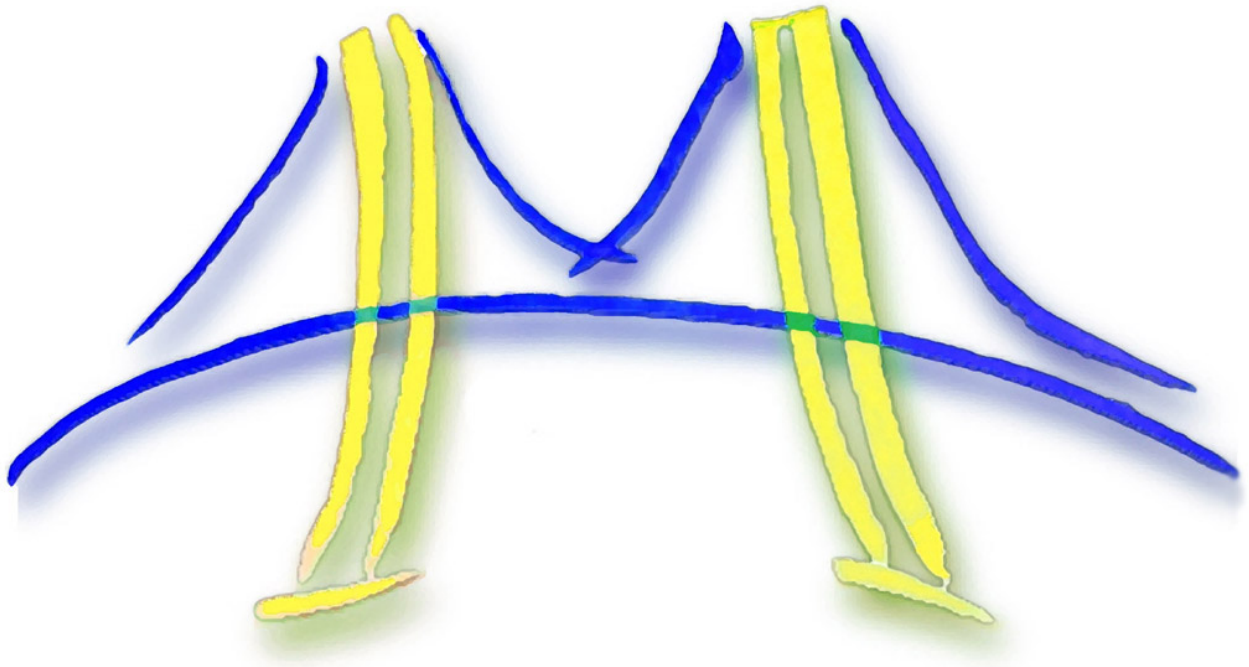


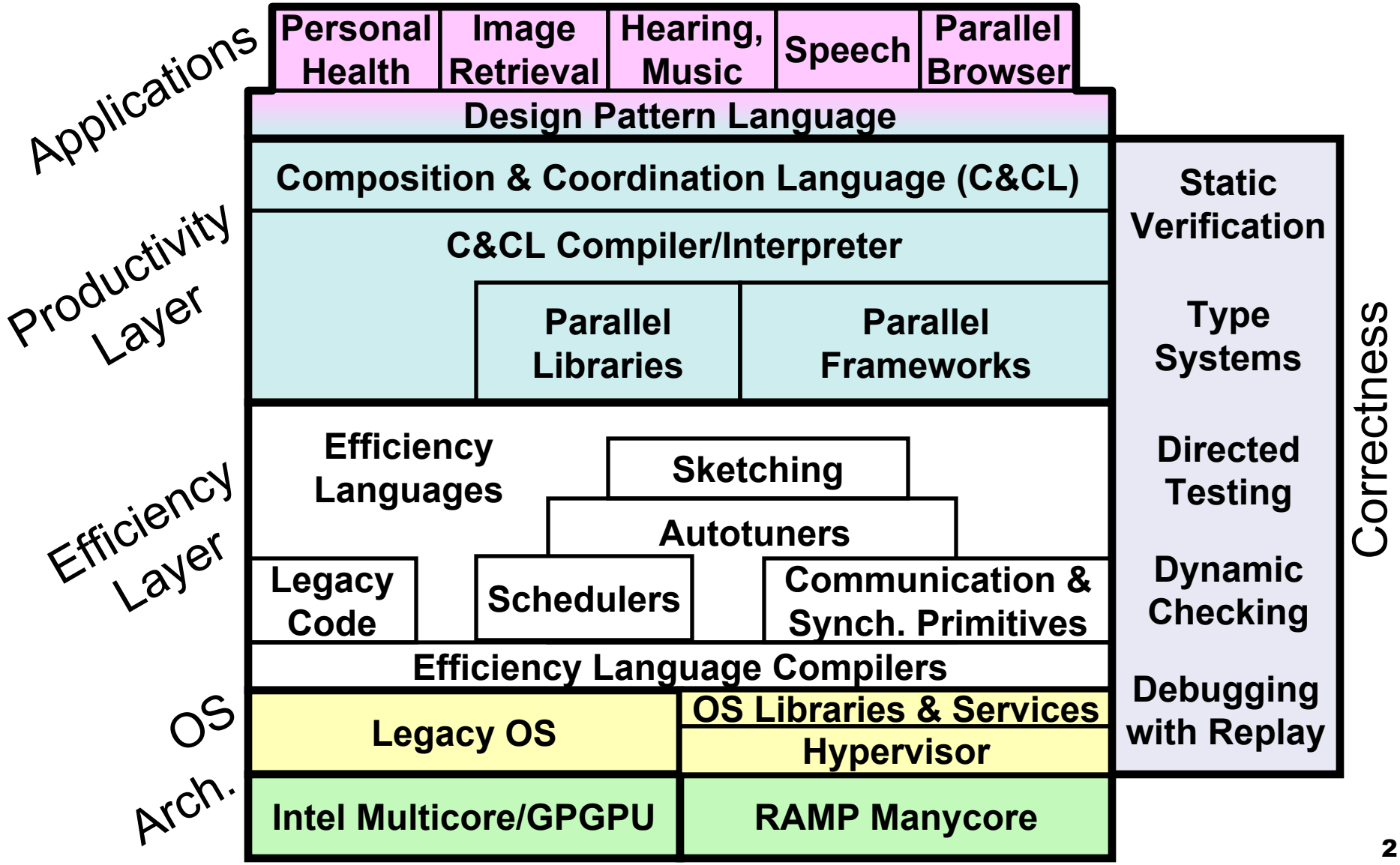
Patterns for Parallel Programming

Tim Mattson (Intel)

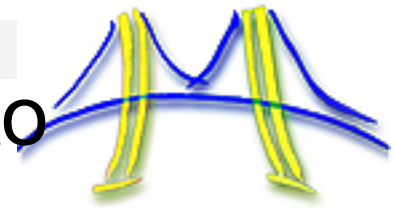
Kurt Keutzer (UCB EECS)

UCB's Par Lab: Research Overview

Easy to write correct software that runs efficiently on manycore



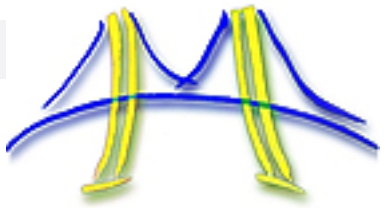
Our goal: use the patterns to guide us to the right frameworks



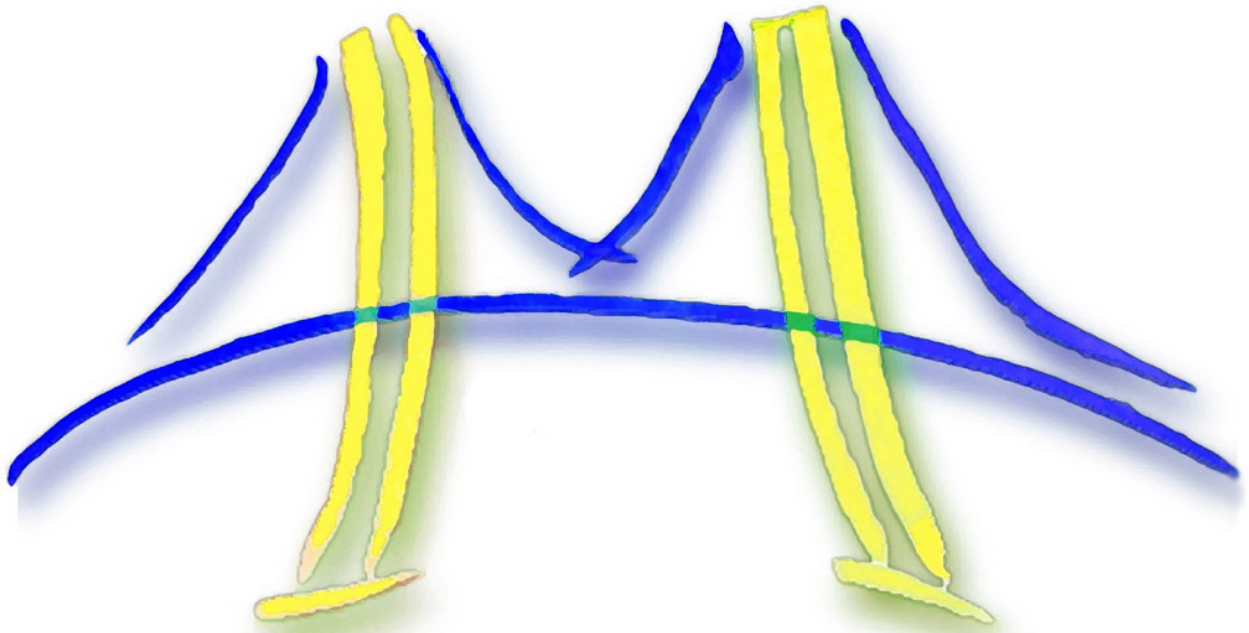
1. **Domain Experts** + **Application patterns & frameworks** → **End-user, application programs**
2. **Domain literate programming gurus (1% of the population).** + **Parallel patterns & programming frameworks** → **Application frameworks**
3. **Parallel programming gurus (1-10% of programmers)** → **Parallel programming frameworks**

The hope is for Domain Experts to create parallel code with little or no understanding of parallel programming.

Leave hardcore "bare metal" efficiency layer programming to the parallel programming experts



- But this is a course on parallel programming languages.
- I have something much more important to talk to you about than patterns.
  - … besides, Kurt Keutzer or I can always come back and talk about patterns any time you want.

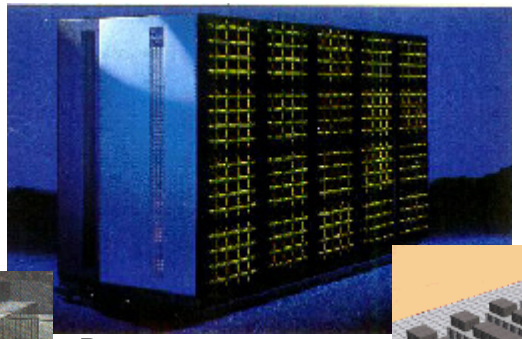


Programmability and the Parallel Programming Problem

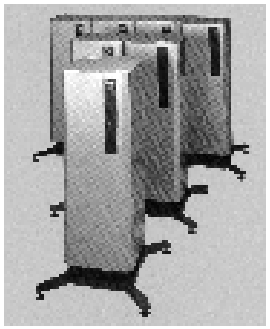
Tim Mattson (Intel)

Computing with 100s or 1000s of cores is not new …. Intel has been doing it for decades.

