



Teaching people how to “think parallel”

Tim Mattson

Principal Engineer
Application Research Laboratory
Intel Corp

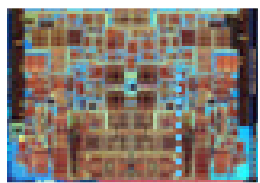
Disclaimer

- This is not an “Intel talk”.
- The views expressed in this presentation are my own and do not represent the views of Intel
- Also note that these slides have not been formally reviewed to verify proper use of trademarks, references to third party products, or other legal-issues.
- So please, if there are any problems, blame me and not Intel

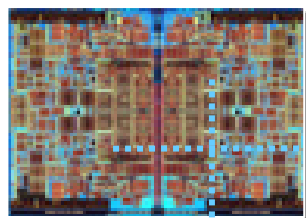
Agenda

- ⇒ ■ Many core motivation slides
- Psychology of programming
- Design Patterns and Pattern Languages
- Berkeley patterns effort

Future processors: How many cores is “many core”



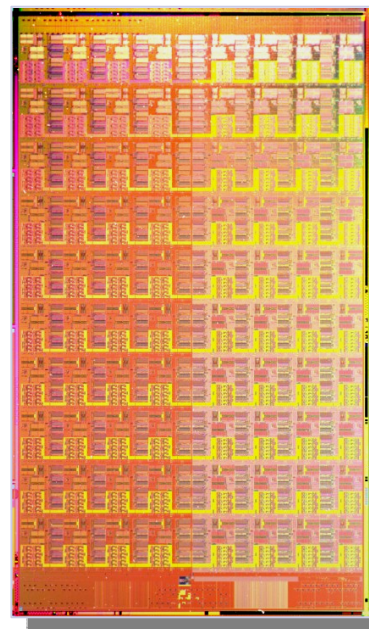
Dual Core (2006)



Quad-Core (2007)

Think big ... dozens or even hundreds of cores is not too far down the road.

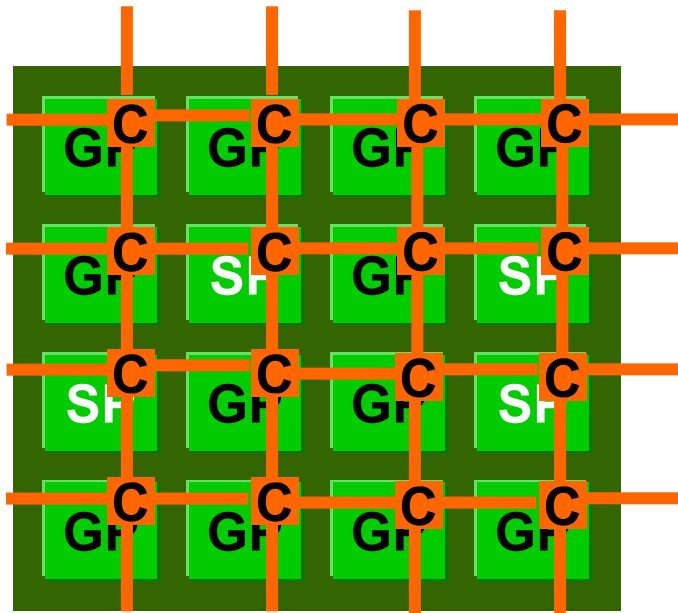
**Tera-scale research chip:
1 SP TFLOP* for 97 W
~100 Million transistors**



80-Core (2007)

*single precision TFLOP with the Stencil application kernel. Note: this is a research chip and not a product

Future Multi-core CPU



General Purpose Cores

Special Purpose HW

Interconnect fabric

**If this is going to be a general purpose CPU,
we need “all software” to be parallel.
Where will the parallel software come from?**

We need to educate a new generation of parallel programmers

- Parallel Programming is difficult, error prone and only accessible to small cadre of experts “do it” ... Clearly we haven't been doing a very good job at educating programmers.
- Consider the following:
 - All known programmers are human beings.
 - Hence, human psychology, not hardware design, should guide our work on this problem.

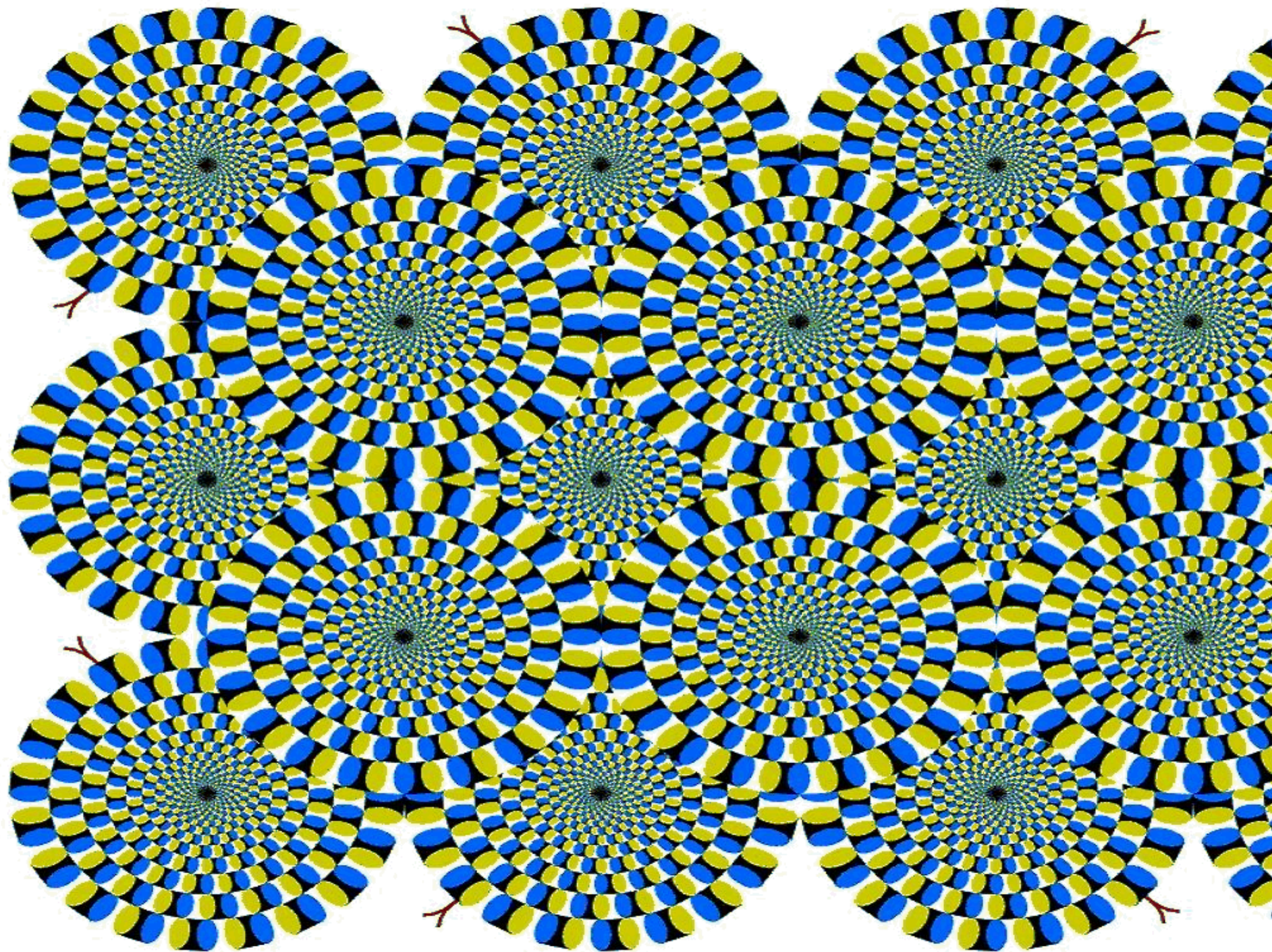
Teach Parallel programming around the needs of the programmer, not the needs of the computer.

