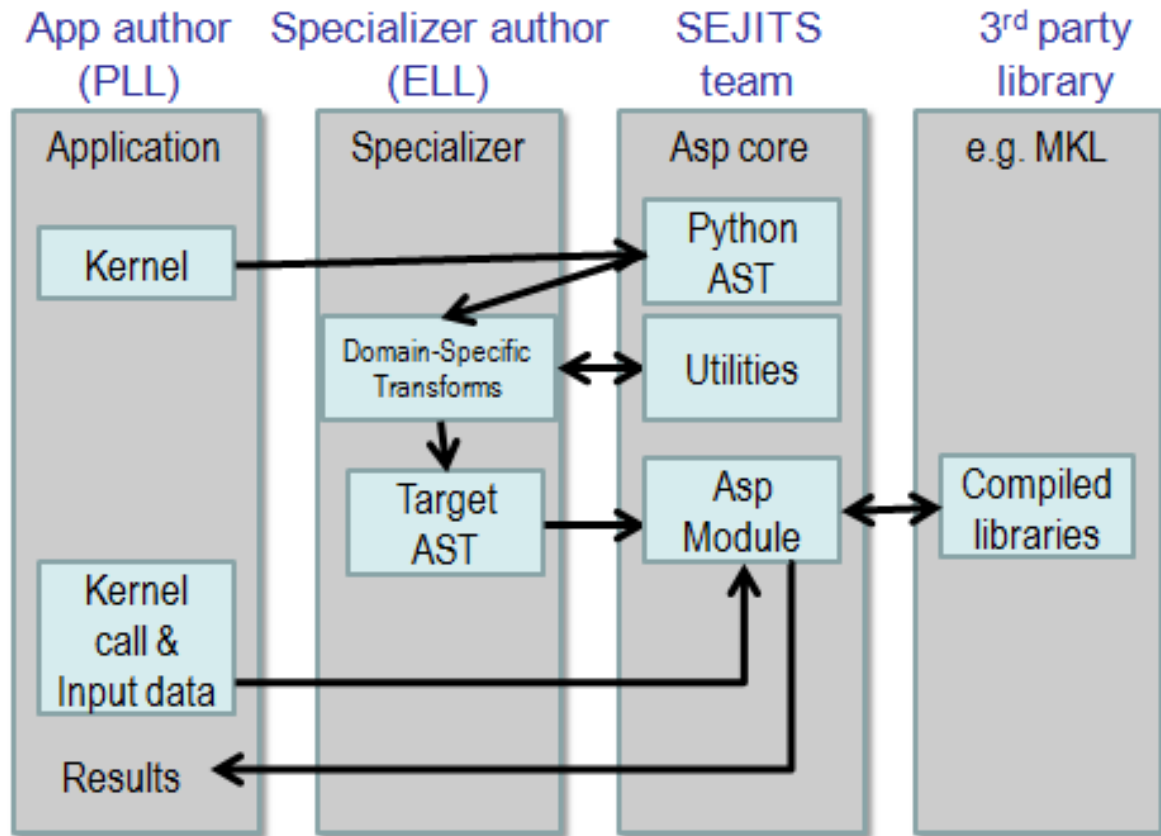
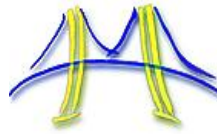


The detailed structure of SEJITS

ASP: A SEJITS for Python



Turning Patterns expressed as Python code into high performance parallel code



ASP ... a platform to write domain specific frameworks.

Helps turn design patterns into code.

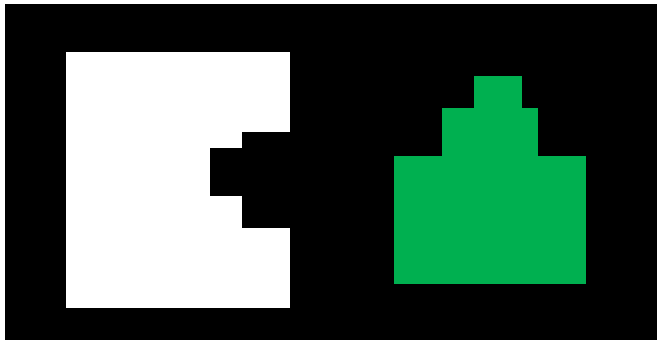
ASP: SEJITS for Python

AST: Abstract Syntax Tree
PLL: Productivity Level Language
ELL: Efficiency Level Language

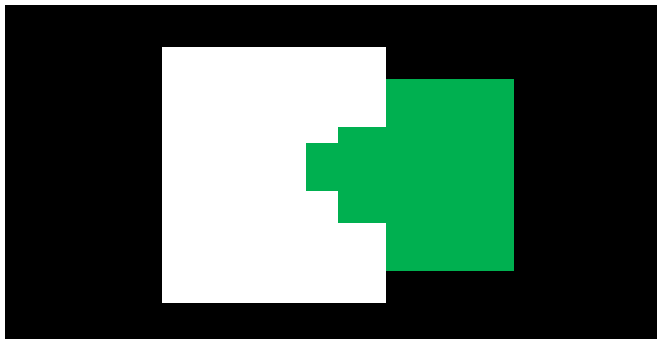
Outline

- Solving the parallel programming problem
- Patterns and frameworks
- Making code written by productivity programmers run fast
 - ➡ – FTDOCK: a simple example of SEJITS
 - Stencil: Complex software transformations with SEJITS
 - A few additional SEJITS specializers
 - PyCASP: From Pattern mining to extensible frameworks
- Next Steps and conclusions

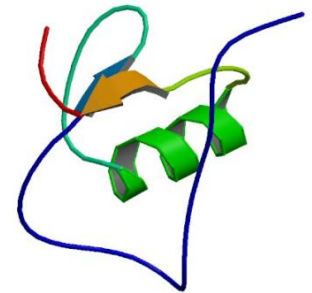
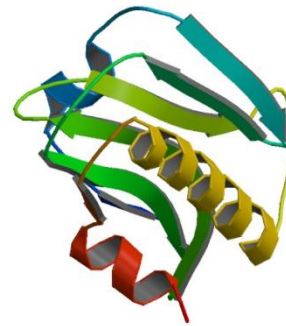
Example Application: Shape Fitting



How do these two shapes fit together?



Pretty obvious.



How do *these* two shapes fit together? Not as obvious when dealing with complex, 3D molecular structures.

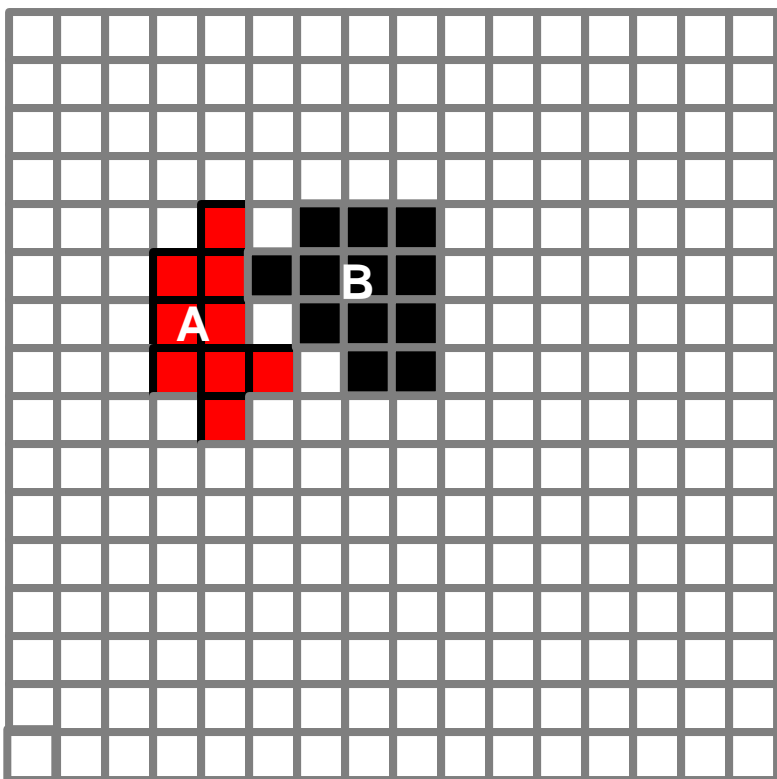
Why does it matter how molecules fit together? Because most biological processes involve molecular binding.

This is the first “external” project to use SEJITS:

- Productivity prog: Henry Gabb (UIUC)
- Efficiency prog: Tim Mattson (Intel)

Shape Fitting by Cartesian Grid Correlations

Project molecules A and B onto a grid and assign values to nodes based on locations of atoms.



$$C_{\alpha\beta\gamma} = \sum_{i=1}^N \sum_{j=1}^N \sum_{k=1}^N A_{ijk} \times B_{i+\alpha, j+\beta, k+\gamma}$$

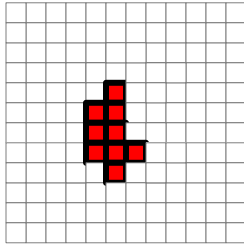
Translate/rotate molecules to maximize the correlation.

Inefficient: $O(N^6)$, N^3 additions and multiplications for every N^3 translations (α, β, γ).

Solve more efficiently using Fourier correlation: $O(N^3 \log N^3)$.