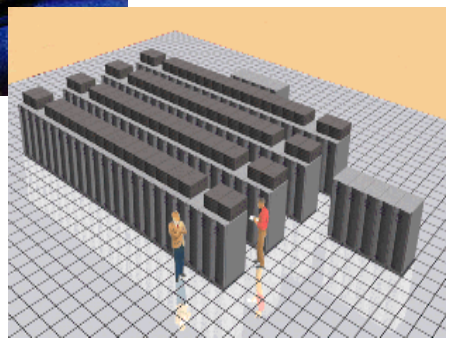
iPSC/860



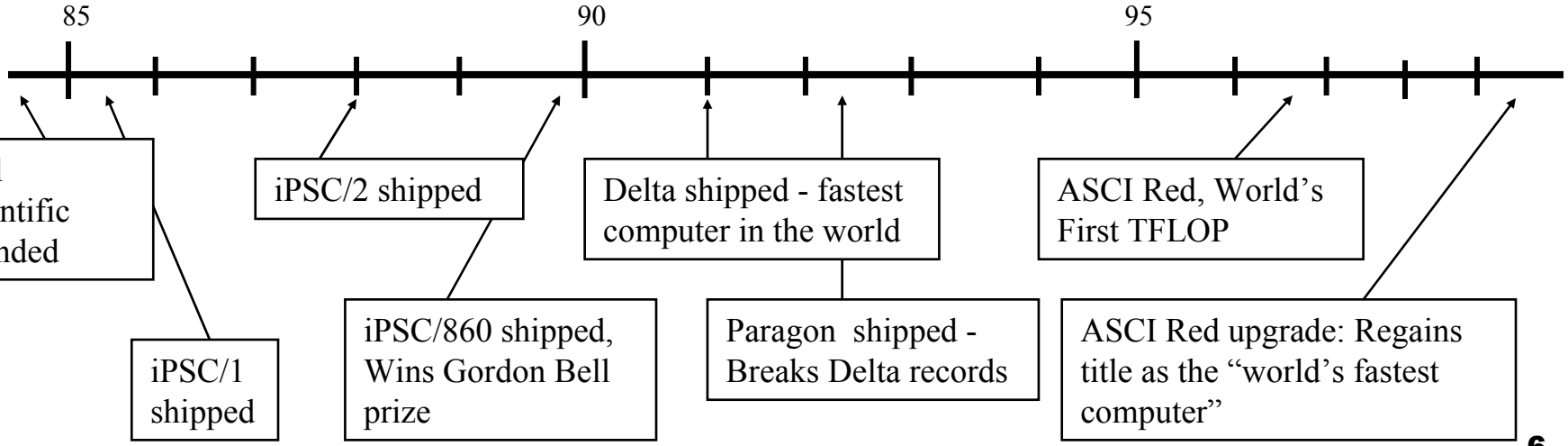
Paragon



iPSC/2



ASCI Option Red

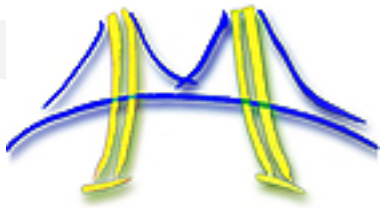


Source: Intel

The result … membership in the “Dead Architecture Society”

Any product names on this slide are the property of their owners.

What went wrong …. Software



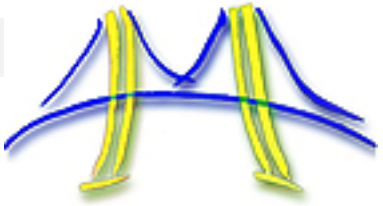
- Parallel systems are useless without parallel software.
  - Can we generate parallel software automatically?
    - NO!!! After years of trying … we know it just doesn't work.
  - Our only hope is to get programmers to create parallel software.
- But after 25+ years of research, we are no closer to solving the parallel programming problem …

  Only a tiny fraction of programmers write parallel code.
- Will the "if you build it they will come" principle apply?
  - Many hope so, but ..

    that implies that people didn't really try hard enough over the last 25 years. Does that really make sense?

All you need is a good Parallel Programming Language, right?

Parallel Programming environments in the 90's



ABCPL
ACE
ACT++
Active messages
Adl
Adsmith
ADDAP
AFAPI
ALWAN
AM
AMDC
AppLeS
Amoeba
ARTS
Athapascan-0b
Aurora
Automap
bb_threads
Blaze
BSP
BlockComm
C*.
"C* in C
C**
CarlOS
Cashmere
C4
CC++
Chu
Charlotte
Charm
Charm++
Cid
Cilk
CM-Fortran
Converse
Code
COOL

CORRELATE
CPS
CRL
CSP
Cthreads
CUMULVS
DAGGER
DAPPLE
Data Parallel C
DC++
DCE++
DDD
DICE.
DIPC
DOLIB
DOME
DOSMOS.
DRL
DSM-Threads
Ease .
ECO
Eiffel
Eilean
Emerald
EPL
Excalibur
Express
Falcon
Filaments
FM
FLASH
The FORCE
Fork
Fortran-M
FX
GA
GAMMA
Glenda

GLU
GUARD
HAsL.
Haskell
HPC++
JAVAR.
HORUS
HPC
IMPACT
ISIS.
JAVAR
JADE
Java RMI
javaPG
JavaSpace
JIDL
Joyce
Khoros
Karma
KOAN/Fortran-S
LAM
Lilac
Linda
JADA
WWWinda
ISETL-Linda
ParLin
Eilean
P4-Linda
Glenda
POSYBL
Objective-Linda
LiPS
Locust
Lparx
Lucid
Maisie
Manifold

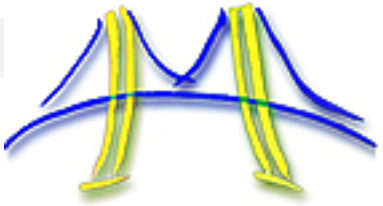
Mentat
Legion
Meta Chaos
Midway
Millipede
CparPar
Mirage
MpC
MOSIX
Modula-P
Modula-2*
Multipol
MPI
MPC++
Munin
Nano-Threads
NESL
NetClasses++
Nexus
Nimrod
NOW
Objective Linda
Occam
Omega
OpenMP
Orca
OOF90
P++
P3L
p4-Linda
Pablo
PADE
PADRE
Panda
Papers
AFAPI.
Para++
Paradigm

Parafrase2
Paralation
Parallel-C++
Parallaxis
ParC
ParLib++
ParLin
Parmacs
Parti
pC
pC++
PCN
PCP:
PH
PEACE
PCU
PET
PETSc
PENNY
Phosphorus
POET.
Polaris
POOMA
POOL-T
PRESTO
P-RIO
Prospero
Proteus
QPC++
PVM
PSI
PSDM
Quake
Quark
Quick Threads
Sage++
SCANDAL
SAM

pC++
SCHEDULE
SciTL
POET
SDDA.
SHMEM
SIMPLE
Sina
SISAL.
distributed smalltalk
SMI.
SONiC
Split-C.
SR
Sthreads
Strand.
SUIF.
Synergy
Telegrphos
SuperPascal
TCGMSG.
Threads.h++.
TreadMarks
TRAPPER
uC++
UNITY
UC
V
ViC*
Visifold V-NUS
VPE
Win32 threads
WinPar
WWWinda
XENOOPS
XPC
Zounds
ZPL

All you need is a good Parallel Programming Language, right?

Parallel Programming environments in the 90's



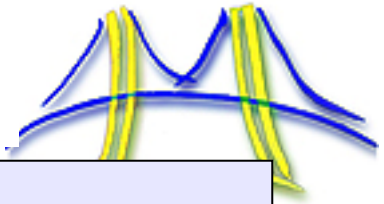
ABCPL
ACE
ACT++
Active messages
Adl
Adsmith
ADDAP
AFAPI
ALWAN
CORRELATE
CPS
CRL
CSP
Cthreads
CUMULVS
DAGGER
DAPPLE
Data Parallel C
GLU
GUARD
HAsL.
Haskell
HPC++
JAVAR.
HORUS
HPC
IMPACT
Mentat
Legion
Meta Chaos
Midway
Millipede
CparPar
Mirage
MpC
MOSIX
Parafrase2
Paralation
Parallel-C++
Parallaxis
ParC
ParLib++
ParLin
Parmacs
Parti
pC++
SCHEDULE
SciTL
POET
SDDA.
SHMEM
SIMPLE
Sina

Before creating any new languages, maybe we should figure out why parallel programming language research been so unproductive.

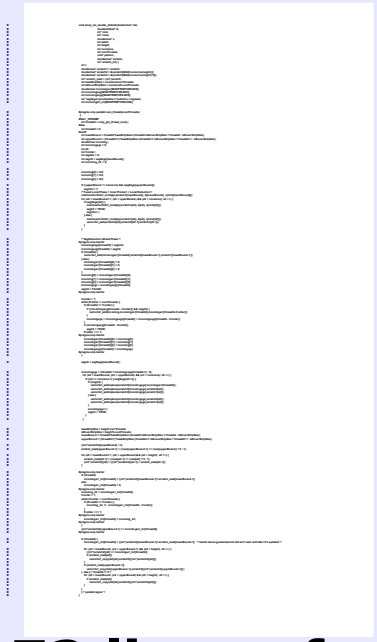
Maybe part of the problem is how we compare programming languages.

C4
CC++
Chu
Charlotte
Charm
Charm++
Cid
Cilk
CM-Fortran
Converse
Code
COOL
Express
Falcon
Filaments
FM
FLASH
The FORCE
Fork
Fortran-M
FX
GA
GAMMA
Glenda
ParLin
Eilean
P4-Linda
Glenda
POSYBL
Objective-Linda
LiPS
Locust
Lparx
Lucid
Maisie
Manifold
OOF90
P++
P3L
p4-Linda
Pablo
PADE
PADRE
Panda
Papers
AFAPI.
Para++
Paradigm
Prospero
Proteus
QPC++
PVM
PSI
PSDM
Quake
Quark
Quick Threads
Sage++
SCANDAL
SAM
UNITY
UC
V
ViC*
Visifold V-NUS
VPE
Win32 threads
WinPar
WWWinda
XENOOPS
XPC
Zounds
ZPL

Comparing programming languages: Ct and OpenMP



OpenMP



172 lines of code

Nested Data Parallel

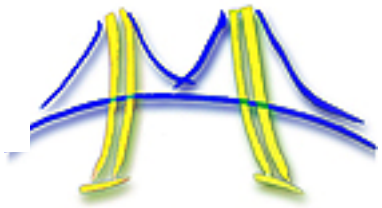
```
VEC<double> sparseMatrixVectorProduct(
  VEC<double> A, VEC<int> rowindex,
  VEC<int> cols, VEC<double> v)
{
  VEC expv = distribute(v,cols);
  VEC product = A*expv;
  return multiReduceSum(product,rowindex);
}
```

6 lines of code
Better performance & scalability

Nested data parallelism enables safe and scalable composition of software modules

- **What conceptually does this comparison tell us? Anything?**
- **Isn't this just marketing-speak disguised as reasoned analysis?**

Win32 API vs OpenMP: Which would you rather use?



```
#include <windows.h>
#define NUM_THREADS 2
HANDLE thread_handles[NUM_THREADS];
CRITICAL_SECTION hUpdateMutex;
static long num_steps = 100000;
double step;
double global_sum = 0.0;

void Pi (void *arg)
{
   int i, start;
   double x, sum = 0.0;


   start = *(int *) arg;
   step = 1.0/(double) num_steps;

   for (i=start;i<= num_steps; i=i+NUM_THREADS){
        x = (i-0.5)*step;
       sum = sum + 4.0/(1.0+x*x);
   }
   EnterCriticalSection(&hUpdateMutex);
   global_sum += sum;
   LeaveCriticalSection(&hUpdateMutex);
}
void main ()
{
   double pi; int i;
   DWORD threadID;
   int threadArg[NUM_THREADS];

   for(i=0; i<NUM_THREADS; i++)   threadArg[i] = i+1;

   InitializeCriticalSection(&hUpdateMutex);

   for (i=0; i<NUM_THREADS; i++){
             thread_handles[i] = CreateThread(0, 0,
                 (LPTHREAD_START_ROUTINE) Pi,
                &threadArg[i], 0, &threadID);
}

   WaitForMultipleObjects(NUM_THREADS,
               thread_handles, TRUE,INFINITE);

   pi = global_sum * step;

   printf(" pi is %f \n",pi);
}
```

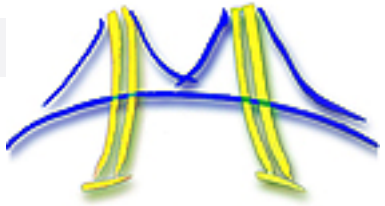
Win32 Threads

```
#include <omp.h>
static long num_steps = 100000;        double step;
void main ()
{              int i;       double x, pi, sum = 0.0;
               step = 1.0/(double) num_steps;
#pragma omp parallel for reduction(+:sum) private(x)
               for (i=1;i<= num_steps; i++){
                              x = (i-0.5)*step;
                              sum = sum + 4.0/(1.0+x*x);
               }
               pi = step * sum;
}
```

OpenMP

I'm just as bad …. Here is a comparison I often make in my OpenMP talks.