Agenda

- Many core motivation slides
- ➔ ■ Psychology of programming
- Design Patterns and Pattern Languages
- Berkeley patterns effort

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 and models

- Programming is a process of successive refinement of a problem over a hierarchy of models. [**Brooks83**]
- 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

Specification

Programming

Computation

Machine
(AKA Cost Model)

Domain

Problem Specific: polygons,
rays, molecules, etc.

OpenMP's fork/join,
Actors

Threads – shared memory
Processes – shared nothing

Registers, ALUs, Caches,
Interconnects, etc.

Programming process: getting started.

- The programmer starts by constructing a high level model from the specification.
- Successive refinement starts by using some combination of the following techniques:
 - The problem's state is defined in terms of objects belonging to abstract data types with meaning in the original problem domain.
 - Key features of the solution are identified and emphasized. These features, sometimes referred to as "beacons" [**Wiedenbeck89**] , emphasize key aspects of the solution.

The programming process

- Programmers use an informal, internal notation based on the problem, mathematics, programmer experience, etc.
 - Within a class of programming languages, the program generated is only weakly dependent on the language. **[Robertson90] [Petre88]**
- Programmers think about code in chunks or “plans”. **[Rist86]**
 - Low level plans code a specific operation: e.g. summing an array.
 - High level or global plans relate to the overall program structure.

Programming Process: Strategy + Opportunistic Refinement

- Common strategies are:
 - Backwards goal chaining - start at result and work backwards to generate sub goals. Continue recursively until plans emerge.
 - Forward chaining: Directly leap to plans when a problem is familiar. **[Rist86]**
- Opportunistic Refinement: **[Petre90]**
 - Progress is made at multiple levels of abstraction.
 - Effort is focused on the most productive level.

Programming Process: the role of testing

- Programmers test the emerging solution throughout the programming process.
- Testing consists of two parts:
 - Hypothesis generation: The programmer forms an idea of how a portion of the solution should behave.
 - Simulation: The programmer runs a mental simulation of the solution within the problem models at the appropriate level of abstraction. [**Guindom90**]

From psychology to software

- My central hypothesis:
 - A Design Pattern language provides the roadmap to apply results from the psychology of programming to software engineering:
 - **Design patterns capture the essence of plans**
 - **The structure of patterns in a pattern language should mirror the types of models programmers use.**
 - **Connections between patterns must fit well with goal-chaining and opportunistic refinement.**

Agenda

- Many core motivation slides
- Psychology of programming
- ■ Design Patterns and Pattern Languages
- Berkeley patterns effort

Design Patterns and Pattern Languages

- 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

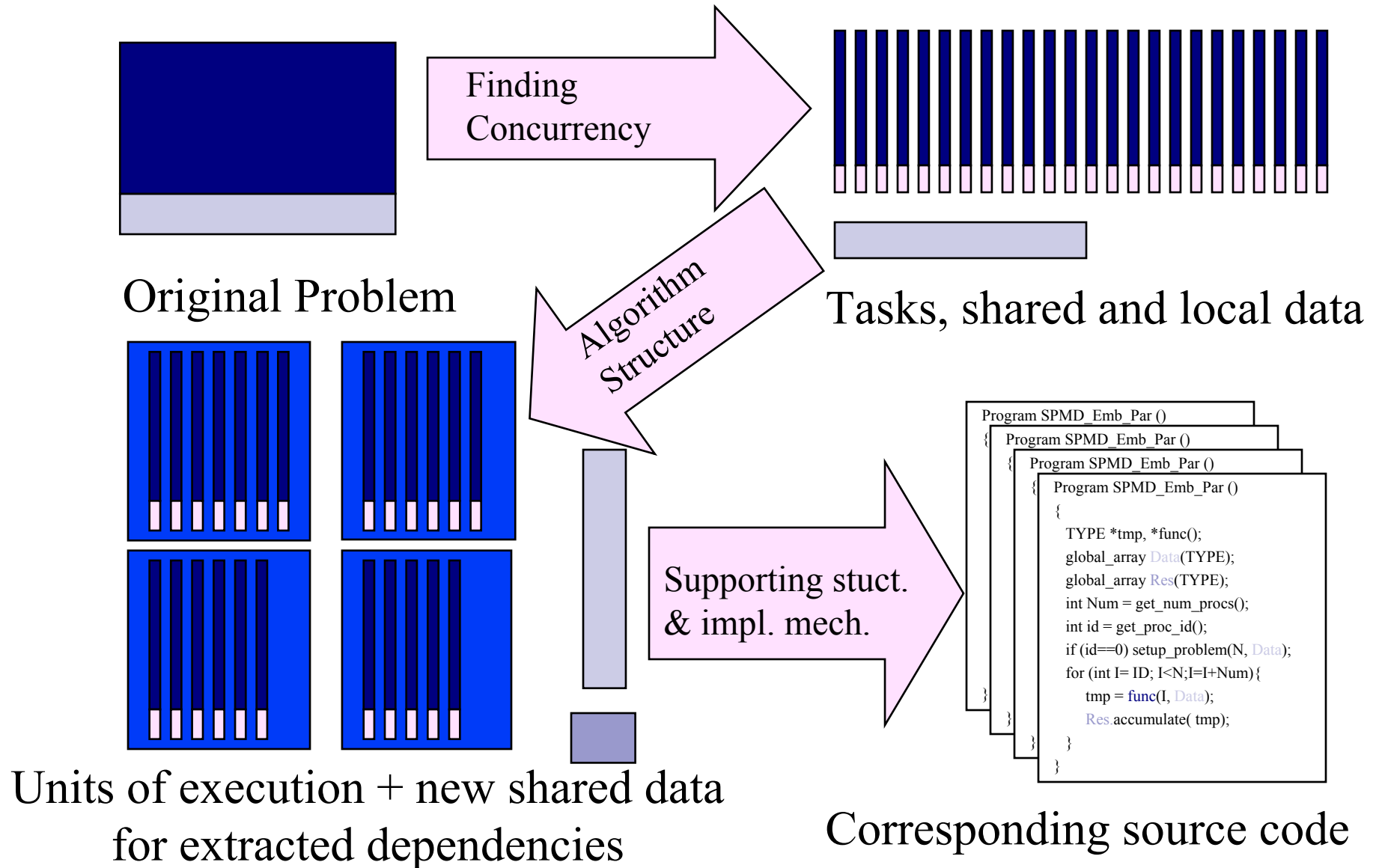
- Name: Money Pipeline
- Context: **You want to get rich and all you have to work with is a C.S. degree and programming skills.** How can you use software to get rich?
- Forces: **The solution must resolve the forces:**
 - **It must give the buyer something they believe they need.**
 - **It can't be 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.**