Application "Box-and-Arrow" Diagram

Molecule A



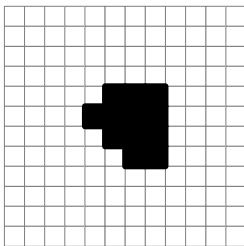
Fourier Transform



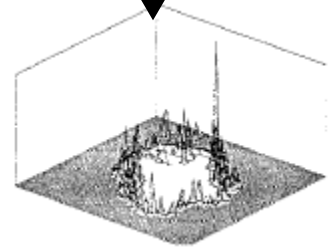
Complex Conjugate



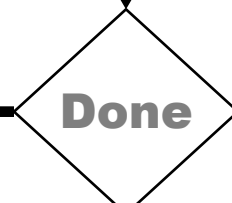
Molecule B



Fourier Transform



Fourier Correlation



Sort Geometries

Yes

No



Rotate



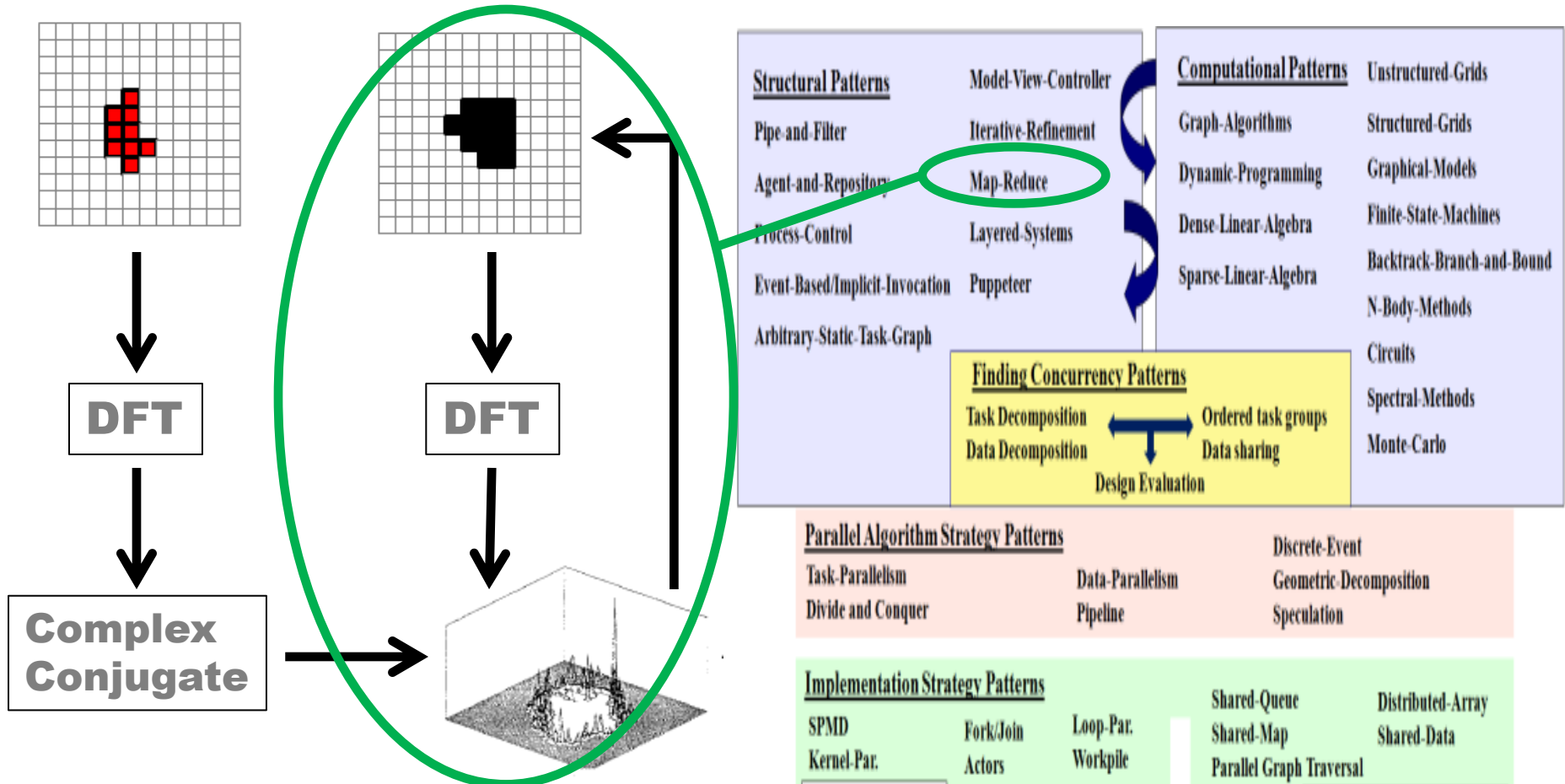
Productivity Programmer Responsibilities

Original loop-based, iterative code:

```
for a in range(-1.0, 1.0 + del, del):  
    for b in range(-1.0, 1.0 + del, del):  
        for g in range(-1.0, 1.0 + del, del):  
  
            # ftdock algorithm
```

**The productivity programmer knows
the body of this loop-nest is
“embarrassingly parallel” ... but it is
prohibitively difficult for a compiler
could figure this out**

Parallel Design Patterns



To expose the most concurrency in a natural way, it was best to recast the problem in terms of map-reduce.

i.e. the productivity programmer is responsible for a good design.

Productivity Programmer Responsibilities

Original loop-based, iterative code:

```
for a in range(-1.0, 1.0 + del, del):  
    for b in range(-1.0, 1.0 + del, del):  
        for g in range(-1.0, 1.0 + del, del):  
  
            # ftdock algorithm
```

New Code inspired by the map-reduce pattern:

```
a = b = g = list(range(-1.0, 1.0 + del, del))  
geometries = AllCombMap([a, b, g], ftdock, *args)
```

A SEJITS specializer “hooked into” AllCombMap() and mapped it onto a low level programming language to support parallel execution.

SEJITS/FTDock Results

- What SEJITS did for FTDock
 - Parallelism exploited through a map-reduce module
 - Used SEJITS templating tools to replace original FFTs with FFTW... with no changes to application code.
- Minimal burden on productivity programmer:
 - Pattern-based design of application
 - Functional programming style
 - Significantly easier development:
 - Original version: 4,700 lines of C and Perl
 - New version: 500 lines of Python
 - *Caveat: LOC not necessarily a good measure of productivity*
- Performance (16-core Xeon):
 - Serial: ~24 hours
 - Parallel: ~3 hours

Incorporating new specializers

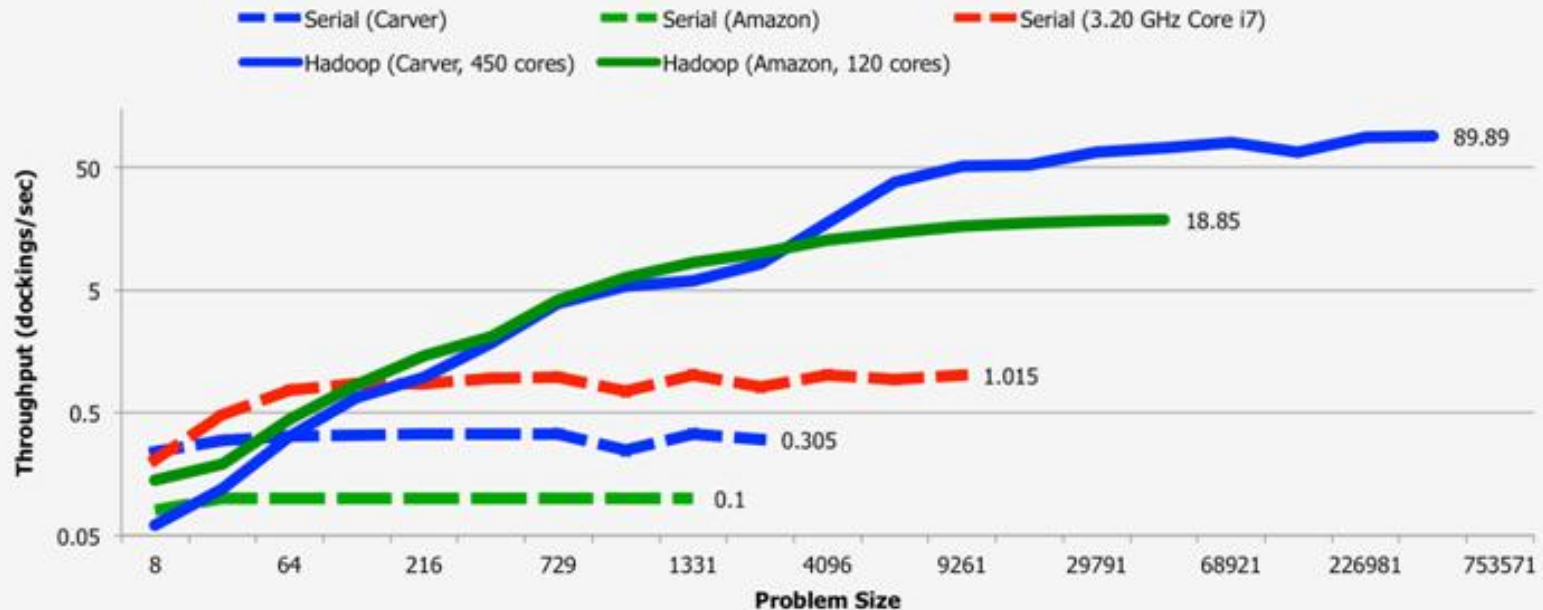
FTDock – Protein Docking

- Independent dockings in 3D search space
- Requires one-line change to application.
- Achieves **290x speedup** on 450 cores.

```
class FtdockMapper(  
    def mapper(  
        args = self.  
        score = ftdo  
        yield 1, score
```

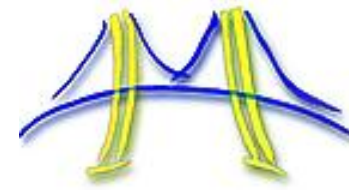
Independently, the ParLab team created a new specializer to map this onto “the cloud” ... an example of the power of the “separation of concerns” concept.