Comparing programming languages/APIs



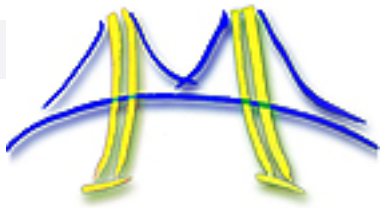
- If we are going to make progress on the parallel programming problem, we must stop taking pot-shots at each other and stop acting like marketers.
- We need to objectively compare parallel programming languages.
  - Categorize the way people work with parallel programming languages and use those categories to evaluate different languages.
  - Define A common terminology for aspects of programmability
  - Create a set of useful programmability benchmarks

If we want to be "good academics" we need to build new languages based on a clear understanding of past work.

We need a "theory of programmability" grounded in productive peer review.

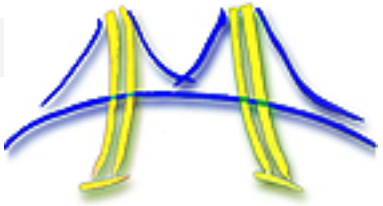
And the first step is to define a human language of programmability

Towards a human language of programmability

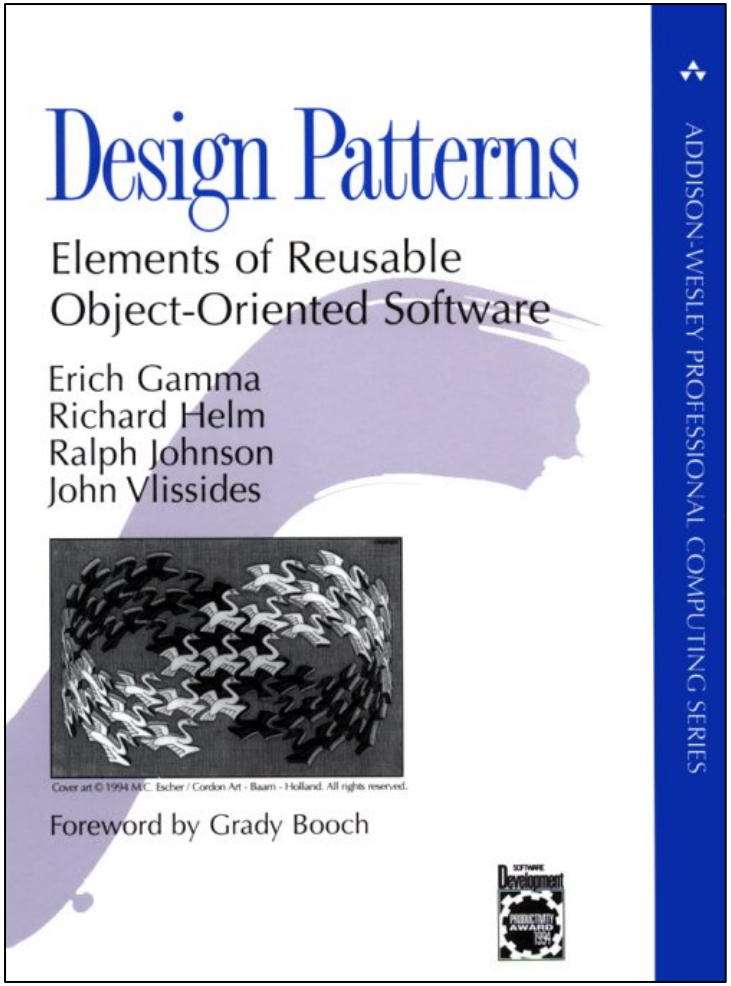


- Algorithms: How do people think about parallel programming?
- A Language of programmability
- Programmability metrics/benchmarks

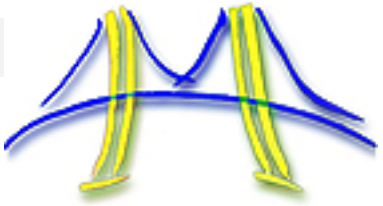
Design patterns were used help people “think OOP”



- A design pattern is:
  - A “solution to a problem in a context”.
  - A structured description of high quality solutions to recurring problems
  - A quest to encode expertise so all designers can capture that “quality without a name” that distinguishes truly excellent designs
- A pattern language is:
  - A structured catalog of design patterns that supports design activities as “webs of connected design patterns”.



Design Patterns:
A silly example



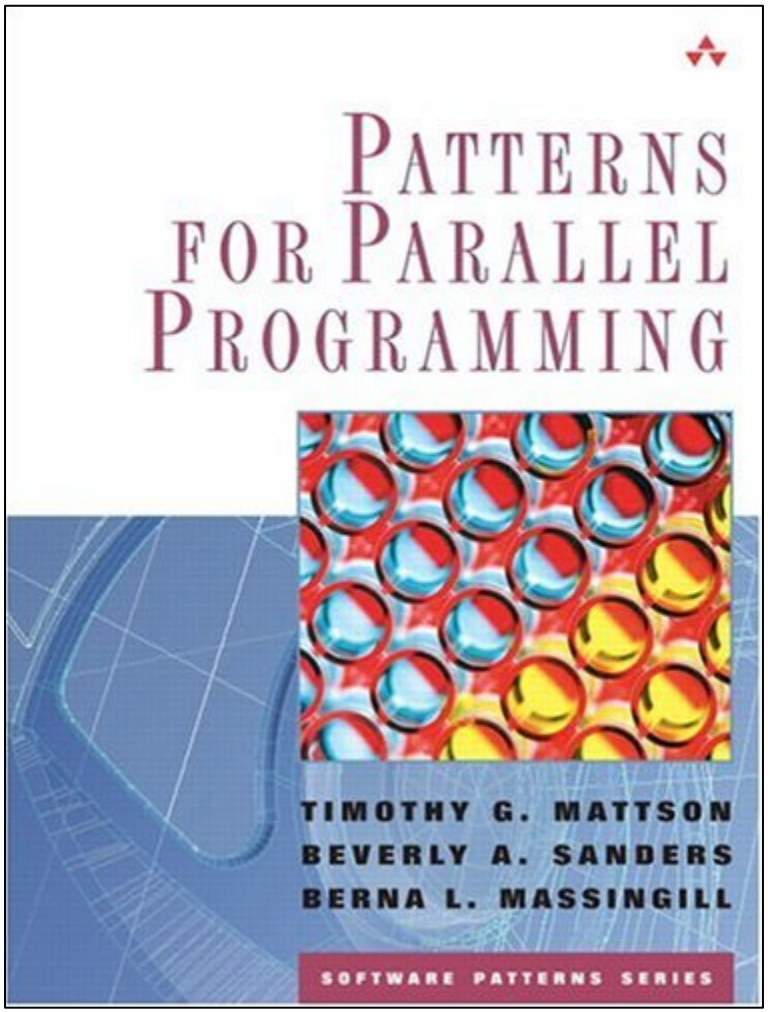
- Name: Money Pipeline

- Context: **You want to get rich and all you have to work with is a C.S. degree and programming skills.** How can you use software to get rich?

- Forces: **The solution must resolve the forces:**
  - **It must give the buyer something they believe they need.**
  - **It can't be too good, or people won't need to buy upgrades.**
  - **Every good idea is worth stealing -- anticipate competition.**

- Solution**: Construct a money pipeline**
  - **Create SW with enough functionality to do something useful most of the time. This will draw buyers into your money pipeline.**
  - **Promise new features to thwart competitors.**
  - **Use bug-fixes and a slow trickle of new features to extract money as you move buyers along the pipeline.**

Let's use Design patterns to help people "think parallel"

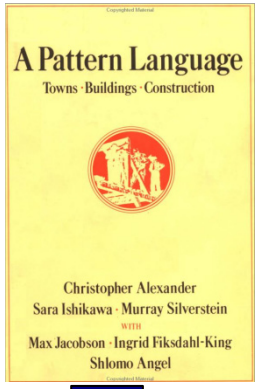


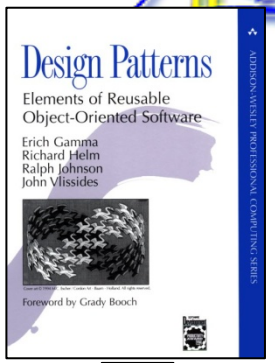
A pattern language for parallel algorithm design with examples in MPI, OpenMP and Java.

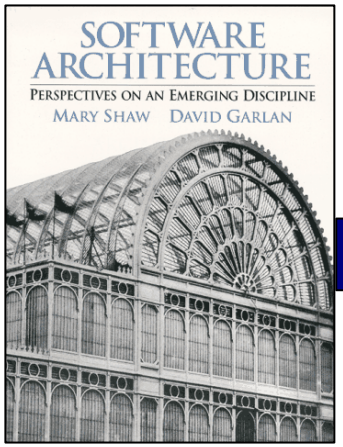
This is our hypothesis for how programmers think about parallel programming.

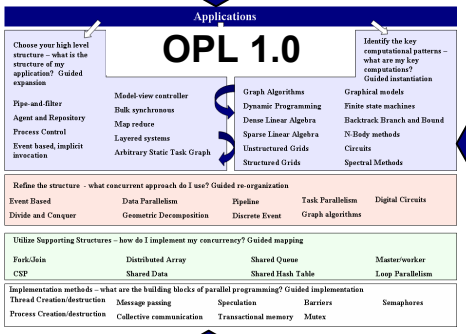
Now available at a bookstore near you!