Design Patterns

- Some example design patterns:
 - **Iterator**: visit each object in a collection without exposing the object's internal representation.
 - **Task Queue**: the well known approach for embarrassingly parallel problems. Master-worker algorithms are a common example.
 - **Facade**: Provide a unified interface to a set of distinct interfaces in a subsystem.
- Each pattern is written down in a specific format and includes all the information required to apply it for a particular problem.

Our Pattern Language's structure:

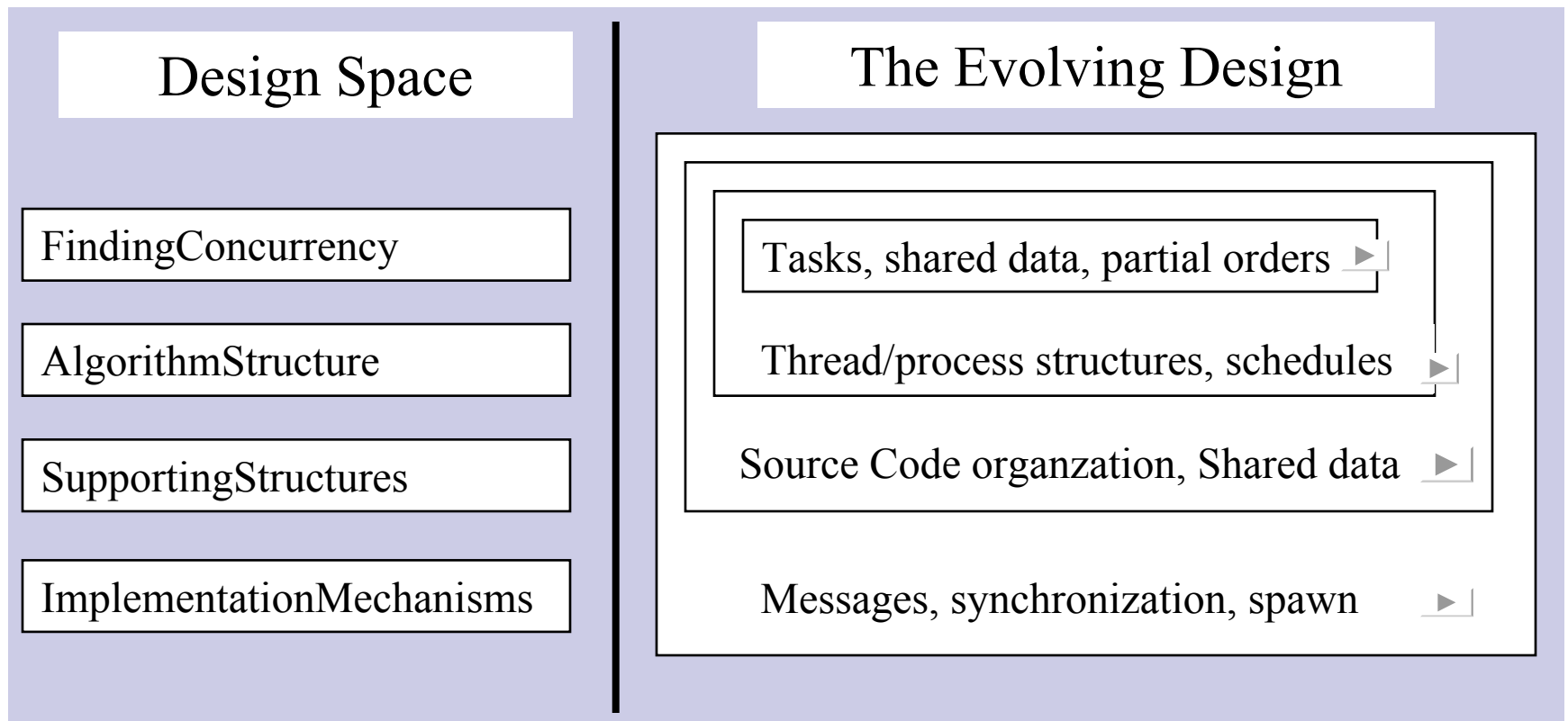
Four design spaces in parallel software development

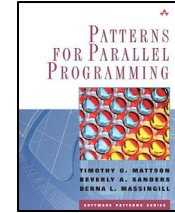


Our Pattern Language's Structure

A software design can be viewed as a series of refinements.