FTDock Throughput vs. Problem Size



Outline

- Solving the parallel programming problem
- Patterns and frameworks
- Making code written by productivity programmers run fast
 - FTDOCK: a simple example of SEJITS
 - ➡ – Stencil: Complex software transformations with SEJITS
 - A few additional SEJITS specializers
 - PyCASP: From Pattern mining to extensible frameworks
- Next Steps and conclusions

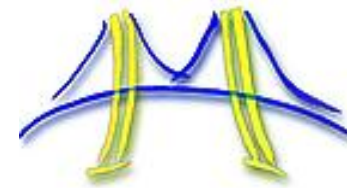


Example Code

```
from stencil_kernel import *

class Laplacian3D(StencilKernel):

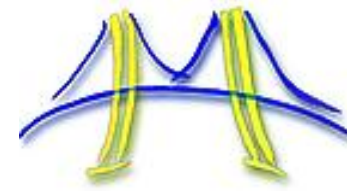
    def kernel(self, in_grid, out_grid):
        for x in self.interior_points(out_grid):
            for y in self.neighbors(in_grid, x, 1):
                out_grid[x] += (1.0/6.0) * in_grid[y]
```



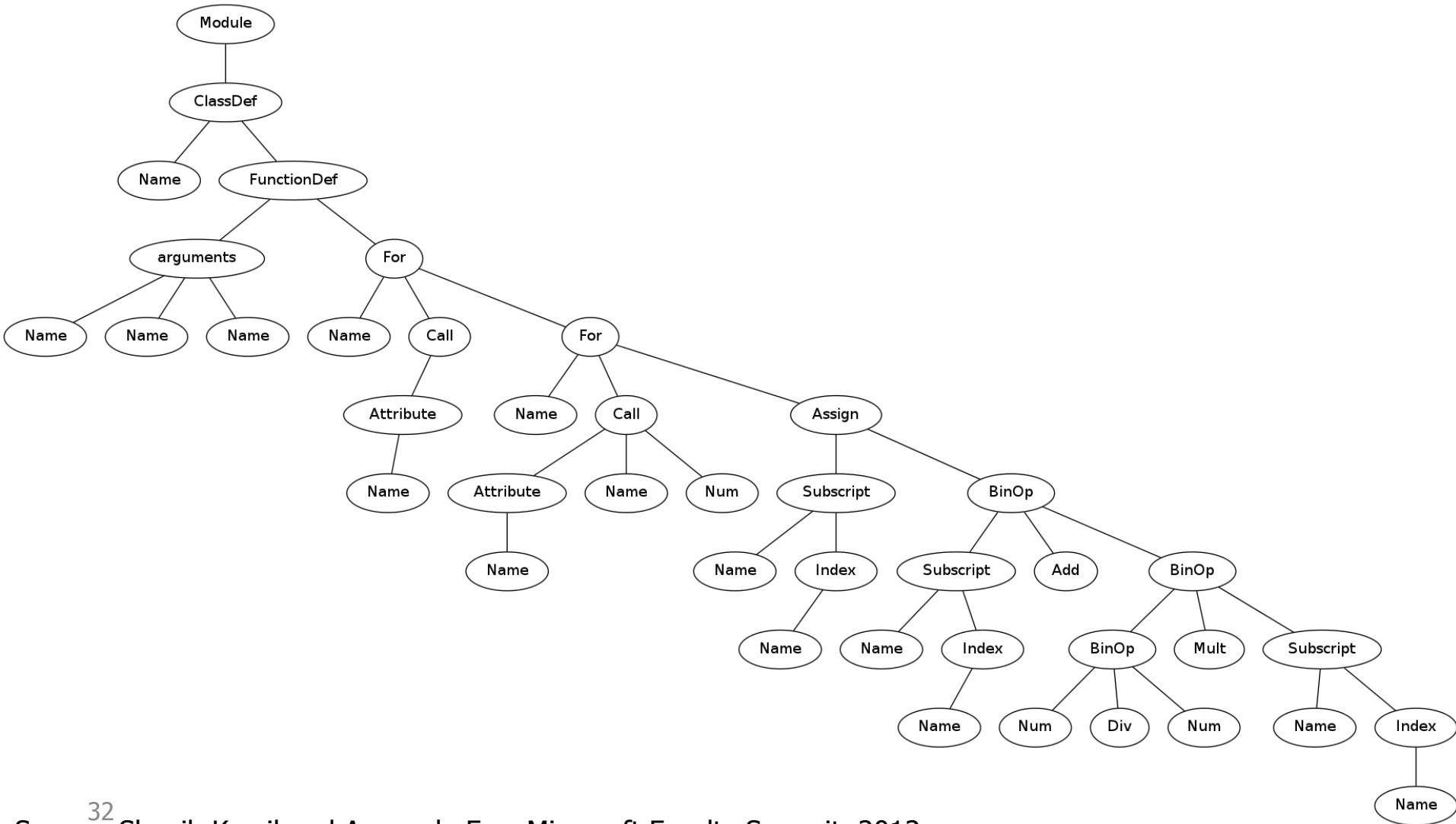
Example Code

```
from stencil_kernel import *  
  
class Laplacian3D(StencilKernel):  
  
    def kernel(self, in_grid, out_grid):  
        for x in self.interior_points(out_grid):  
            for y in self.neighbors(in_grid, x, 1):  
                out_grid[x] += (1.0/6.0) * in_grid[y]
```

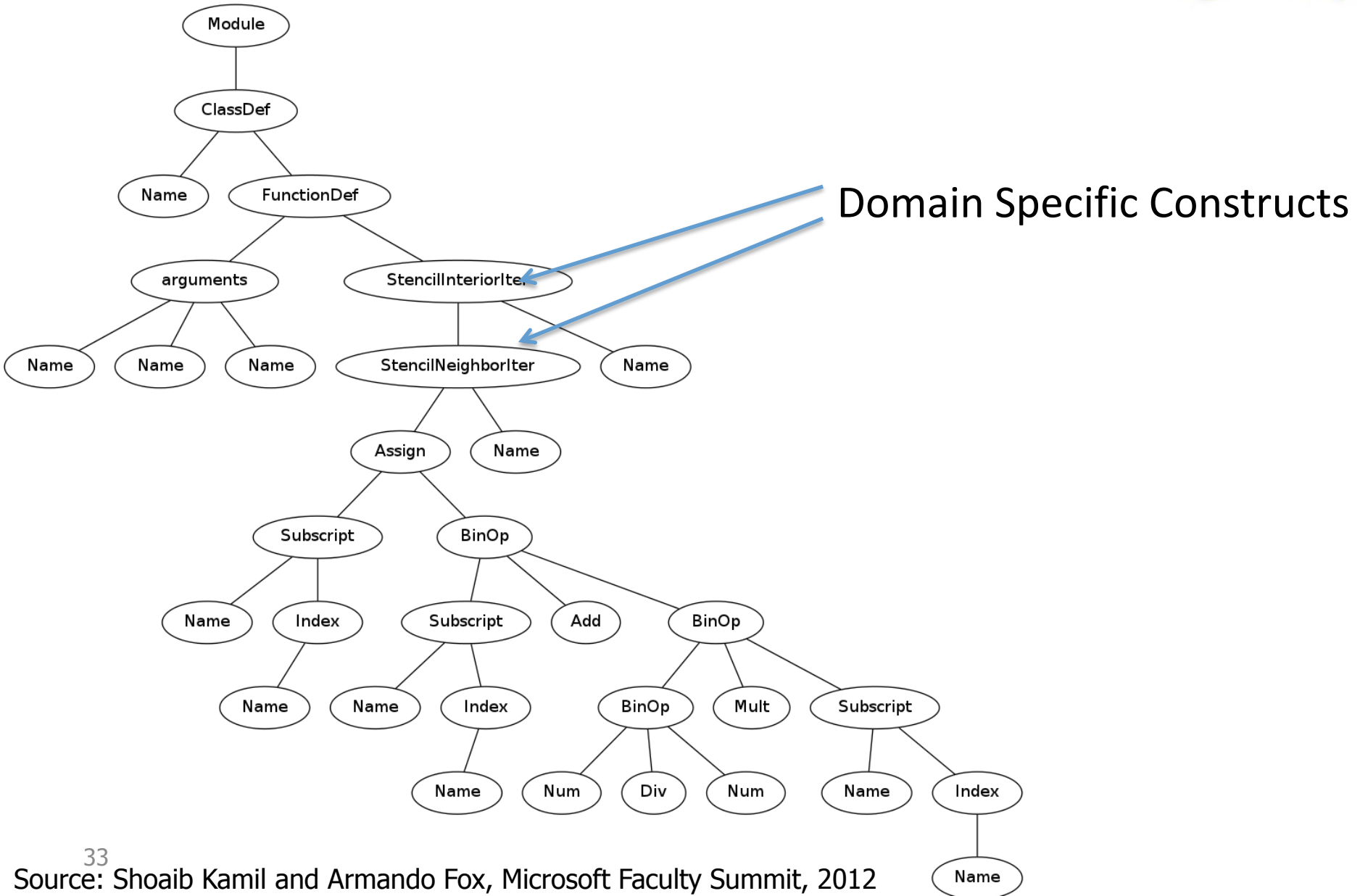
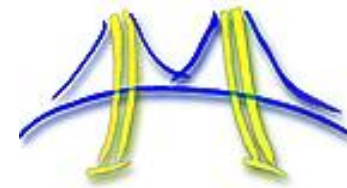
An instance of the structured Grid computational pattern mapped onto the loop-level parallelism pattern.