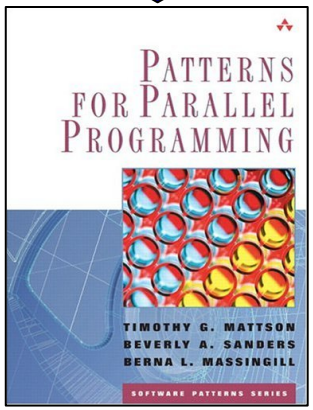
CS294 2008 – putting it all together … Our Pattern Language (OPL 1.0)



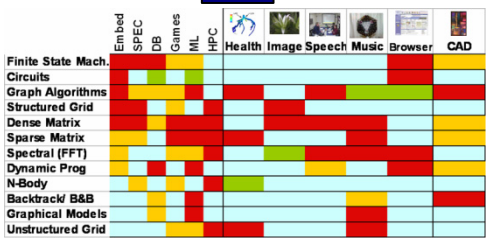






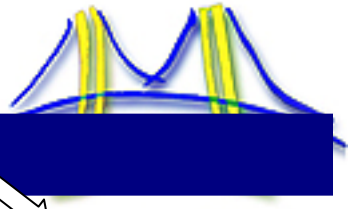


PPPL: Parallel Programming Pattern language



13 dwarves

CS294-2009: OPL Version 2.0



Applications

Productivity Layer

SW Architecture

Choose your high level structure – what is the structure of my application? Guided expansion

- Pipe-and-filter
- Agent and Repository
- Process Control
- Event based, implicit invocation
- Model-view controller
- Bulk iterative
- Map reduce
- Layered systems
- Arbitrary Static Task Graph

Computation

Identify the key computational patterns – what are my key computations? Guided instantiation

- Graph Algorithms
- Dynamic Programming
- Dense Linear Algebra
- Sparse Linear Algebra
- Unstructured Grids
- Structured Grids
- Finite State Machines
- Backtrack Branch and Bound
- N-Body methods
- Circuits
- Spectral Methods
- Monte Carlo

Algorithm

Parallel algorithm strategy - what high level strategy do I use for my parallel algorithm? Guided re-organization

- Task Parallelism
- Recursive splitting
- Data Parallelism
- Pipeline
- Decomposition
- Discrete Event
- Digital Circuits
- Graph Partitioning

Efficiency Layer

Source Code

Implementation strategy – how do I express t… programming

- SPMD
- Strict-data-par
- Fork/Join
- Actors
- Master/worker
- Loop Parallelism
- BSP
- Shared Queue
- Shared Hash Table
- Distributed Array
- Shared Data

Prog struc

Data struc

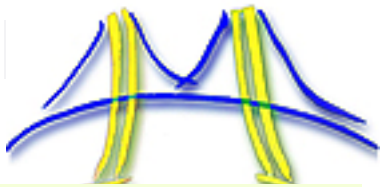
Execution

Concurrent execution – how do we support t… thm? Guided HW mapping.

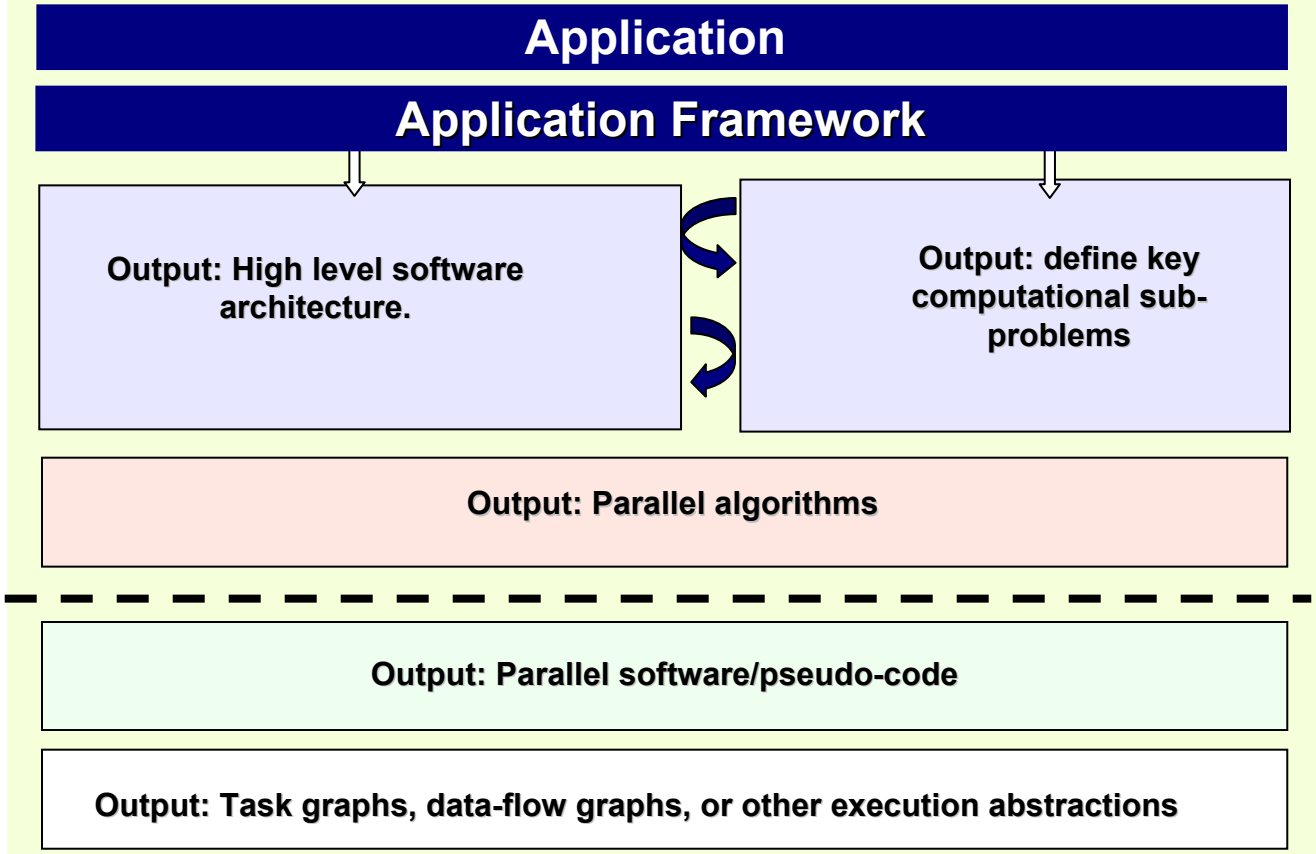
- CSP
- Task Graph
- SIMD
- Thread Pool
- Speculation
- data flow
- Adv. program counters
- Msg pass
- coll comm
- mutual exclusion
- Pt-2-pt sync
- coll sync
- Trans. Mem

Coordination

OPL 2.0: outputs and targets



What is output at each layer of the pattern language?



Primary Audience

End user

Application programmer

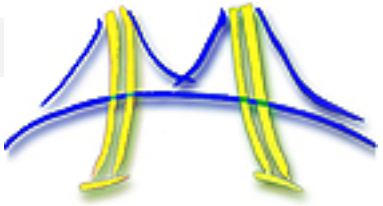
App framework developer

Programming Framework developer

Par framework developer

HW or system SW architects

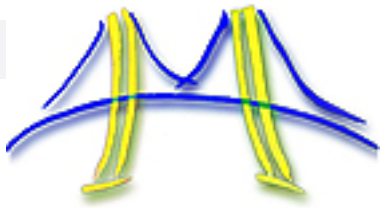
Status of OPL



- Conceptually we have made huge progress
  - Our overall structure is sound.
  - A complete list of patterns with a breif description are in the appendix to this lecture.
- Current draft of patterns are available on line at
  - http://parlab.eecs.berkeley.edu/wiki/patterns/patterns
- Structural and Computational patterns:
  - Many have text and are in decent shape:
    - Often need better examples.
    - Need figures and more detailed examples.
- Lower three layers:
  - Many can be quickly generated based on content from PPPL
  - New patterns reflect more detailed analysis since PPPL was finished.
  - Need focus on the needs of efficiency-layer programmers and SW/HW architects.

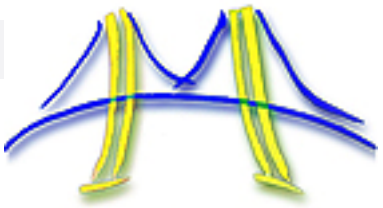
We will finish OPL during CS294'2009!

OPL Example: the Linear Algebra dwarf



- **Reservoir modeling**
- **Quantum Chemistry**
- **Image processing**
- **Least squares and other statistics calculations**
- **Seismic Signal processing**
- **Financial analytics**
- **Business analytics (operations research)**
- **Graphics**
- **… and more areas than I could hope to list**

Linear Algebra applications



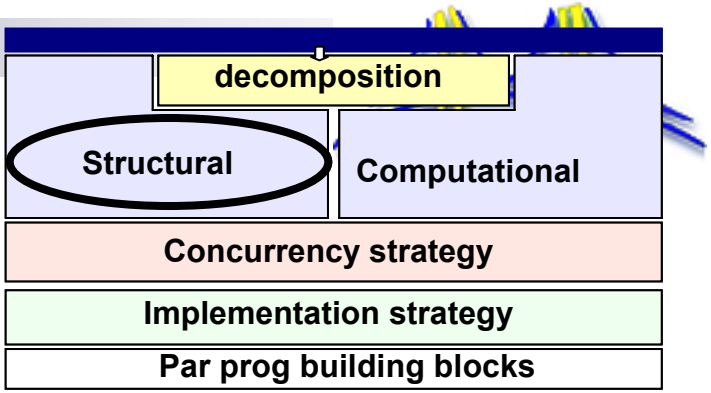
Linear Algebra Applications follow a similar pattern

- Loop over appropriate parameter:
  - Build matrices
  - Operate on matrices
  - Update result
- Continue till termination condition

The program is organized around the matrices (dense or sparse) … in scientific computing with a *monolithic architecture* … these problems use the geometric decomposition and distributed array patterns.

Ideally the library routines used to operate on the matrices dictate the structure of the matrices … the programmer needs to understand these so he or she can build the matrices with the required structure.

What if we used a more rigorous architectural approach?



- **Pipe-and-filter**
- **Agent and Repository**
- **Process Control**
- **Event based, implicit invocation**
- **Model-view-controller**
- **Bulk iterative**
- **Map reduce**
- **Layered systems**
- **Arbitrary Static Task Graph**

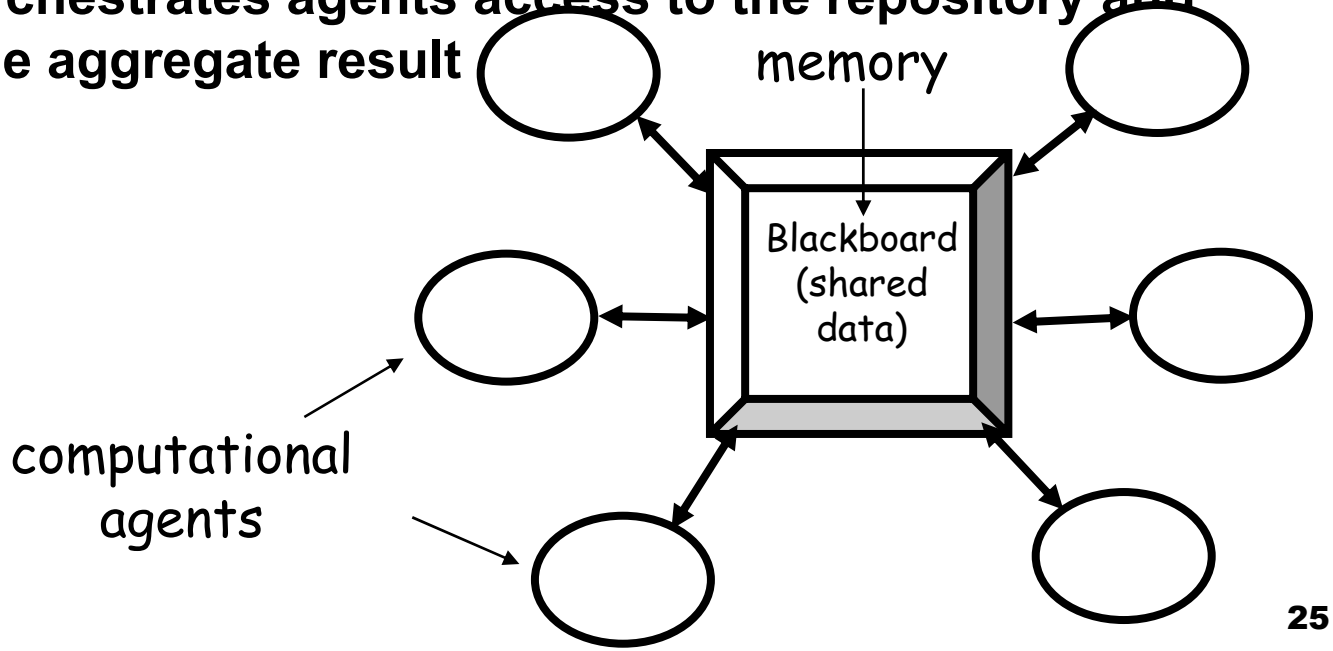
Which Architectural Style is appropriate for linear algebra applications?