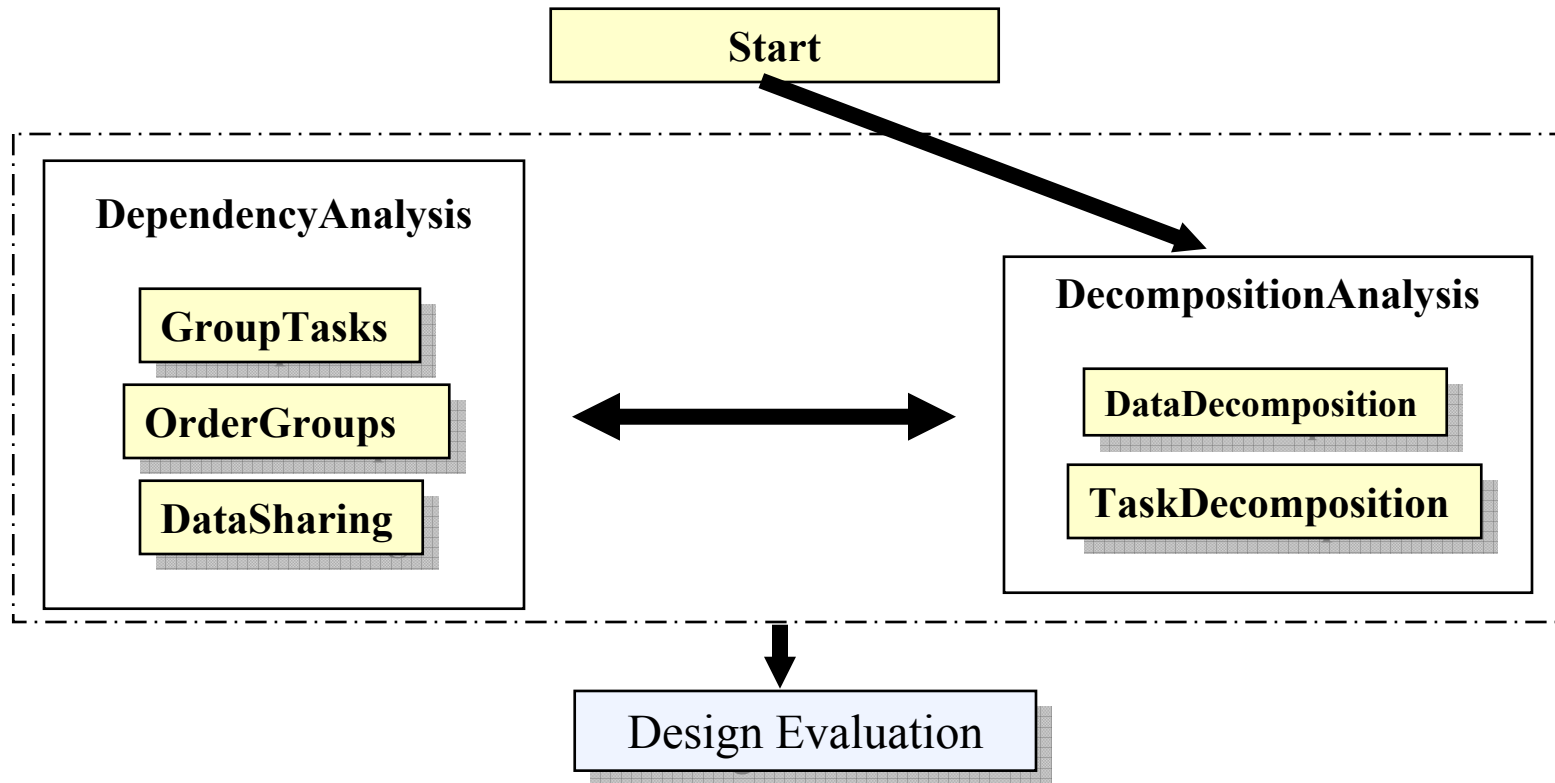
We consider the process in terms of 4 *design spaces* which add progressively lower level elements to the design.



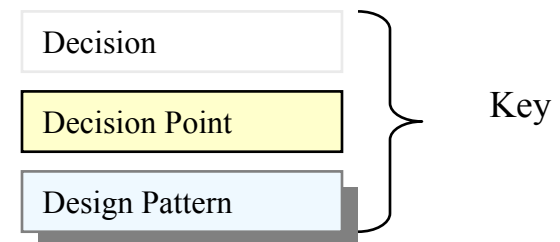
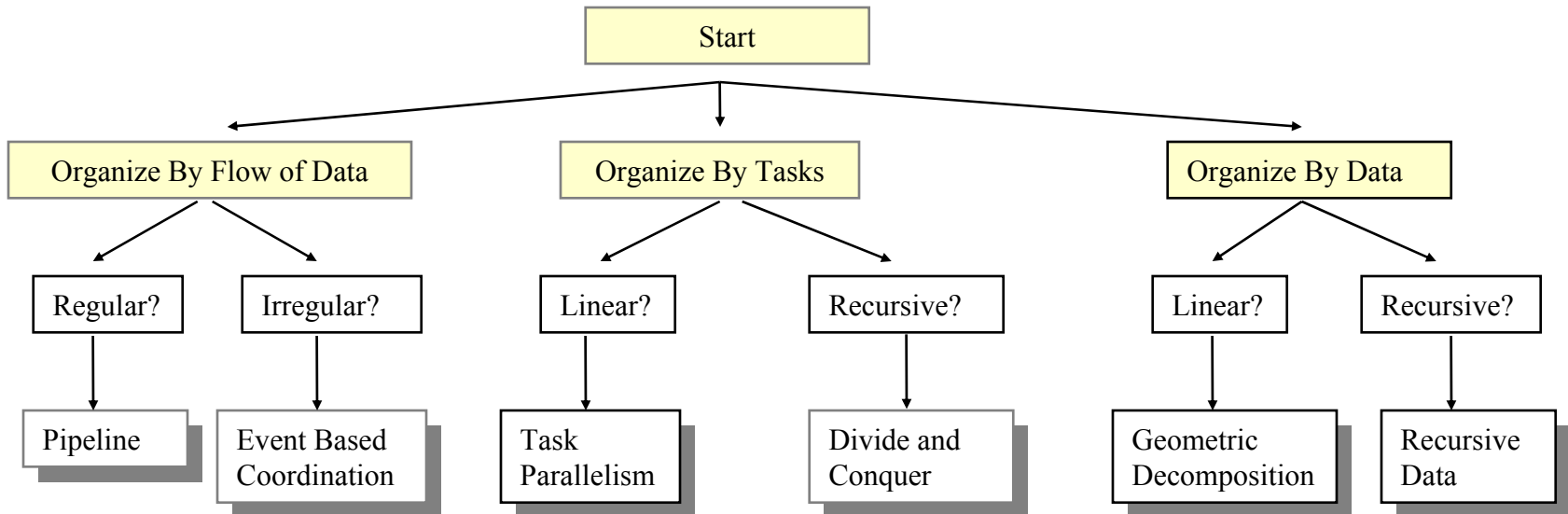
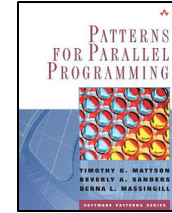


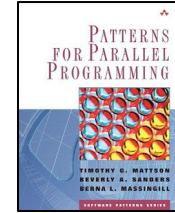
Decomposition (Finding Concurrency)

Start with a specification that solves the original problem -- finish with the problem decomposed into tasks, shared data, and a partial ordering.