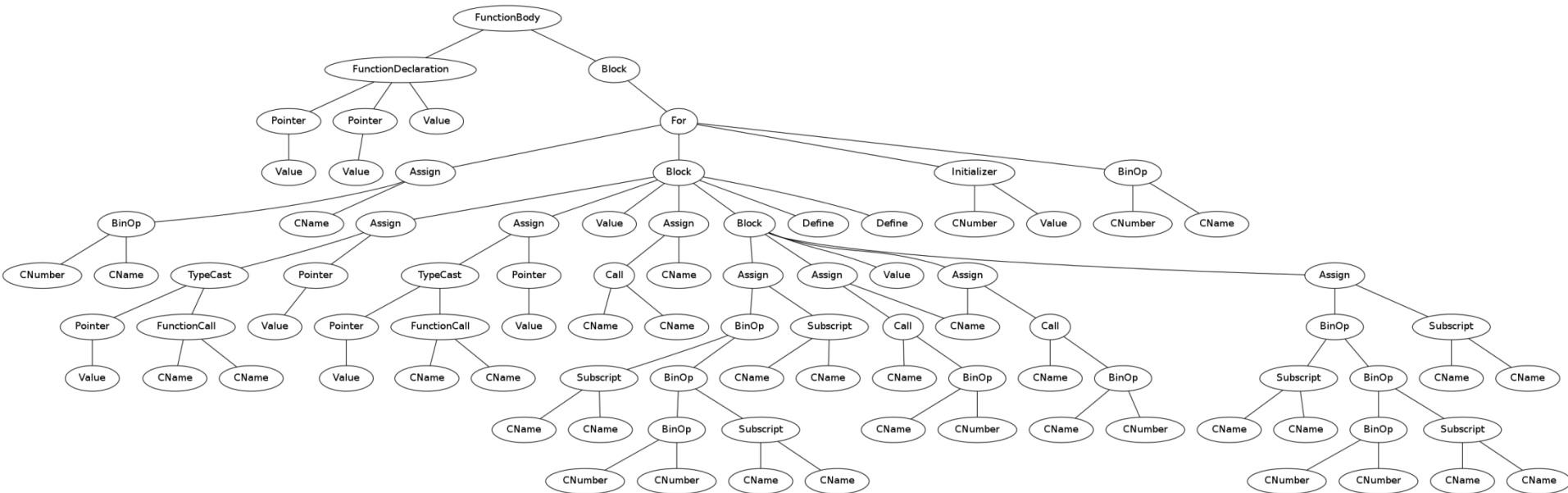
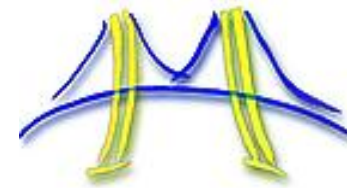
Introspect to Get AST



Transform into IR

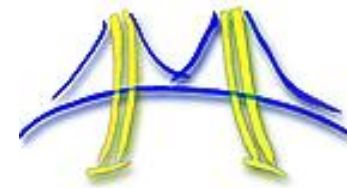


Transform into Platform AST & Optimize



- This is the phase of the infrastructure where the efficiency programmer does most of his/her work
- Use of Asp's infrastructure for common transformations

Optimized Output



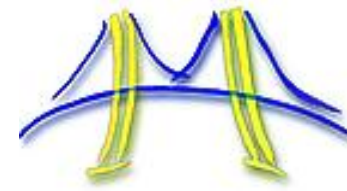
Parallelization

```
void
kernel_optimized(double* in_grid, double* out_grid) {
#define min(_a,_b) (_a < _b ? _a : _b)
#define _idx(_d0,_d1,_d2) (_d2+(_d0 * 258*258)+(_d1 * 258))

  for (int x1x1 = 1; (x1x1 <= 256); x1x1 = (x1x1 + (1 * 192))) {
    for (int x2x2 = 1; (x2x2 <= 256); x2x2 = (x2x2 + (1 * 160))) {
#pragma omp parallel for
      for (int x1 = x1x1; (x1 <= min((x1x1 + 191), 256)); x1 = (x1 + 1)) {
        for (int x2 = x2x2; (x2 <= min((x2x2 + 159), 256)); x2 = (x2 + 1)) {
#pragma ivdep
          for (int x3 = 1; (x3 <= (256 - 3)); x3 = (x3 + (1 * 4))) {
            int x4;
            x4 = __idx(x1, x2, x3);
            out_grid[x4] = (out_grid[x4] + ((1.0 / 6.0) * in_grid[_idx((x1 + 1), (x2 + 0), (x3 + 0))]));
            out_grid[x4] = (out_grid[x4] + ((1.0 / 6.0) * in_grid[_idx((x1 + -1), (x2 + 0), (x3 + 0))]));
            out_grid[x4] = (out_grid[x4] + ((1.0 / 6.0) * in_grid[_idx((x1 + 0), (x2 + 1), (x3 + 0))]));
            out_grid[x4] = (out_grid[x4] + ((1.0 / 6.0) * in_grid[_idx((x1 + 0), (x2 + -1), (x3 + 0))]));
            out_grid[x4] = (out_grid[x4] + ((1.0 / 6.0) * in_grid[_idx((x1 + 0), (x2 + 0), (x3 + 1))]));
            out_grid[x4] = (out_grid[x4] + ((1.0 / 6.0) * in_grid[_idx((x1 + 0), (x2 + 0), (x3 + -1))]));
            x4 = __idx(x1, x2, (x3 + 1));
            out_grid[x4] = (out_grid[x4] + ((1.0 / 6.0) * in_grid[_idx((x1 + 1), (x2 + 0), ((x3 + 1) + 0))]));
            out_grid[x4] = (out_grid[x4] + ((1.0 / 6.0) * in_grid[_idx((x1 + -1), (x2 + 0), ((x3 + 1) + 0))]));
            out_grid[x4] = (out_grid[x4] + ((1.0 / 6.0) * in_grid[_idx((x1 + 0), (x2 + 1), ((x3 + 1) + 0))]));
            out_grid[x4] = (out_grid[x4] + ((1.0 / 6.0) * in_grid[_idx((x1 + 0), (x2 + -1), ((x3 + 1) + 0))]));
            out_grid[x4] = (out_grid[x4] + ((1.0 / 6.0) * in_grid[_idx((x1 + 0), (x2 + 0), ((x3 + 1) + 1))]));
            out_grid[x4] = (out_grid[x4] + ((1.0 / 6.0) * in_grid[_idx((x1 + 0), (x2 + 0), ((x3 + 1) + -1))]));
            x4 = __idx(x1, x2, (x3 + 2));
            out_grid[x4] = (out_grid[x4] + ((1.0 / 6.0) * in_grid[_idx((x1 + 1), (x2 + 0), ((x3 + 2) + 0))]));
            out_grid[x4] = (out_grid[x4] + ((1.0 / 6.0) * in_grid[_idx((x1 + -1), (x2 + 0), ((x3 + 2) + 0))]));
            out_grid[x4] = (out_grid[x4] + ((1.0 / 6.0) * in_grid[_idx((x1 + 0), (x2 + 1), ((x3 + 2) + 0))]));
            out_grid[x4] = (out_grid[x4] + ((1.0 / 6.0) * in_grid[_idx((x1 + 0), (x2 + -1), ((x3 + 2) + 0))]));
            out_grid[x4] = (out_grid[x4] + ((1.0 / 6.0) * in_grid[_idx((x1 + 0), (x2 + 0), ((x3 + 2) + 1))]));
            out_grid[x4] = (out_grid[x4] + ((1.0 / 6.0) * in_grid[_idx((x1 + 0), (x2 + 0), ((x3 + 2) + -1))]));
            x4 = __idx(x1, x2, (x3 + 3));
            out_grid[x4] = (out_grid[x4] + ((1.0 / 6.0) * in_grid[_idx((x1 + 1), (x2 + 0), ((x3 + 3) + 0))]));
            out_grid[x4] = (out_grid[x4] + ((1.0 / 6.0) * in_grid[_idx((x1 + -1), (x2 + 0), ((x3 + 3) + 0))]));
            out_grid[x4] = (out_grid[x4] + ((1.0 / 6.0) * in_grid[_idx((x1 + 0), (x2 + 1), ((x3 + 3) + 0))]));
            out_grid[x4] = (out_grid[x4] + ((1.0 / 6.0) * in_grid[_idx((x1 + 0), (x2 + -1), ((x3 + 3) + 0))]));
            out_grid[x4] = (out_grid[x4] + ((1.0 / 6.0) * in_grid[_idx((x1 + 0), (x2 + 0), ((x3 + 3) + 1))]));
            out_grid[x4] = (out_grid[x4] + ((1.0 / 6.0) * in_grid[_idx((x1 + 0), (x2 + 0), ((x3 + 3) + -1))]));
          }
        }
      }
    }
  }
}
```