Agent & Repository (Blackboard)

- **Data-centered repository styles: blackboard or database**

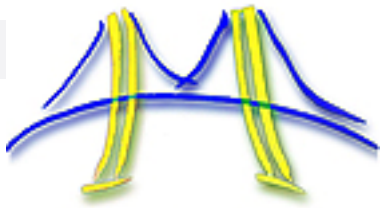
  Key elements:

  - **Repository (Blackboard) of the resulting creation that is shared by all agents**
  - **Agents: intelligent agents that will act on blackboard**
  - **Controller: orchestrates agents access to the repository and creation of the aggregate result**



25

The controller

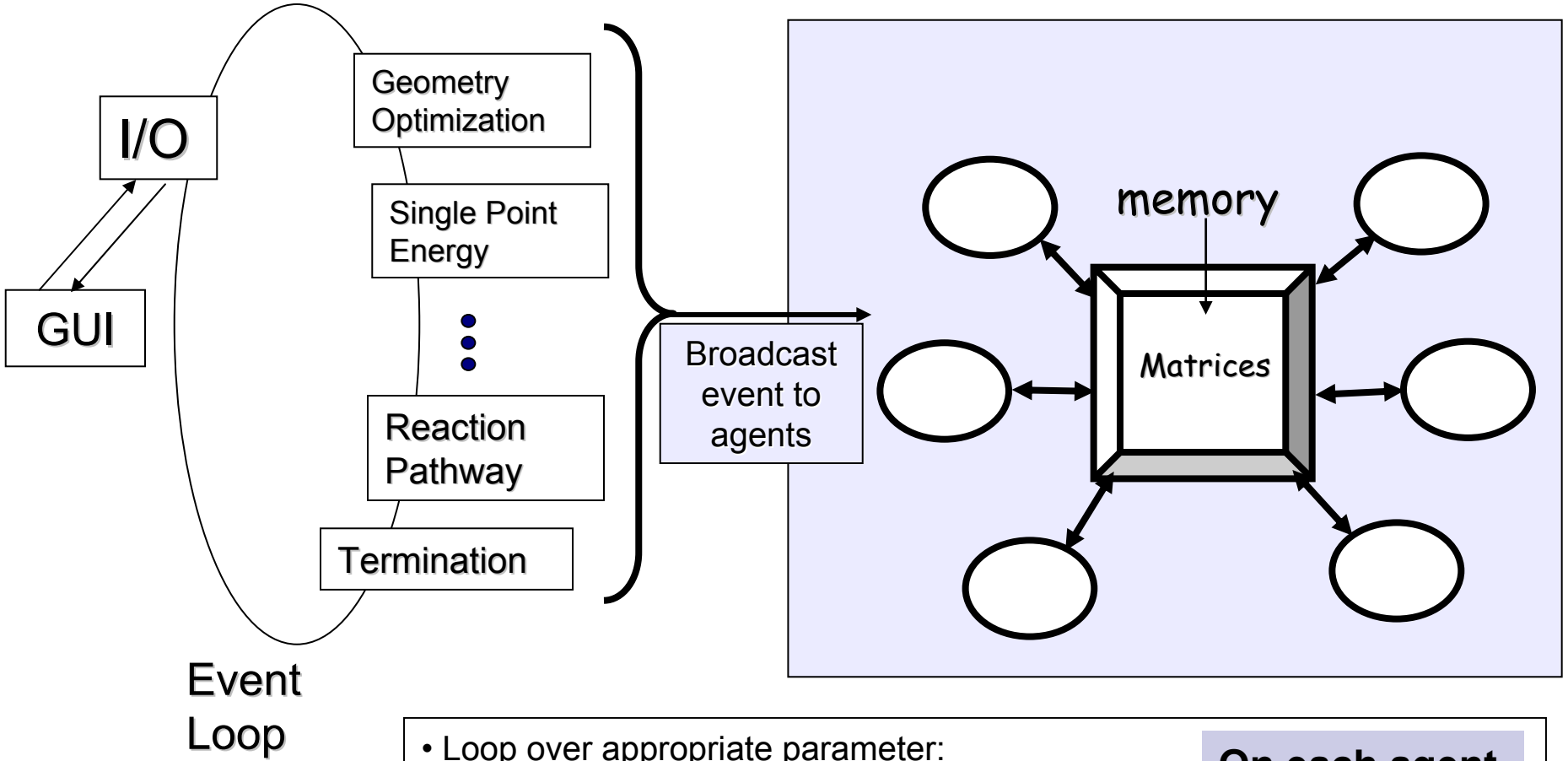


- **The controller for the agents can use a variety of approaches:**
  - **Master worker pattern when the updates vary widely and unpredictably.**
  - **Geometric decomposition with a static mapping onto agents**

- **The Blackboard is a "logical structure":**
  - **Even on a shared memory machine, we organize it into blocks to optimize data locality … hence treat it quite similar to distributed memory environments.**
  - **The black board may be distributed among the agents:**
    - **Use an owner-computes filter … i.e. equally spread out the black board, and each agent updates the blocks it "owns".**

26

Case Study: Quantum Chemistry

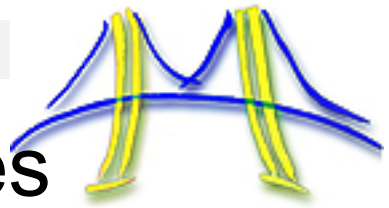


- Loop over appropriate parameter:
  - Build matrix
  - Diagonalize matrix
  - Compute physical quantities from Eigenvalues/Eigenvectors
- Exit on termination condition

On each agent

27

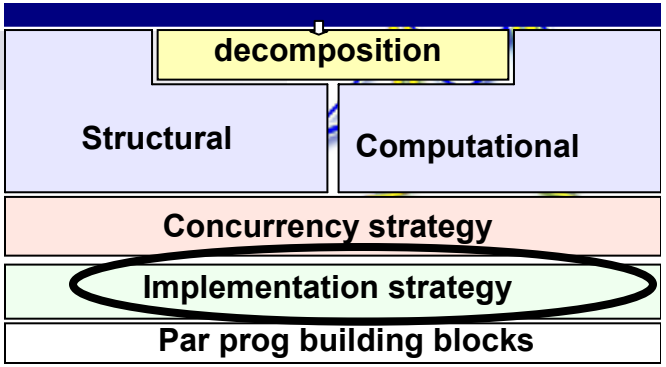
Managing concurrent access to matrices



- **Matrices are stored according to the needs of the libraries and the target platforms.**
- **How do we abstract these complexities from the programmer?**

28

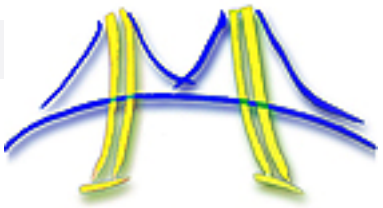
Distributed Array Pattern



- **Name: Distributed array**
- **Problem:**
  - **Arrays often need to be partitioned between multiple UEs. How can we do this so the resulting program is both readable and efficient?**
- **Forces**
  - **Large number of small blocks organized to balance load.**
  - **Able to specialize organization to different platforms/problems.**
  - **Understandable indexing to make programming easier.**
- **Solution:**
  - **Express algorithm in blocks**
  - **Abstract indexing inside mapping functions … programmer works in an index space natural to the domain, functions map into distribution needed for efficient execution.**
  - **The text of the pattern defines some of these common mapping functions (which can get quite confusing … and in the literature are usually left as "an exercise for the reader").**