Concurrency Strategy (Algorithm Structure)

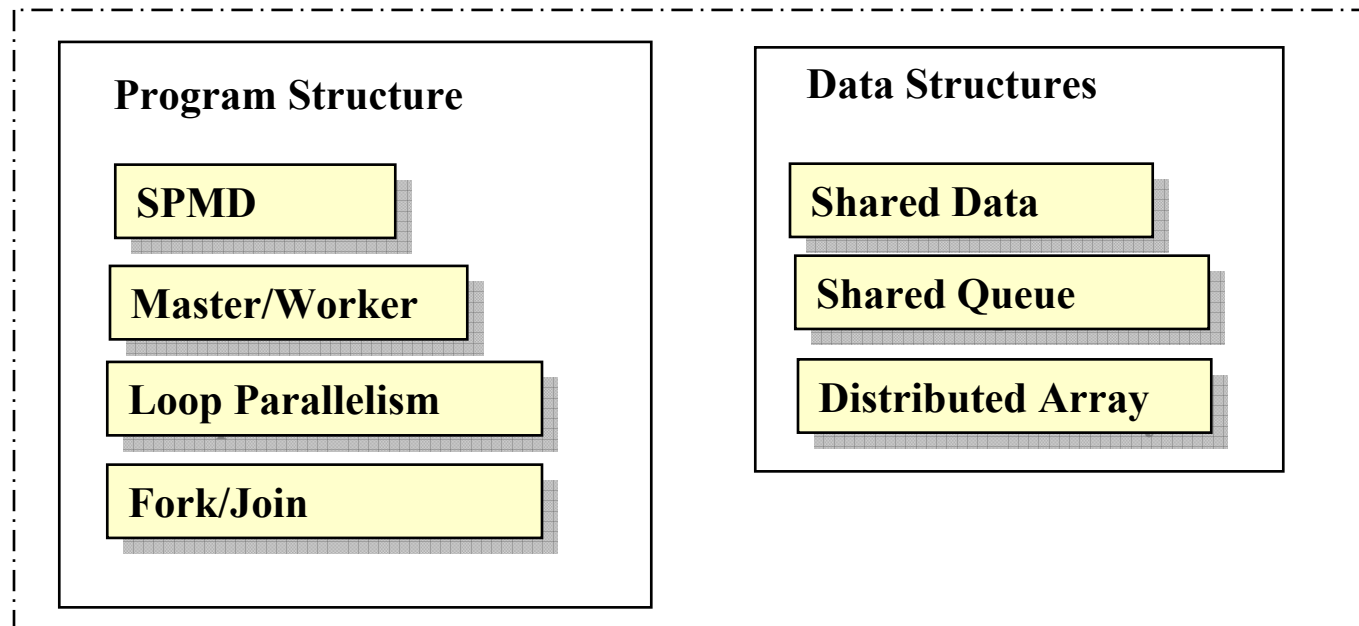


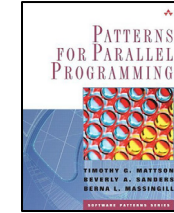


Implementation strategy

(Supporting Structures)

High level constructs impacting large scale organization of the source code.