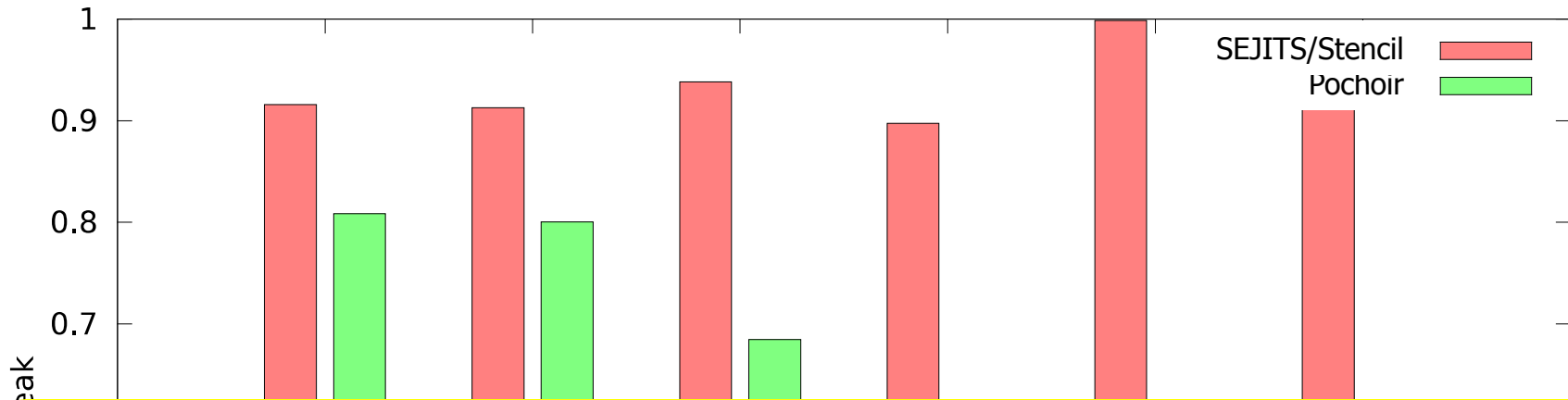
Cache blocking

Unrolling/register blocking



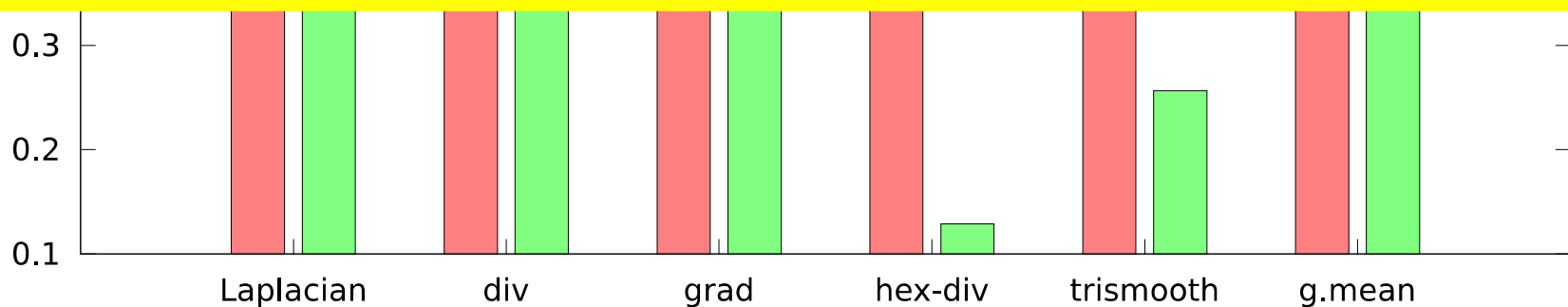
Updates: Stencil DSEL Performance

Structured Grid Fraction of Peak Performance (postbop)



~2.5x faster than Pochoir

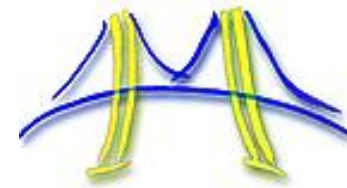
Geometric mean of 93% of attainable peak.



Outline

- Solving the parallel programming problem
- Patterns and frameworks
- Making code written by productivity programmers run fast
 - FTDOCK: a simple example of SEJITS
 - Stencil: Complex software transformations with SEJITS
 - ➡ – A few additional SEJITS specializers
 - PyCASP: From Pattern mining to extensible frameworks
- Next Steps and conclusions

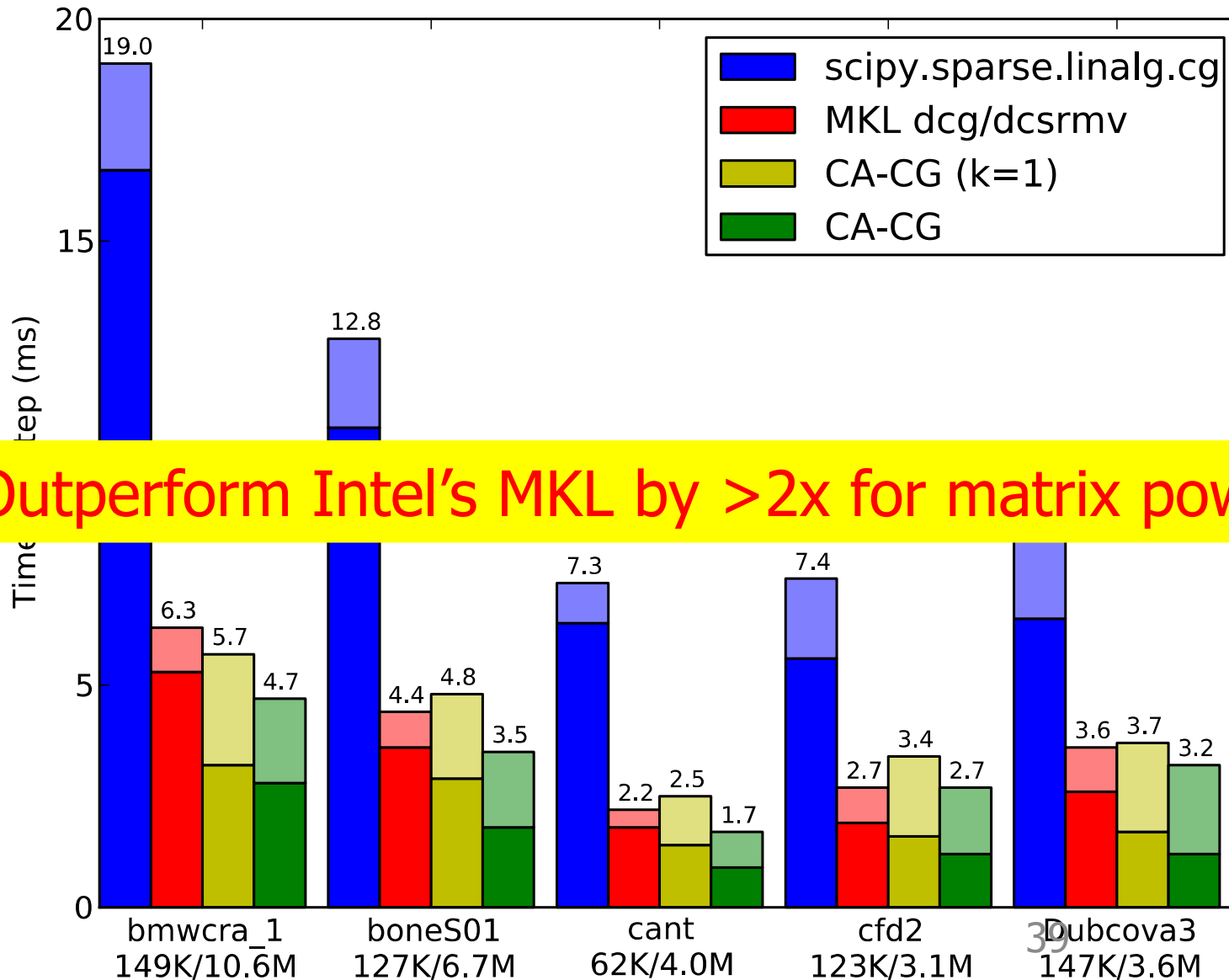
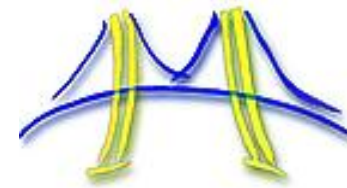
Implemented Specializers/Libraries



DSEL/Library	Platforms
Stencil/Structured Grid	x86+OpenMP
Semantic Graphs Filtering & Semiring Operations in KDT	x86+MPI
Parallel Map	x86+processes, cloud
Gaussian Mixture Modeling	CUDA, Cilk Plus
CA Matrix Powers for CA Krylov Subspace Methods	x86+pthreads
Bag of Little Bootstraps*	x86+Cilk Plus, Cloud via Spark
GraphLab DSEL for Machine Learning via Graphs*	x86+pthreads
CA Parallel Recursive Structural Pattern*	x86+Cilk Plus