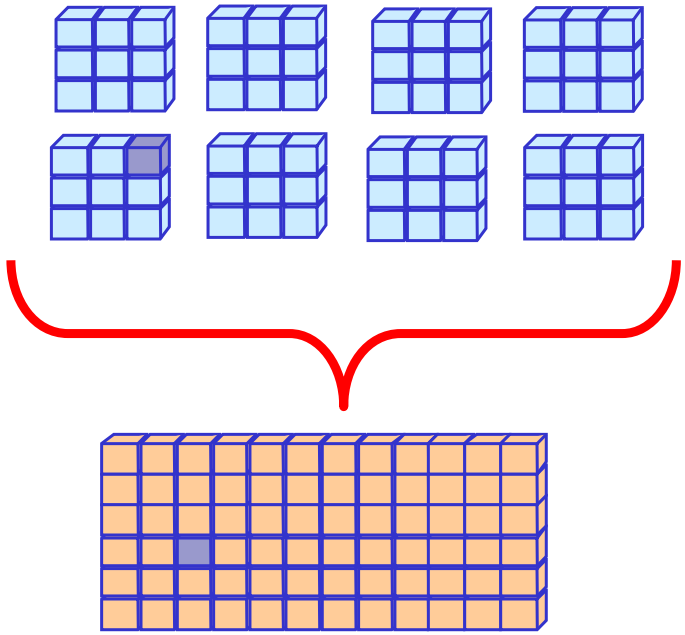
29

Global Arrays



Distributed dense arrays that can be accessed through a shared data-like style

Physically distributed data



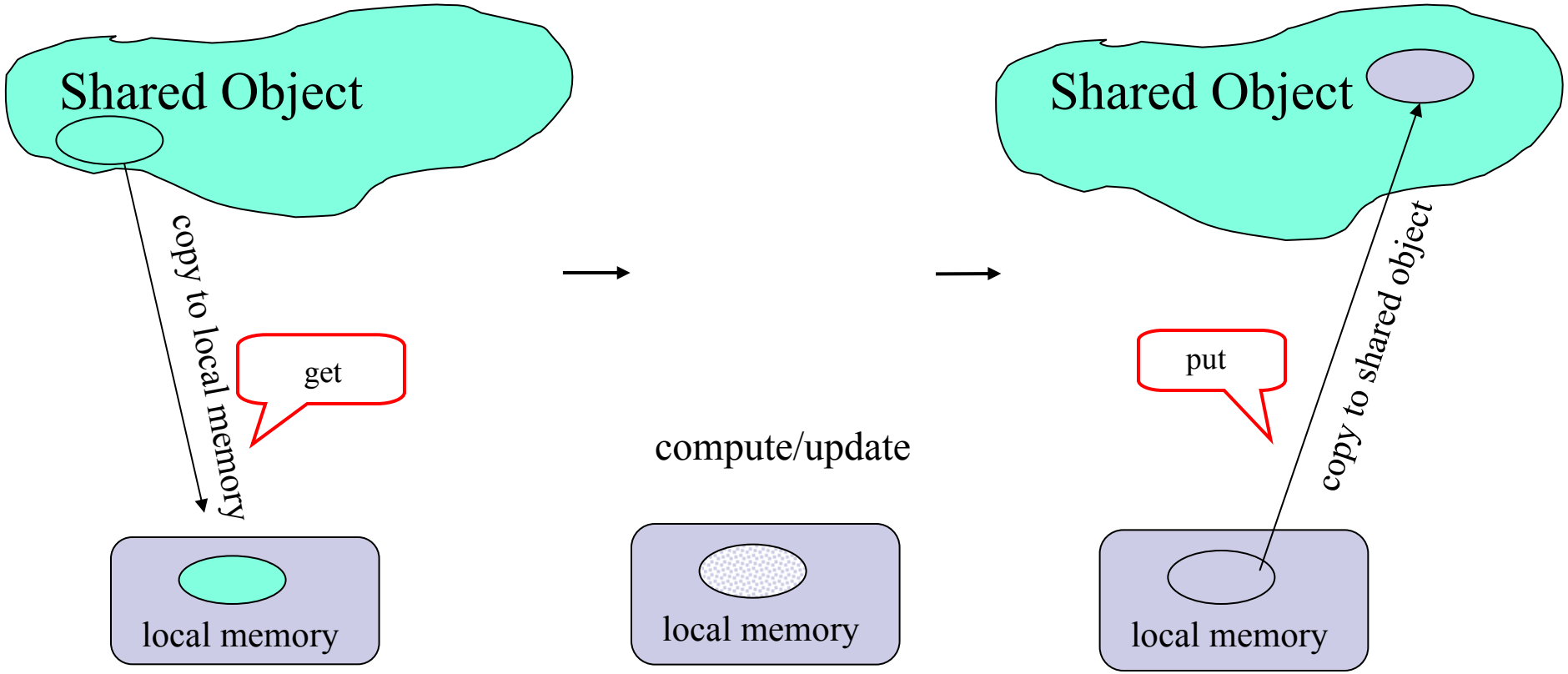
Global Address Space

single, shared data structure/ global indexing

e.g., access A(4,3) rather than buf(7) on task 2

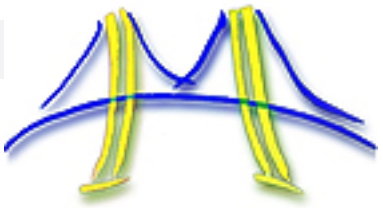
30

Global Array Model of Computations



31

Use of the patterns



- Patterns help us describe expert solutions to parallel programming
- They give us a language to describe the architecture of parallel software.
- They provide a roadmap to the frameworks we need to support general purpose programmers.

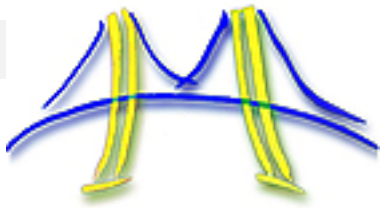
… And they give us a way to systematically map programming languages onto of parallel algorithms … thereby comparing their range of suitability

Example: Using patterns to discuss programmability

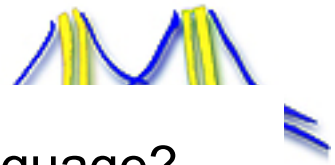
Pattern	OpenMP	OpenCL	MPI
Task Parallelism			
Recursive splitting	with 3.0		
Pipeline			
Geometric Decomposition			
Discrete Event			
SPMD			
SIMD			
Fork/Join			with MPI 2.0
Actors			
BSP			
Loop Parallelism			
Master/worker			

Good mapping
Weak mapping
Not suitable

Towards a human language of programmability



- Algorithms: what are the standard algorithms parallel programming experts take for granted?
- A Language of programmability
- Programmability metrics/benchmarks



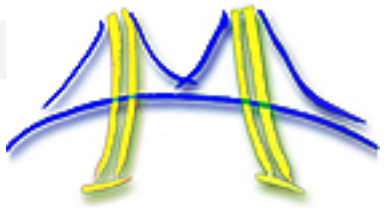
How do we describe human interaction with the programming language?

- Thomas Green is a well known researcher in the “psychology of programming” community.
- After years of work on formal cognitive models with little to show for it, he concluded:

  The way forward is not to make strong, simple claims about how cognitive process work. The way forward is to study the details of how notations convey information.

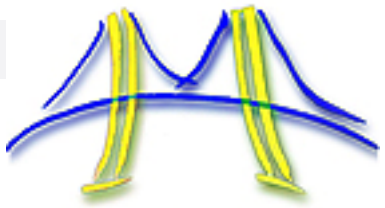
- He proposed a set of “Cognitive Dimensions” as a “discussion framework” for information notations.
- Cognitive Dimensions in action
  - First used to analyze visual programming languages.
  - Since then, its used to analyze a number of information appliances.
  - Used by Steven Clarke of Microsoft to analyze C#

Cognitive dimensions



- There are around 13 of them. The 10 most important to parallel programming are:
  - Viscosity: how hard is it to introduce small changes.
  - Hidden Dependencies: does a change in one part of a program cause other parts to change in ways not overtly apparent in the program text?
  - Error Proneness: How easy is it to make mistakes?
  - Progressive Evaluation: can you check a program while incomplete? Can parallelism be added incrementally?
  - Abstraction Gradient: how much is required? How much abstraction is possible
  - Closeness of mapping: how well does the language map onto the problem domain?
  - Premature commitment: Does the notation constrain the order you do things? AKA imposed look ahead.
  - Consistency: Similar semantics implied by similar syntax. Can you guess one part of the notation given other parts?
  - Hard mental operations: does the notation lead you to complex combinations of primitive operations
  - Terseness: how succinct is the language?
- For parallel programming, I'll add two more
  - HW visibility: is a useful cost model exposed to the programmer?
  - Portability: does the notation assume constraints on the hardware?

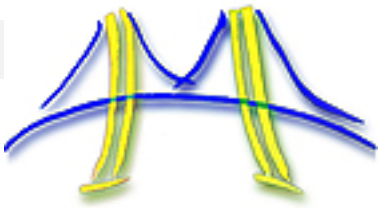
Cognitive Dimensions: viscosity



- How easy is it to introduce small changes to an existing parallel program?
- Low viscosity example: To change how loop iterations are scheduled in OpenMP, just change a single clause

```
#pragma omp parallel for reduction(+:sum) private(x) schedule(dyanmic)
        for (i=1;i<= num_steps; i++){
                x = (i-0.5)*step;
                sum = sum + 4.0/(1.0+x*x);
        }
        pi = step *h sum;
```

Cognitive Dimensions: viscosity

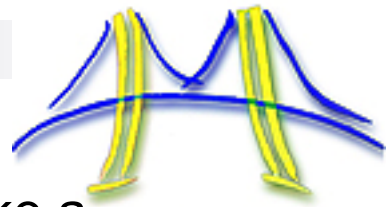


- How easy is it to introduce small changes to an existing parallel program?
- High viscosity example: To change how loop iterations are scheduled in Win32 threads, change multiple lines of code

```
step = 1.0/(double) num_steps;
for (i=start;i<= num_steps; i=i+NUM_THREADS){
      x = (i-0.5)*step;
     sum = sum + 4.0/(1.0+x*x);
}
EnterCriticalSection(&hUpdateMutex);
global_sum += sum;
LeaveCriticalSection(&hUpdateMutex);
}
```

```
step = 1.0/(double) num_steps;
Initiallize_task_queue(num_steps);
while(!done){
     I = get_next()
     x = (i-0.5)*step;
     sum = sum + 4.0/(1.0+x*x);
     done = termination_test(i);
}
EnterCriticalSection(&hUpdateMutex);
global_sum += sum;
LeaveCriticalSection(&hUpdateMutex);
}
```

Cognitive Dimensions: Error Proneness

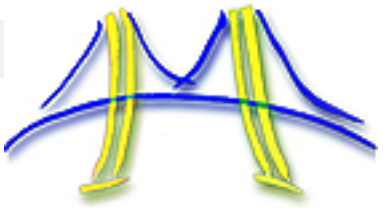


- Does the notation make it easy for the programmer to make a cognitive slip and introduce an error?
- Shared address space languages (such as OpenMP) are very error prone … make it easy to introduce races by unintended sharing of data.

```
#include <omp.h>
static long num_steps = 100000;        double step;
void main ()
{              int i;       double x, pi, sum = 0.0;
               step = 1.0/(double) num_steps;
#pragma omp parallel for reduction(+:sum)
               for (i=1;i<= num_steps; i++){
                              x = (i-0.5)*step;
                              sum = sum + 4.0/(1.0+x*x);
               }
               pi = step * sum;
}
```

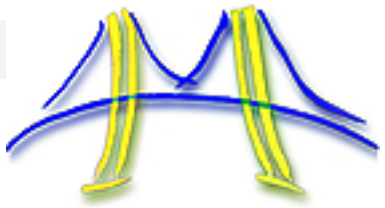
By forgetting a simple "private(x)" clause, I've introduced a race condition

Cognitive Dimensions: Abstraction depth



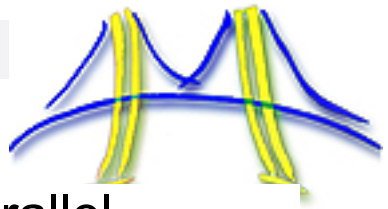
- Does the notation give the programmer the ability to build his/her own abstractions?
  - Abstraction rich parallel languages:
    - TBB (thread building blocks); generic programming and standard template libraries meets parallel programming.
      - Build abstract containers, and introduce parallelism by using concurrent containers
      - Change how concurrency is executed by changing containers.
  - Abstraction poor languages:
    - OpenMP: Programmer has very little support from the notation for building abstractions. Very little abstraction is possible.
- Abstraction barriers: how much abstraction is required just to get started.
  - TBB has a very high abstraction barrier.

Cognitive Dimensions: Hidden dependencies

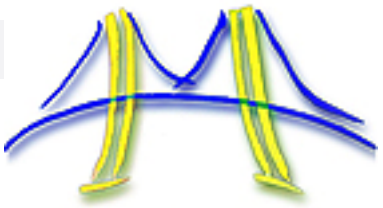


- Hidden Dependencies: make a change in one location and effects seen elsewhere … in ways not apparent in the program text.
- Abstraction rich languages increase problems from hidden dependencies:
  - Change member function in a base class, and an object of a derived class changes its behavior.

Cognitive dimensions



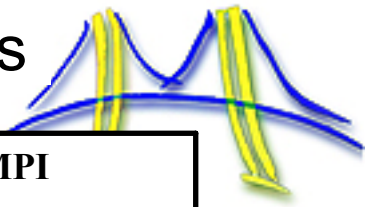
- There are around 13 of them. The 10 most important to parallel programming are:
  - Viscosity: how hard is it to introduce small changes.
  - Hidden Dependencies: does a change in one part of a program cause other parts to change in ways not overtly apparent in the program text?
  - Error Proneness: How easy is it to make mistakes?
  - Progressive Evaluation: can you check a program while incomplete? Can parallelism be added incrementally?
  - Abstraction Gradient: how much is required? How much abstraction is possible
  - Closeness of mapping: how well does the language map onto the problem domain?
  - Premature commitment: Does the notation constrain the order you do things? AKA imposed look ahead.
  - Consistency: Similar semantics implied by similar syntax. Can you guess one part of the notation given other parts?
  - Hard mental operations: does the notation lead you to complex combinations of primitive operations
  - Terseness: how succinct is the language?
- For parallel programming, I'll add two more
  - HW visibility: is a useful cost model exposed to the programmer?
  - Portability: does the notation assume constraints on the hardware?