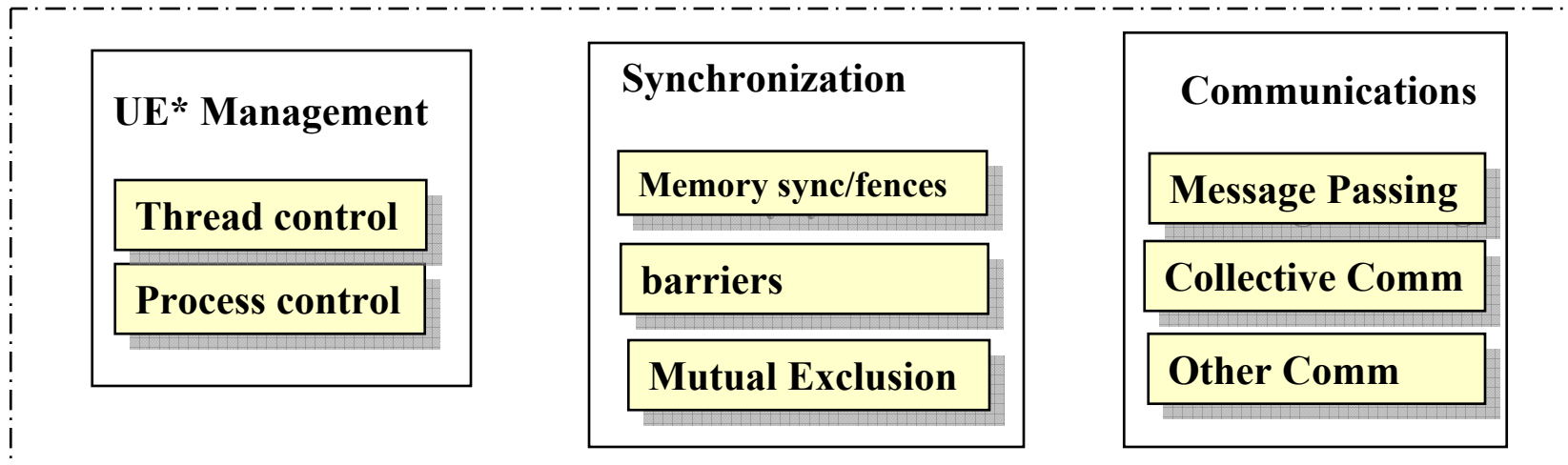




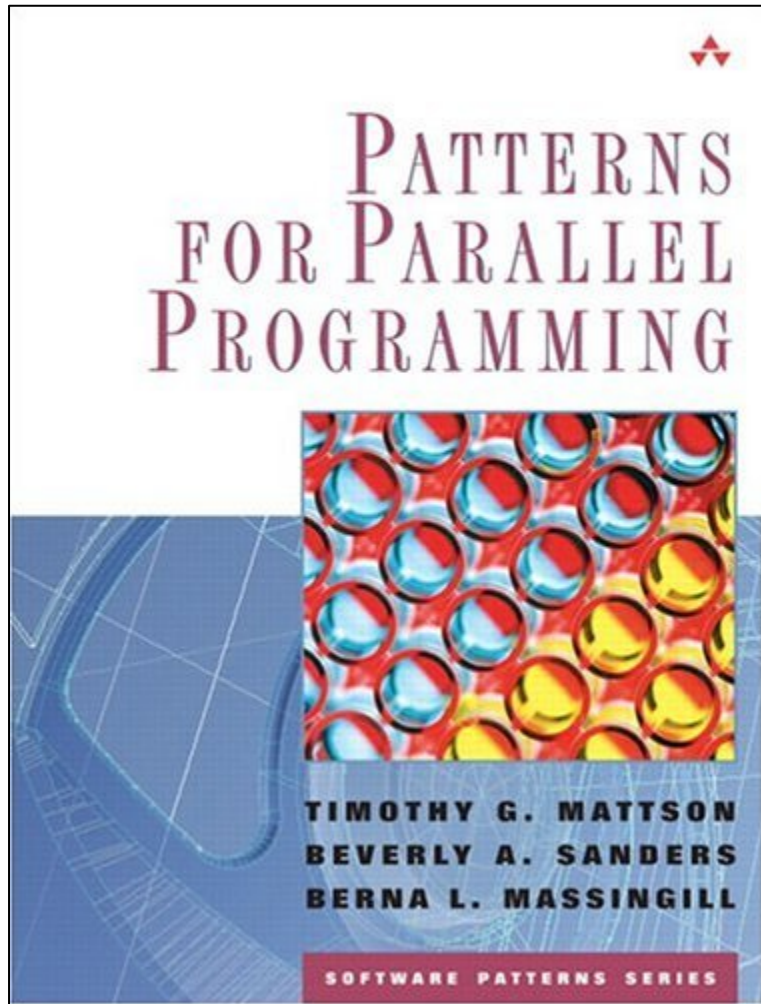
Parallel Programming building blocks

Low level constructs implementing specific constructs used in parallel computing. Examples in Java, OpenMP and MPI.

These are not properly design patterns, but they are included to make the pattern language self-contained.



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!

Name: The Task Parallelism Pattern

- Context:
 - How do you exploit concurrency expressed in terms of a set of distinct tasks?
- Forces
 - Size of task – small size to balance load vs. large size to reduce scheduling overhead.
 - Managing dependencies without destroying efficiency.
- Solution
 - Schedule tasks for execution with balanced load – use master worker, loop parallelism, or SPMD patterns.
 - Manage dependencies by:
 - removing them (replicating data),
 - transforming induction variables,
 - exposing reductions,
 - explicitly protecting (shared data pattern).

Name: The SPMD Pattern

- Context:
 - How do you structure a parallel program to make interactions between threads manageable yet easy to integrate with the core computation?
- Forces
 - Fewer programs are easier to manage, but complex algorithms often need very different instruction streams on each thread.
 - Balance the conflicting needs of scalability, maintainability, and portability.
- Solution
 - Use a single program for all the threads.
 - Keep it simple ... use the threads ID to select different pathways through the program.
 - Keep interactions between threads explicit and at a minimum.

Name: The Loop Parallelism Pattern

■ Context:

- How do you transform a serial program dominated by compute intensive loops into a parallel program without radically changing the semantics?

■ Forces

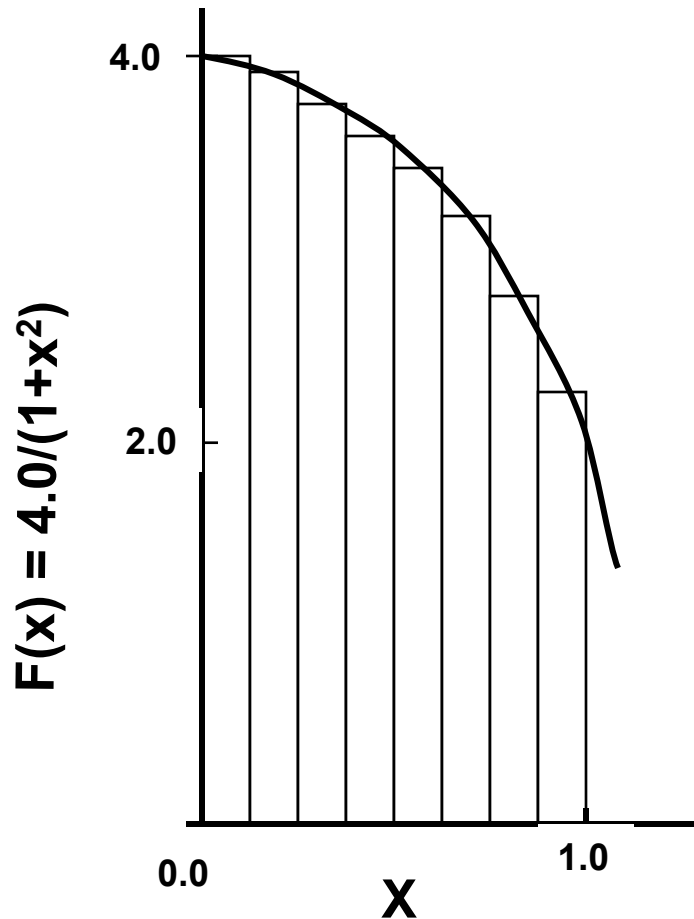
- An existing program implies expected output ... and this must be preserved even as the programs execution changes due to parallelism.
- High performance requires restructuring data to optimize for cache ... but this must be done carefully to avoid changing semantics.
- Assuring correctness suggests that the parallelization process should be incremental with each transformation subject to testing.

■ Solution

- Identify key compute intensive loops.
- Use directives/pragmas to parallelize these loops (i.e. at no point do you use the ID to manage loop parallelization by hand).
- Optimize incrementally testing at each step of the way – merging loops, modifying schedules, etc.

A simple pedagogical Example: The PI program

Numerical Integration



Mathematically, we know that:

$$\int_0^1 \frac{4.0}{(1+x^2)} dx = \pi$$

We can approximate the integral as a sum of rectangles:

$$\sum_{i=0}^N F(x_i) \Delta x \approx \pi$$

Where each rectangle has width Δx and height $F(x_i)$ at the middle of interval i .