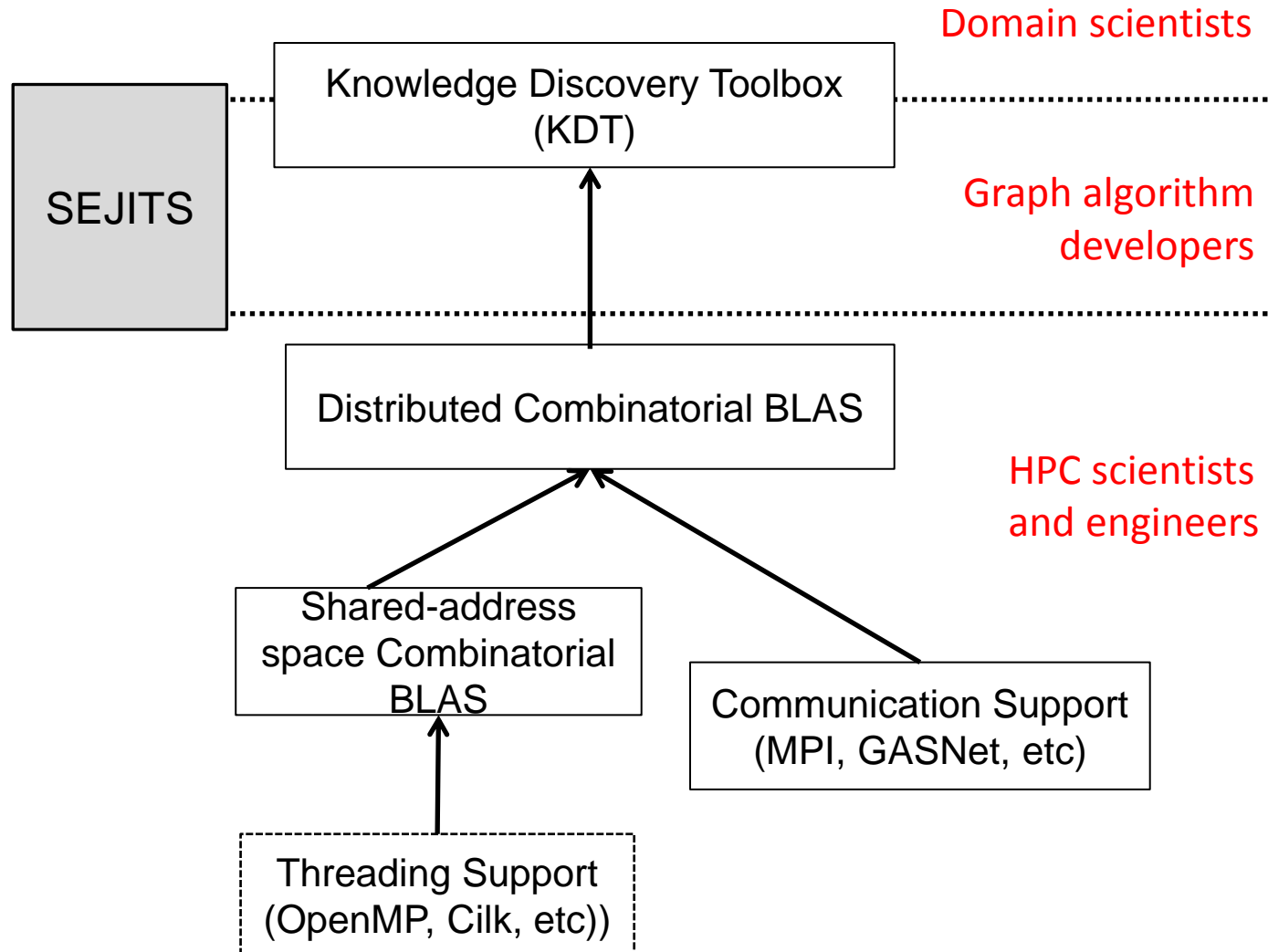
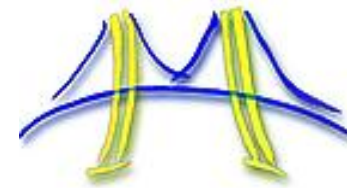
Under development

Updates: CA (communication avoiding) Powers Performance

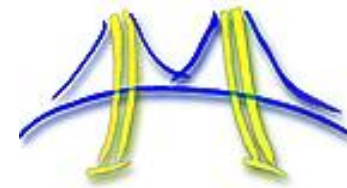


Outperform Intel's MKL by >2x for matrix powers.

Knowledge Discovery Toolbox



Specializers for Filtering in KDT



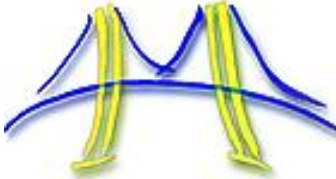
- Perform graph algorithm on subset of graph
- Allows writing filters that return True if and only if edge should be included

```
class MyFilter(PcbFilter):
    def __init__(self, target_date):
        self.target = strtotime(target_date)

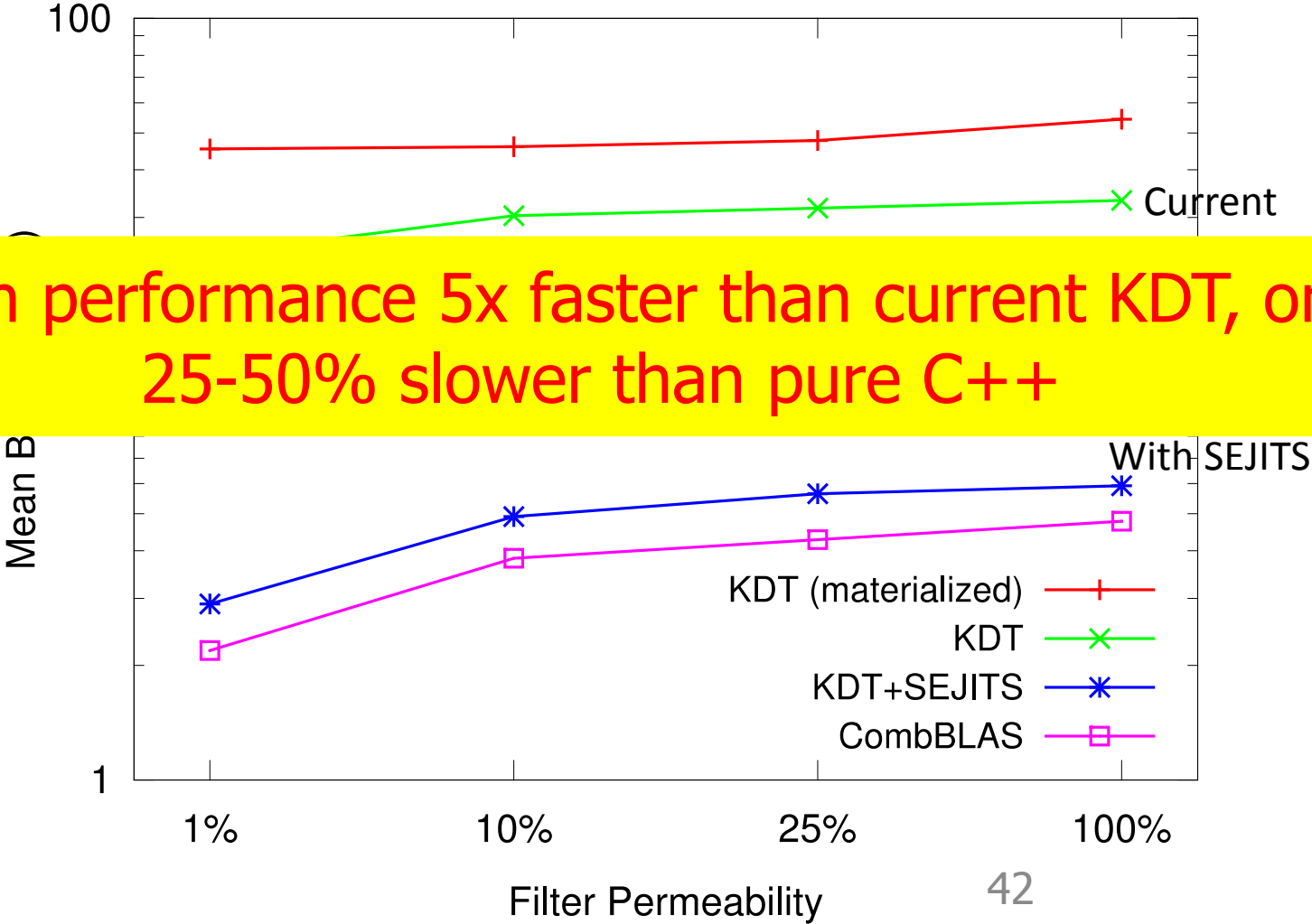
    def filter(e):
        # if it is a retweet edge
        if (e.count > 0 and
            # and it is before the target date
            e.latest < self.target):
            return True
        else:
            return False
```

SEJITS "inlines"
this filter into the
C++ code that
carries out the
graph algorithm.

Filter Specializers Results



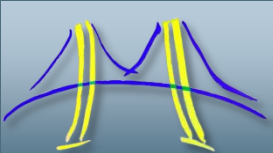
BFS Performance With Varying Filter Permeability (boxboro)



Python performance 5x faster than current KDT, only 25-50% slower than pure C++

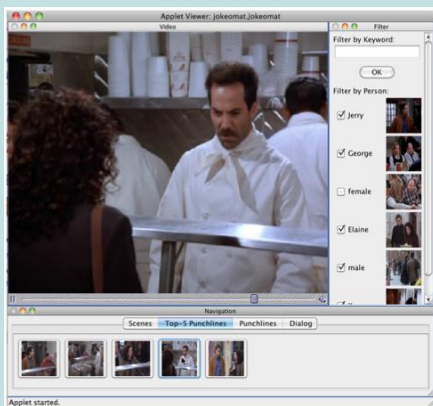
Outline

- Solving the parallel programming problem
- Patterns and frameworks
- Making code written by productivity programmers run fast
 - FTDOCK: a simple example of SEJITS
 - Stencil: Complex software transformations with SEJITS
 - A few additional SEJITS specializers
- ➡ – PyCASP: From Pattern mining to extensible frameworks
- Next Steps and conclusions



Application driven Framework development

Speaker Diarization



- Who spoke when?
 - 20 – 60 min meeting recordings

corpus.amiproject.org/

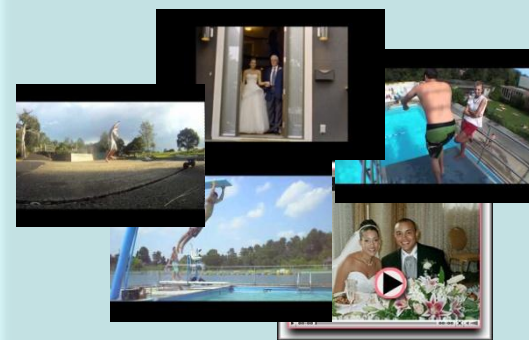
Music Recommendation



- Recommend songs most similar to a query

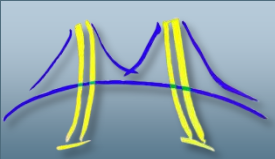
1 Million Song Dataset
labrosa.ee.columbia.edu/millionsong/