Cognitive dimensions of OpenMP and MPI

Cognitive Dimension	OpenMP	MPI
Viscosity	Low viscosity: pragma have minimal semantic weight … easy to move around	High viscosity: sends/recvs paired, data structures explicitly decomposed
Error Proneness	High: shared address space = hard to detect race conditions	Medium-low: disjoint memory makes races rare and deadlock easy to find. Long argument lists are a problem.
HW visibility	Poor: An abstract API that hides hardware	Fair to Good: hardware model implied but usually visible.
Progressive evaluation	High: Semantically neutral constructs allow incremental parallelism.	Low: rip prog. apart to expose distributed data and tasks, and test once you put things back together.
Portability	Poor: requires systems with shared address spaces	Great: assumes minimal system support

Comparing prog. languages for data parallel algorithms

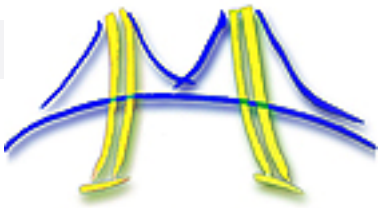


Cognitive Dimension	OpenMP	PeakStream	Ct	RapidMind	CUDA	MPI
Viscosity	Low	Medium-high	Medium-high	Medium … medium-high	Medium	High
Error Proneness	High	Medium	Medium	Medium-low	High	Medium-low
HW visibility	Poor	Poor	Poor	Medium	High	High
Progressive evaluation	High	Poor	Poor	Poor	Poor-medium	Poor
Portability	Poor-good	Good	Good	Good	Poor	Good
Hard Mental Operations	Low	High	High	Medium	Medium-high	Medium-high
Closeness of Mapping / Generality	Medium	Poor	Poor-low (today)	Low-medium	Low	High
Abstraction	Low-intermediate	Very high	Very high	Intermediate	Low	Low

Green: good for programmers. red: bad for programmers. yellow-to-orange: in between

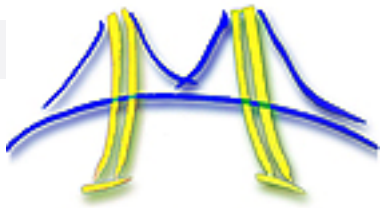
Source: Charles Congdon of Intel

Towards a human language of programmability



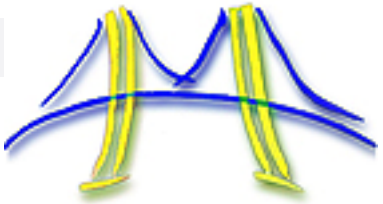
- Algorithms: what are the standard algorithms parallel programming experts take for granted?
- A Language of programmability
- Programmability metrics/benchmarks

Metrics of programmability



- So we have a consistent terminology for programmability, now we need metrics.
- We have benchmarks for performance, how about for programmability?
  - HPCS took a stab at the problem with their synthetic compact applications, but they didn't take it far enough.
  - The old Salishan problems were great, but need updating.
    - Hamming's Problem (compute ordered sets of prime numbers): recursive streams with producer/consumer parallelism and recursive tasks.
    - The Paraffin Problem: nested loop-level parallelism over complex tree structures
    - The doctor's office: asynchronous processes with circular dependencies
    - Skyline matrix solver: solving structure sparse problems.

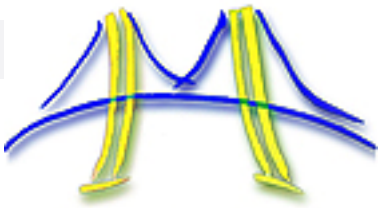
A programmability benchmark suite



- Let's define a set of programmability benchmarks.
  - The key is coverage … we must cover the major classes of applications and parallel algorithms.
- The programmability benchmarks must be:
  - Provided as serial code in a common high level language.
  - Contain lots of concurrency; accessible but not too easily.
  - Have a "right" answer that can be easily verified.
  - Short … you want users to focus on the parallel notation, not the program itself.

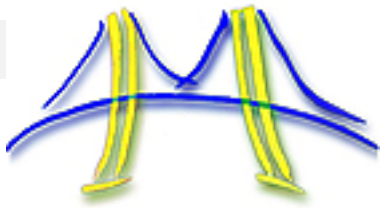
- Maybe we could use an "interesting" subset of The thirteen dwarves:
  - **Dense Linear Alg.**
  - **Sparse Lin. Alg.**
  - **Spectral methods**
  - **N-body methods**
  - **Structured grids**
  - **Unstruc. grids**
  - **MapReduce**
  - **Combinatorial logic**
  - **Graph traversal**
  - **Dynamic prog**
  - **Back-track/branch and bound**
  - **Graphical methods**
  - **Finite state mach.**

Conclusion



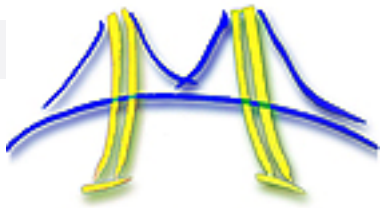
- People have been creating parallel programming languages for many years … and frankly we don't have much to show for all this hard work.
- Maybe we should change how we do things
  - Before we create new languages, perhaps we should spend some time figuring out how to productively compare them … so we can "peer review" them and make systematic progress.
- Let's define "a human language of programmability":
  - A community accepted design pattern language defining standard practice in parallel algorithm design.
  - A discussion framework of programmability.
  - Standard programmability benchmarks.

Appendices



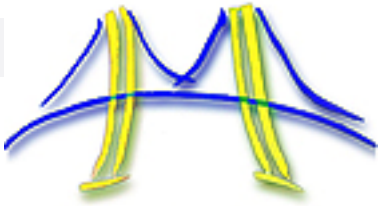
- Patterns in Our Pattern Language (OPL)
- Programming models for comparing parallel programming languages

OPL: Our Pattern Language



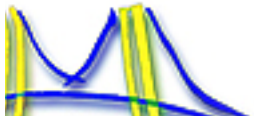
- The following slides list the patterns in Our Pattern Language (OPL) and provide a brief summary for each pattern.
- The patterns will be changing throughout the spring of 2009. Interested readers should consult the project wiki for updates.
  - http://parlab.eecs.berkeley.edu/wiki/patterns/patterns

Architectural Patterns



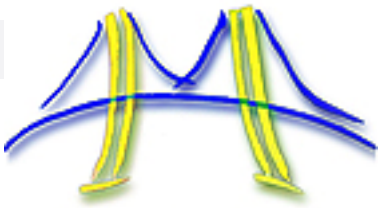
- These patterns define the overall architecture for a program.
  - Pipe-and-filter: view the program as filters (pipeline stages) connected by pipes (channels). Data flows through the filters to take input and transform into output.
  - Agent and Repository: a collection of autonomous agents update state managed on their behalf in a central repository.
  - Process control: the program is structured analogously to a process control pipeline with monitors and actuators moderating feedback loops and a pipeline of processing stages.
  - Event based implicit invocation: The program is a collection of agents that post events they watch for and issue events for other agents. The architecture enforces a high level abstraction so invocation of an agent is implicit; i.e. not hardwired to a specific controlling agent.
  - Model-view-controller: An architecture with a central model for the state of the program, a controller that manages the state and one or more agents that export views of the model appropriate to different uses of the model.
  - Bulk Iterative (AKA bulk synchronous): A program that proceeds iteratively … update state, check against a termination condition, complete coordination, and proceed to the next iteration.
  - Map reduce: the program is represented in terms of two classes of functions. One class maps input state (often a collection of files) into an intermediate representation. These results are collected and processed during a reduce phase.
  - Layered systems: an architecture composed of multiple layers that enforces a separation of concerns wherein (1) only adjacent layers interact and (2) interacting layers are only concerned with the interfaces presented by other layers.
  - Arbitrary static task graph: the program is represented as a graph that is statically determined meaning that the structure of the graph does not change once the computation is established. This is a broad class of programs in that any arbitrary graph can be used.

Computational Patterns



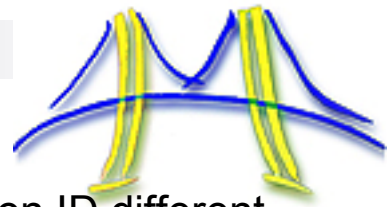
- These patterns describe computations that define the components in a programs architecture.
  - Backtrack, branch and bound: Used in search problems … where instead of exploring all possible points in the search space, we continuously divide the original problem into smaller subproblems, evaluate characteristics of the subproblems, set up constraints according to the information at hand, and eliminate subproblems that do not satisfy the constraints.
  - Circuits: used for bit level computations, representing them as Boolean logic or combinational circuits together with state elements such as flip-flops.
  - Dynamic programming: recursively split a larger problem into subproblems but with memorization to reuse past subsolutions.
  - Dense linear algebra: represent a problem in terms of dense matrices using standard operations defined in terms of Basic linear algebra (BLAS).
  - Finite state machine: Used in problems for which the system can be described by a language of strings. The problem is to define a piece of software that distinguishes between valid input strings (associated with proper behavior) and invalid input strings (improper behavior).
  - Graph algorithms: a diverse collection of algorithms that operate on graphs. Solutions involve preparing the best representation of the problem as a graph, and developing a graph traversal that captures the desired computation.
  - Graphical models: probabilistic reasoning problems where the problem is defined in terms of probability distributions represented as a graphical model.
  - Monte Carlo: A large class of problems where the computation is replicated over a large space of parameters. In many cases, random sampling is used to avoid exhaustive search strategies.
  - N-body: Problems in which each member of a system depends on the state of every other particle in the system. The problems typically involve some scheme to approximate the naïve O(N2) exhaustive sum.
  - Sparse Linear Algebra: Problems represented in terms of sparse matrices. Solutions may be iterative or direct.
  - Spectral methods: Problems for which the solution is easier to compute once the domain has been transformed into a different representation. Examples include Z-transform, FFT, DCT, etc. The transform itself is included in this class of problems.
  - Structured mesh: Problem domains are mapped onto a regular mesh and solutions computed as averages over neighborhoods of points (explicit methods) or as solutions to linear systems of equations (implicit methods)
  - Unstructured mesh: The same as the structured mesh problems, but the mesh lacks structure and hence, the computations involved scatter and gather operations.