OpenMP PI Program:

SPMD Pattern

```
#include <omp.h>
static long num_steps = 100000;      double step;
#define NUM_THREADS 2
void main ()
{
    int i;  double x, pi, sum[NUM_THREADS] = {0};
    step = 1.0/(double) num_steps;
    omp_set_num_threads(NUM_THREADS);
#pragma omp parallel
    {
        double x;  int id, i;
        id = omp_get_thread_num();
        int nthreads = omp_get_num_threads();
        for (i=id;i< num_steps; i=i+nthreads) {
            x = (i+0.5)*step;
            sum[id] += 4.0/(1.0+x*x);
        }
    }
    for(i=0, pi=0.0;i<NUM_THREADS;i++)pi += sum[i] * step;
}
```

SPMD Programs:

Each thread runs the same code with the thread ID selecting thread-specific behavior.

MPI Pi program

SPMD pattern

```
#include <mpi.h>
void main (int argc, char *argv[])
{
    int i, my_id, numprocs; double x, pi, step, sum = 0.0 ;
    step = 1.0/(double) num_steps ;
    MPI_Init(&argc, &argv) ;
    MPI_Comm_Rank(MPI_COMM_WORLD, &my_id) ;
    MPI_Comm_Size(MPI_COMM_WORLD, &numprocs) ;
    for (i=my_id; i<num_steps; i=i+numprocs)
    {
        x = (i+0.5)*step;
        sum += 4.0/(1.0+x*x);
    }
    sum *= step ;
    MPI_Reduce(&sum, &pi, 1, MPI_DOUBLE, MPI_SUM, 0,
              MPI_COMM_WORLD) ;
}
```

SPMD Programs:

Each thread runs the same code with the thread ID selecting thread-specific behavior.

OpenMP PI Program:

Loop level parallelism pattern

```
#include <omp.h>
static long num_steps = 100000;      double step;
#define NUM_THREADS 2
void main ()
{
    int i;  double x, pi, sum =0.0;
    step = 1.0/(double) num_steps;
    omp_set_num_threads(NUM_THREADS);
#pragma omp parallel for private(x) reduction (+:sum)
    for (i=0;i< num_steps; i++){
        x = (i+0.5)*step;
        sum += 4.0/(1.0+x*x);
    }

    pi = sum[i] * step;
}
```

Loop Level Parallelism:

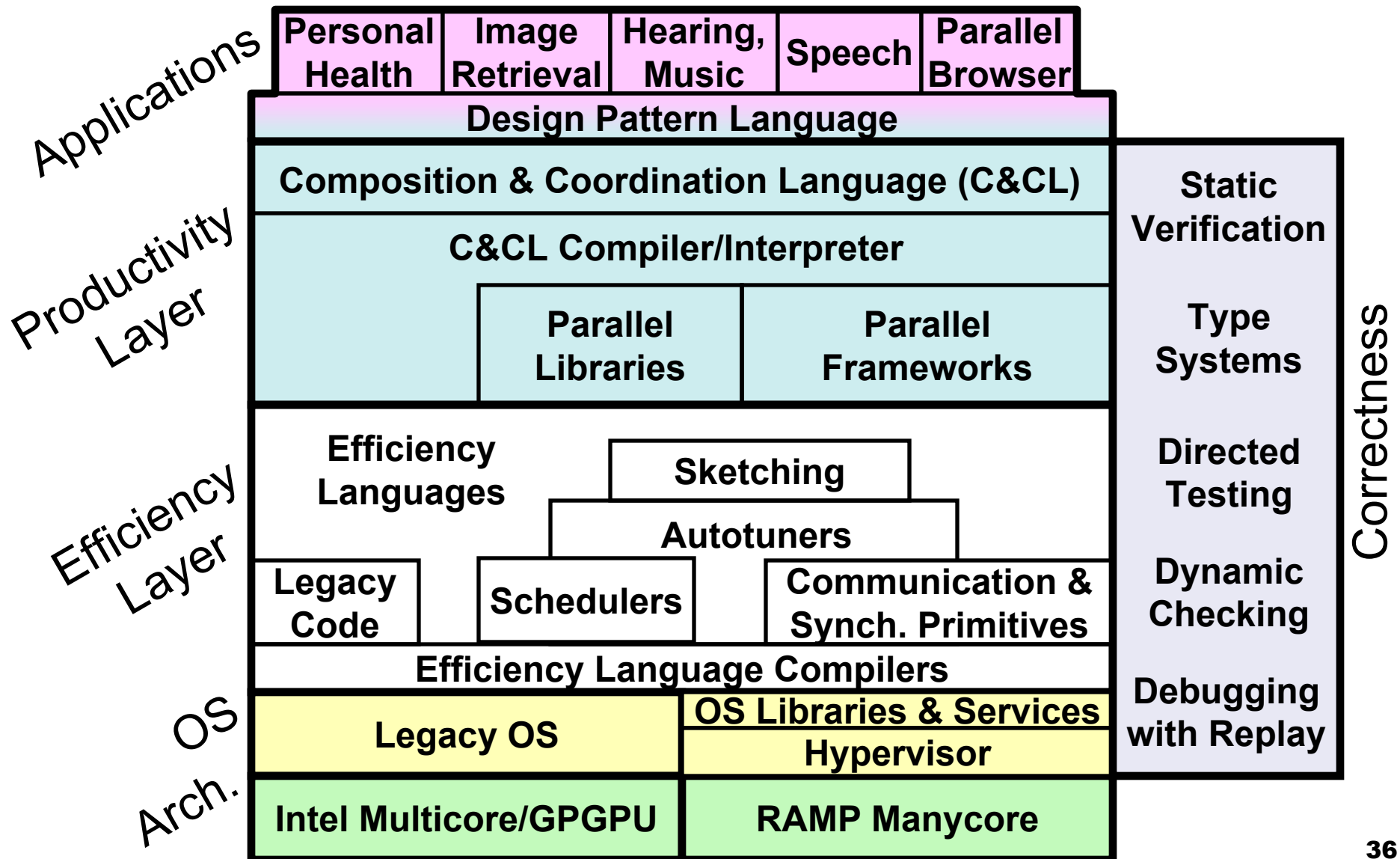
Parallelism expressed solely by (1) exposing concurrency, (2) managing dependencies, and (3) splitting up loops .