Video Event Detection



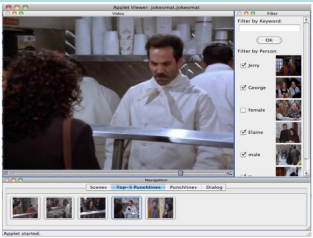
- Detect events in videos based on the soundtrack
- 1-50K video files

www-nlpir.nist.gov/projects/tv2011/



Mining Patterns from Multi media Content Analysis

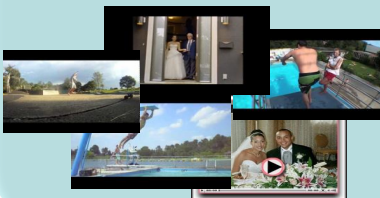
Speaker Diarization



Music Recommendation



Video Event Detection



Application Patterns

Convolution

Orthogonal Transformations (FFT, DCT, Wavelets)

Parametric Clustering (GMM, K-means)

Agglomerative Hierarchical Modeling

Probabilistic Networks (HMM, DBN)

Neural Networks

Eigen Decomposition

Computational Patterns

Dense Linear Algebra

Parametric Clustering (GMM, K-means)

Neural Networks

Graph Algorithms

Probabilistic Networks (HMM, DBN)

Agglomerative Hierarchical Modeling

Spectral Methods

Orthogonal Transformations (FFT, DCT, Wavelets)

Sparse Linear Algebra

Eigen Decomposition

Structured Grids

Convolution



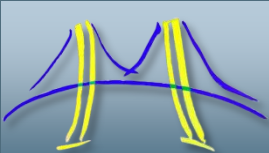
Structural Patterns

MapReduce

Iterative Refinement

Pipe and Filter





What the Framework Will Look Like

Library Components



```
def AHC(self):
    # Get the events, divide them into an initial k clusters and train each GMM on a cluster
    per_cluster = self.N/self.init_num_clusters
    init_training = zip(self.gmm_list,np.vsplit(self.X, range(per_cluster, self.N, per_cluster)))
```

FFT

```
while (best_BIC_score > 0 and len(self.gmm_list) > 1):
```

```
    num_clusters = len(self.gmm_list)
```

```
    # Resegment data based on likelihood scoring
    likelihoods = self.gmm_list[0].score(self.X)
    for g in self.gmm_list[1:]:
        likelihoods = np.column_stack((likelihoods, g.score (self.X)))
    most_likely = likelihoods.argmax(axis=1)
```

```
    # Across 2.5 secs of observations, vote on which cluster they should be associated with
    split_events = split_events_based_on_votes(most_likely, self.X)
```

```
    for g, data in split_events:
        g.train(data)
```

```
    # Score all pairs of GMMs using BIC
    best_merged_gmm = None
    best_BIC_score = 0.0
    merged_tuple = None
```

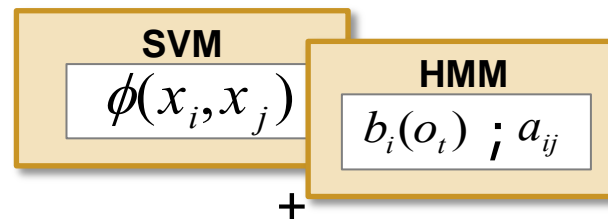
```
    for gmmidx in range(len(merged_tuple)):
```

HMM

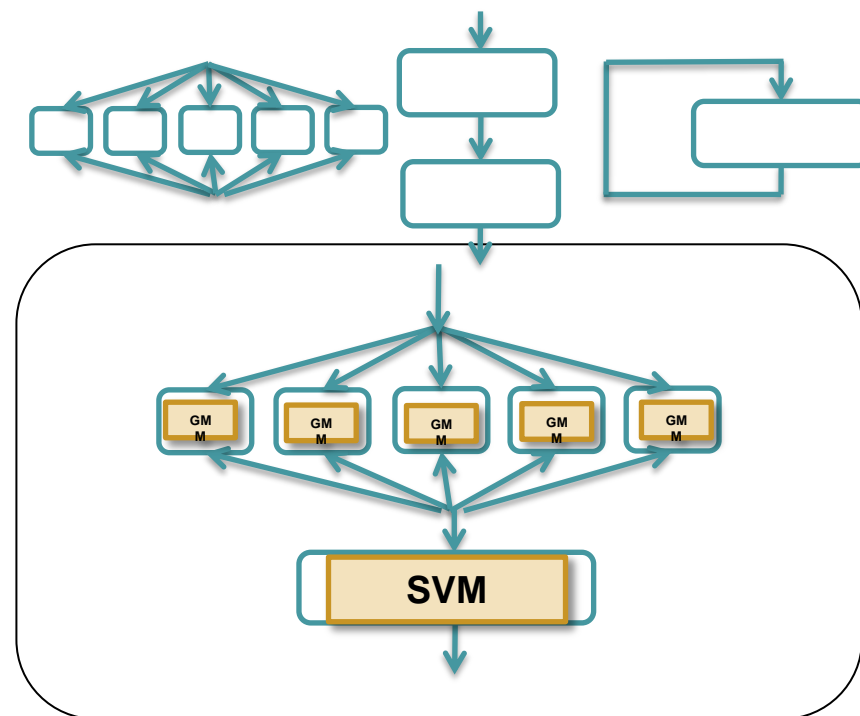
$$b_i(o_t) ; a_{ij}$$

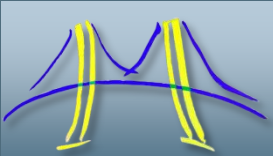
```
    # Merge the winning candidate pair
    if best_BIC_score > 0.0:
        self.gmm_list.remove(merged_tuple[0])
        self.gmm_list.remove(merged_tuple[1])
        self.gmm_list.append(best_merged_gmm)
```

Customizable Components



Structural Patterns





Library Component Example: GMM EM Training

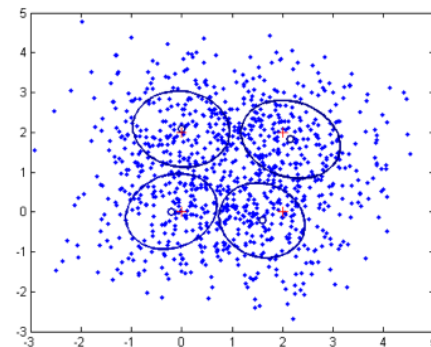
- GMM = probabilistic model for clustering data

GMM

$$p(x | \theta) = \sum_i \pi_i \frac{1}{(2\pi)^{\frac{m}{2}} |\Sigma_i|^{\frac{1}{2}}} \exp\left\{-\frac{1}{2}(x - \mu_i)^T \Sigma_i^{-1} (x - \mu_i)\right\}$$

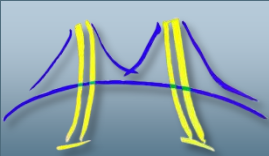
where $\theta_i = (\pi_i, \mu_i, \Sigma_i)$

- Expectation Maximization (EM) Algorithm for training GMMs (find mean, covariance and weights)
 - Multiple parallelization strategies based on problem size and hardware platform characteristics
 - Written in C/CUDA/Cilk+ templates
 - Select best-performing strategy at runtime

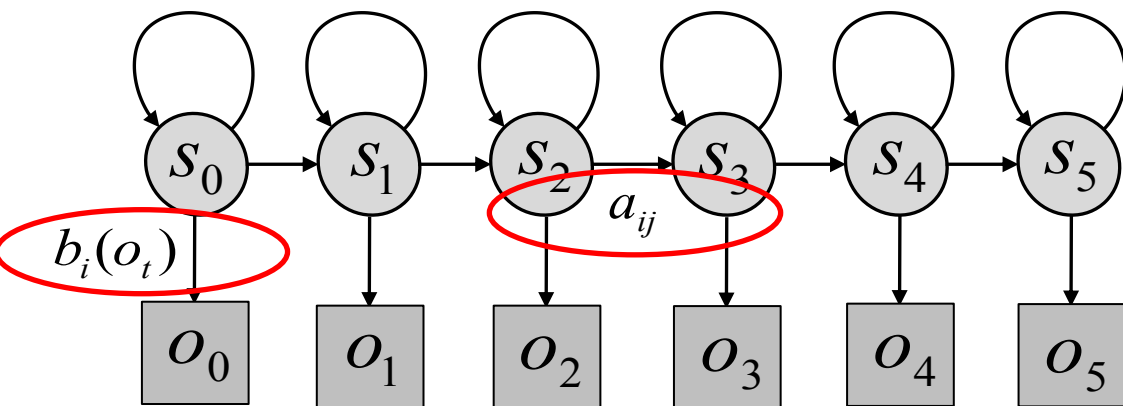


“CUDA-level Performance with Python-level Productivity for Gaussian Mixture Model Applications” Henry Cook, Ekaterina Gonina, Shoaib Kamil, Gerald Friedland, David Patterson, Armando Fox. In Proceedings of the 3rd USENIX conference on Hot topics in parallelism (HotPar’11). USENIX Association, Berkeley, CA, USA.