Algorithm Patterns



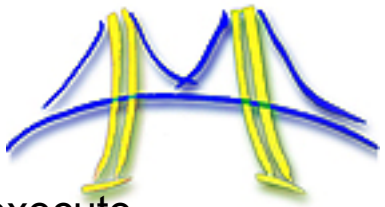
- These patterns describe parallel algorithms used to implement the computational patterns.
  - Task parallelism: Parallelism is expressed as a collection of explicitly defined tasks. This pattern includes the embarrassingly parallel pattern (no dependencies) and separable dependency pattern (replicated data/reduction).
  - Data parallelism: Parallelism is expressed as a single stream of tasks applied to each element of a data structure. This is generalized as an index space with the stream of tasks applied to each point in the index space.
  - Recursive splitting: A problem is recursively split into smaller problems until the problem is small enough to solve directly. This includes the divide and conquer pattern as a subset wherein the final result is produce by reversing the splitting process to assemble solutions to the leaf-node problems into the final global result.
  - Pipeline: Fixed coarse grained tasks with data flowing between them.
  - Geometric decomposition: A problem is expressed in terms of a domain that is decomposed spatially into smaller chunks. Solution is composed of updates across chunk boundaries, updates of local chunks, and then updates to the boundaries of the chunks.
  - Discrete event: a collection of tasks that coordinate among themselves through discrete events. This pattern is often used for GUI design and discrete event simulations.
  - Graph partitioning: Tasks generated by decomposing recursive data structures (graphs)

Software structure Patterns



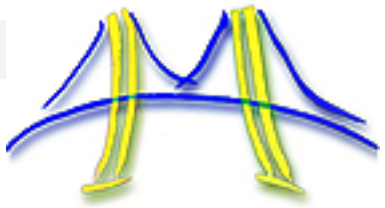
- Program structure
  - SPMD: One program used by all the threads or processes, but based on ID different paths or different segments of data are executed.
  - Strict data parallel: A single instruction stream is applied to multiple data elements. This includes vector processing as a subset.
  - Loop level parallelism: Parallelism is expressed in terms of loop iterations that are mapped onto multiple threads or processes.
  - Fork/join: Threads are logically created (forked), used to carry out a computation, and then terminated (joined).
  - Master-worker/Task-queue: A master sets up a collection or work-items (tasks), a collection of workers pull work-items from the master (a task-queue), carry out the computation, and then go back to the master for more work.
  - Actors: a collection of active software agents (the actors) interact over distinct channels.
  - BSP: The Bulk Synchronous model from Leslie Valiant.
- Data Structure Patterns
  - Shared queue: this pattern describes ways to any of the common queue data structures and manage them in parallel
  - Distributed array: An array data type that is distributed about a threads or processes involved with a parallel computation.
  - Shared hash table: A hash table shared/distributed among a set of threads or processes with any concurrency issues hidden behind an API.
  - Shared data: a “catch all” pattern for cases where data is shared within a shared memory region but the data can not be represented in terms of a well defined and common high level data structure.

Execution Patterns



- Process/thread control patterns
  - CSP or Communicating Sequential Processes: Sequential processes execute independently and coordinate their execution through discrete communication events.
  - Data flow: sequential processes organized into a static network with data flowing between them.
  - Task-graph: A directed acyclic graph of threads or processes is defined in software and mapped onto the elements of a parallel computer.
  - SIMD: A single stream of program instructions execute in parallel for different lanes in a data structure. There is only one program counter for a SIMD program. This pattern includes vector computations.
  - Thread pool: The system maintains a pool of threads that are utilized dynamically to satisfy the computational needs of a program. The pool of threads work on queues of tasks. Work stealing is often used to enforce a more balanced load.
  - Speculation: a thread or process is launched to pursue a computation, but any update to the global state is held in reserve to be entered once the computation is verified as valid.
- Coordination Patterns
  - Message passing: two sided and one sided message passing
  - Collective communication: reductions, broadcasts, prefix sums, scatter/gather etc.
  - Mutual exclusion: mutex and locks
  - Point to point synchronization: condition variables, semaphores
  - Collective synchronization: e.g. barriers
  - Transactional memory: transactions with roll-back to handle conflicts.

Appendices



- Patterns in Our Pattern Language (OPL)
- Programming models for comparing parallel programming languages

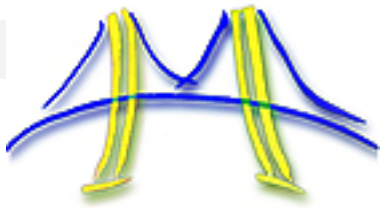
This is an extra topic.

In a more complete discussion of programmability, I describe the need for understanding the models programmers use when reasoning about programming.

These models are a useful way to think about programming languages. The models are their and research has shown people use them. A good programming language makes them accessible to the programmer; bad languages hide them.

Bad languages also may force the programmer to deal with the models in a particular order. In other words, what we want is a language that exposes models, but at an abstract level and under programmer control.

Cognitive Psychology and human reasoning

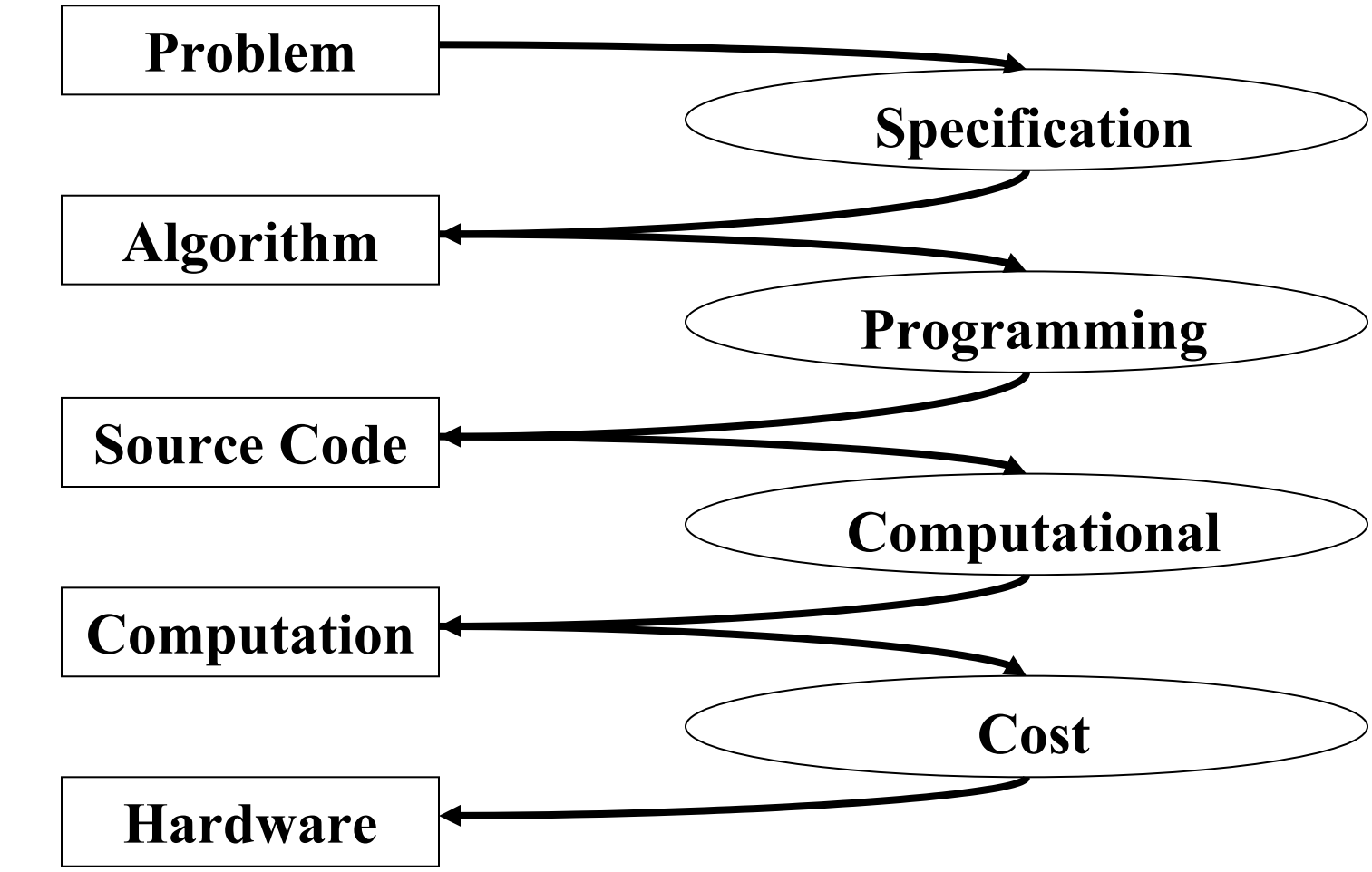


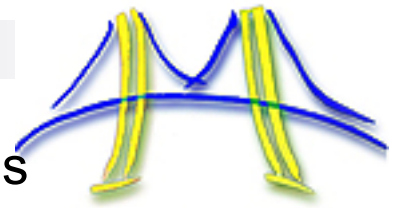
- Human Beings are model builders
  - We build hierarchical complexes of mental models.
  - Understand sensory input in terms of these models.
  - When input conflicts with models, we tend to believe the models.
- Programming is a process of successive refinement of a problem over a hierarchy of models.
- The models represent the problem at a different level of abstraction.
  - The top levels express the problem in the original problem domain.
  - The lower levels represent the problem in the computer's domain.
- The models are informal, but detailed enough to support simulation.

Model based reasoning in programming

Models	Domain
Specification	Problem Specific: polygons, rays, molecules, etc.
Programming	OpenMP's fork/join, Actors
Computation	Threads – shared memory Processes – shared nothing
Machine (AKA Cost Model)	Registers, ALUs, Caches, Interconnects, etc.

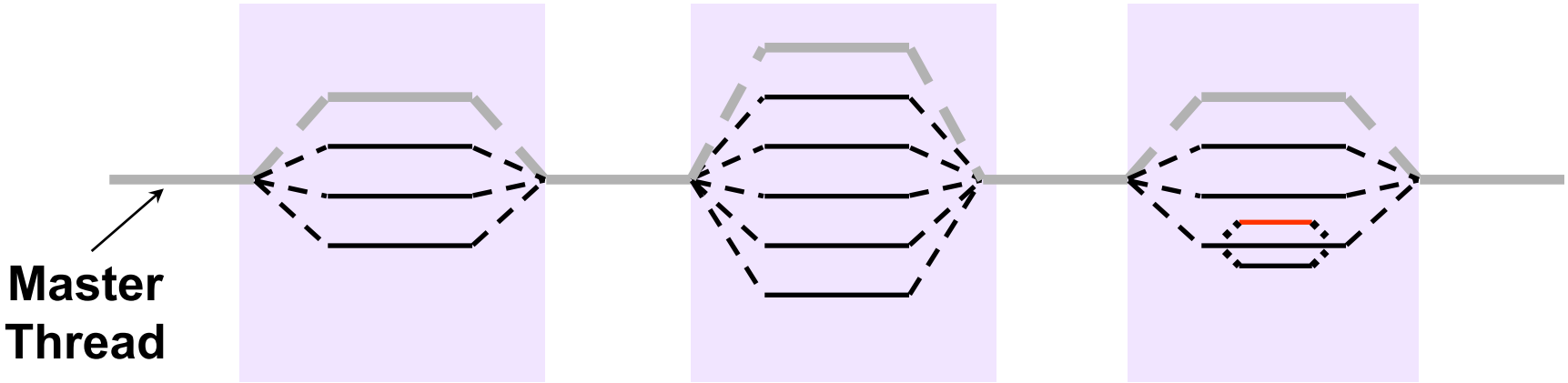
Another View of the models



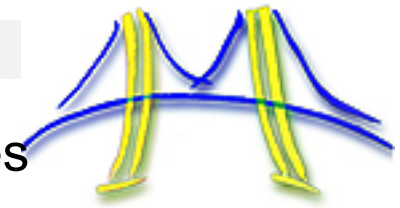


Using the Models to precisely describe programming languages

- Consider OpenMP 2.5:
  - Optimized for loop-level parallelism so limited specification models.
  - Defines a programming model (fork/join)
  - Implies a computational model (SMP)
  - Has no cost model at all … making it very difficult for us to deal with any level of NUMA



- By describing OpenMP in terms of explicit models, we expose the key issues and create a framework for comparison to other approaches.



Using the Models to precisely describe programming languages

- Consider MPI:
  - General parallelism, supports a wide range specification models.
  - Defines a programming model (SPMD, MIMD)
  - Implies a computational model (a variation of CSP)
  - Implies a cost model, so it does well on SMP to NUMA to cluster
- Comparing MPI and OpenMP
  - The foundations of MPI are more complete than for OpenMP since in MPI, each layer of the model hierarchy is addressed.