Example GMM in two dimensions
(Source: www.mathworks.com)



Customizable Component Example: HMM EM Training



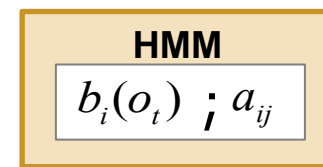
S_i - hidden state i

O_t - observation at time t

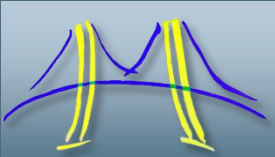
a_{ij} - Transition probability from state i to state j

$b_i(o_t)$ - observation probability of obs t given state i

- Model temporal sequences



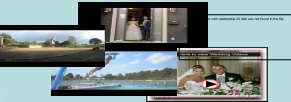


- Training – find parameters A and B given observation sequence O using the Baum-Welsh algorithm (generalized EM)
 - Decoding – find the state sequence S that best matches an observation sequence O (Viterbi algorithm)
- - customizable element

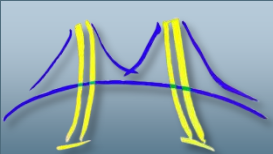


PyCASP Productivity

- Create a **tractable** framework scope by using patterns
- Applications written in **Python**
 - Glue language

Application	Lines of Python Code	Approximated LOC Reduction (vs. C/C++)
Speaker Diarization 	50	60x
Music Recommendation 	500	10-50x
Video Event Detection 	50 + 1	60x + 20x

Specializer	LOC
GMM	1500 C/CUDA 800 Python
Map-Reduce	80 Python



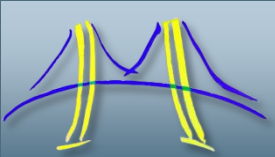
Efficiency

- Speaker Diarization
 - Average faster-than-real-time factor & error rate
 - Averaged across 12 meetings (AMI corpus) [1]



Implementation	Diarization Error Rate	Faster-than-real-time factor
State-of-the-art C++	~22%	1X
PyCASP	24.7%	115X

[1] E. Gonina, G. Friedland, H. Cook and K. Keutzer. "Fast Speaker Diarization Using a High-Level Scripting Language" In Proceedings of IEEE Workshop on Automatic Speech Recognition and Understanding (ASRU), Dec 11-15, 2011, Waikoloa, Hawaii

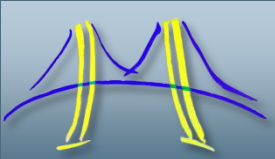


PyCASP Portability

- Speaker Diarization
 - Average faster-than-real-time factor
 - Intel Westmere and two CUDA GPUs
 - Averaged over 12 meetings (AMI corpus)
 - (Augmented Multi-party Interaction corpus)
 - 100 hours of meetings captured using many syn recording devices

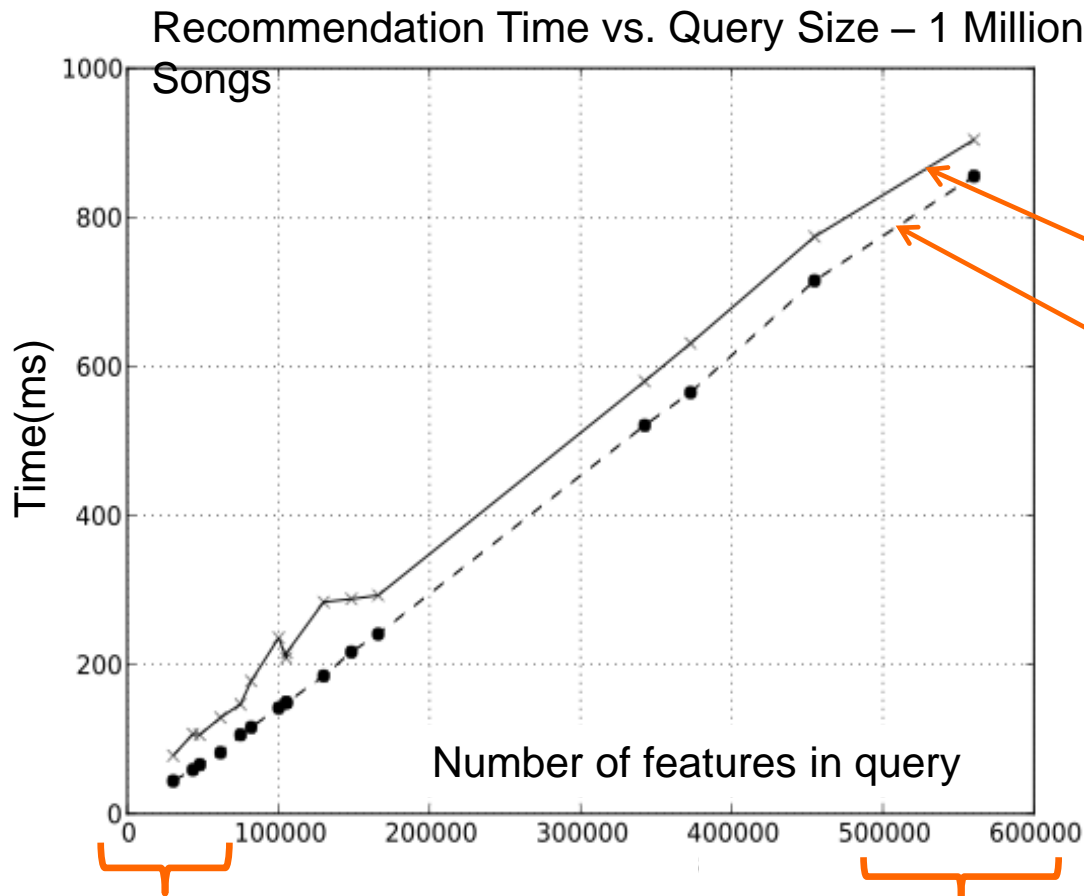
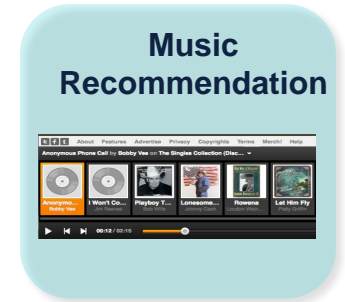
Platform	Faster-than-real-time factor
Intel Westmere	56x
Nvidia GTX285	101x
Nvidia GTX480	115x





PyCASP Scalability

Music Recommendation



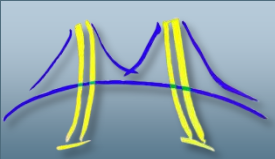
Total recommendation time

Query GMM training time

Under 1 second recommendation time for all queries!

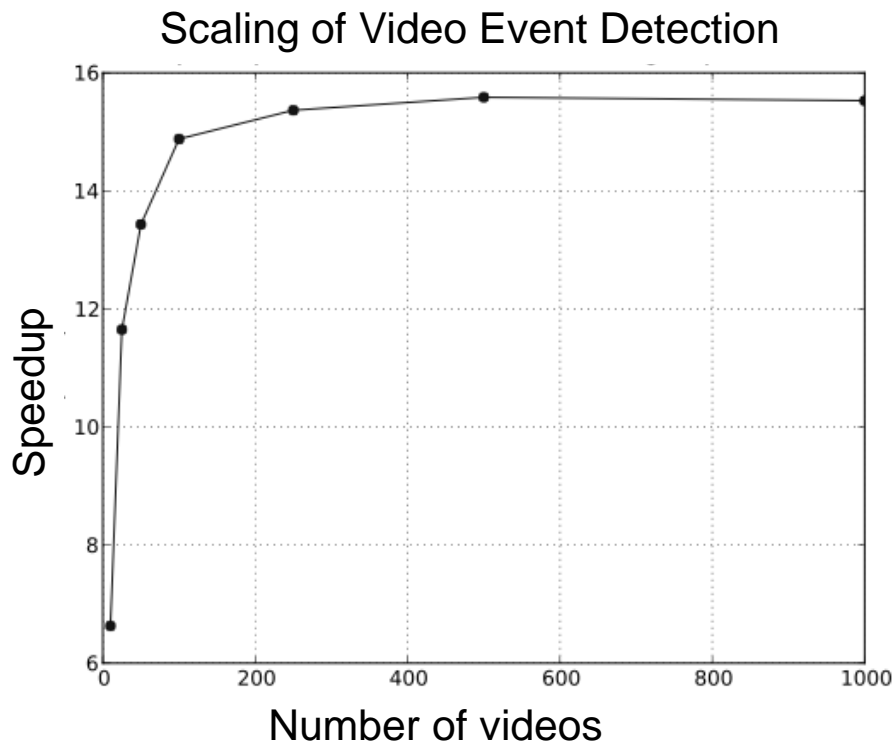
30 songs
"Elton John"

400-500 songs
"Elton John or Eric Clapton or Lady Gaga or Britney Spears"



PyCASP Scalability

- Video Event Detection
- Nearly-optimal scaling on a cluster of GPUs:
 - 15.5x on 16-node cluster for 500 and 1000 videos



Outline

- Solving the parallel programming problem
- Patterns and frameworks
- Making code written by productivity programmers run fast
 - FTDOCK: a simple example of SEJITS
 - Stencil: Complex software transformations with SEJITS
 - A few additional SEJITS specializers
 - PyCASP: From Pattern mining to extensible frameworks
- ➔ • Next Steps and conclusions

Status

- Two independent thrusts at the Berkeley ParLab:
 - Design Patterns: establishing a systematic formalism of parallel software architecture.
 - Composition of Structural and computational parameters define architecture
 - Lower level parallel patterns guide implementation work and inform design of parallel programming environments.
 - SEJITS: software transformation tools to support efficient code written by high level domain experts and supported by efficiency programmers (specializer writers).
- Early integration of two projects have proven successful in early pilot studies.

Next steps

- Patterns:
 - The set of patterns have been defined, but we are still documenting them. We have two large books planned which will document the patterns that underlie ALL of parallel programming (... and yes, we know what a bold and ultimately impossible goal this is).
- SEJITS:
 - The framework has only been used in close collaboration with Berkeley researchers.
 - It needs additional testing by those further removed from Berkeley ... and I expect when this happens “issues” will be found.
 - Creating and debugging specializers is too difficult.
 - To reach its full potential, we need to make specializers reusable ... i.e. to build reusable software frameworks on top of SEJITS ... PyCASP is the first real attempt at this

Conclusion/Summary

- All software must be parallel software. Traditional language based approaches have not solved this problem.
- We need a new approach:
- Support a separation of concerns between *productivity* and *efficiency* programmers
 - Design patterns to “think parallel”.
 - Frameworks to turn patterns into code.
 - SW transformation tools to make the frameworks efficient