Agenda

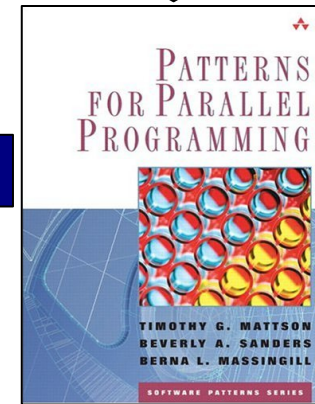
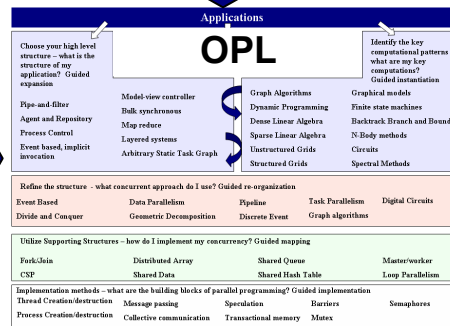
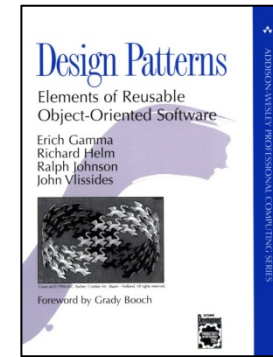
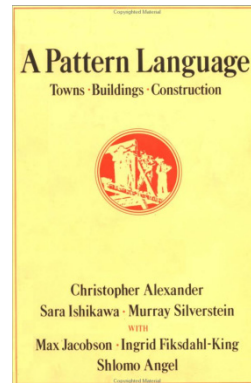
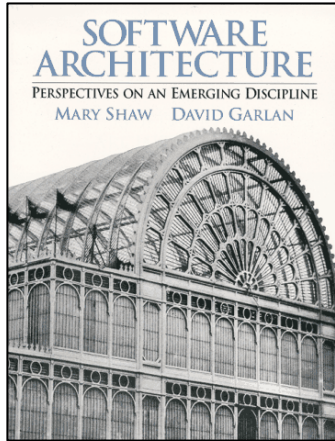
- Many core motivation slides
- Psychology of programming
- Design Patterns and Pattern Languages
- ⇒ ■ Berkeley patterns effort

UCB's Par Lab: Research Overview

Easy to write correct software that runs efficiently on manycore



Influences on Our Pattern Language (OPL)



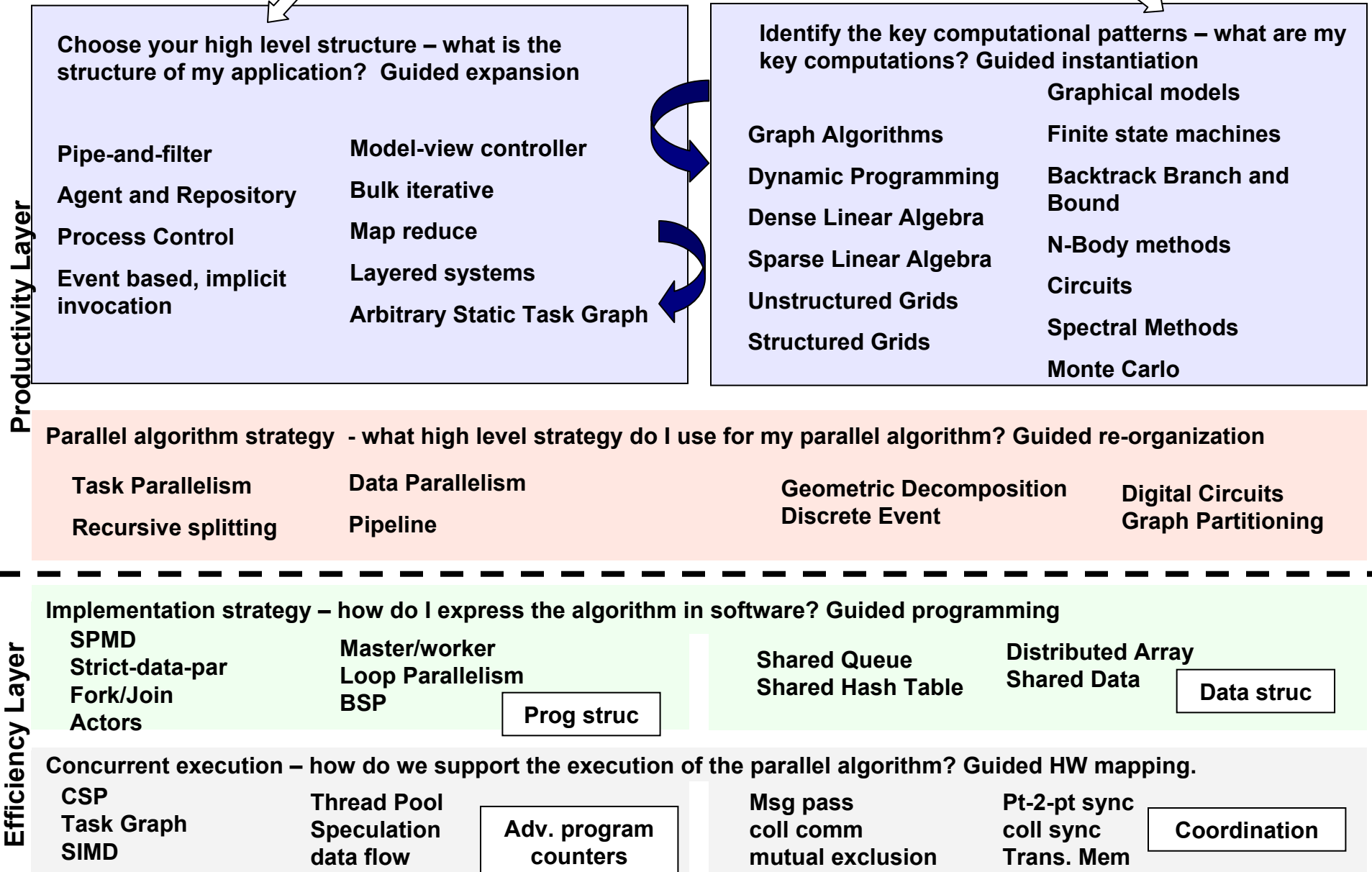
PLPP: Parallel Language for Parallel Programming

	Embed	SPEC	DB	Games	ML	HPC	Health	Image	Speech	Music	Browser	CAD
Finite State Mach.												
Circuits												
Graph Algorithms												
Structured Grid												
Dense Matrix												
Sparse Matrix												
Spectral (FFT)												
Dynamic Prog												
N-Body												
Backtrack/ B&B												
Graphical Models												
Unstructured Grid												

13 dwarves

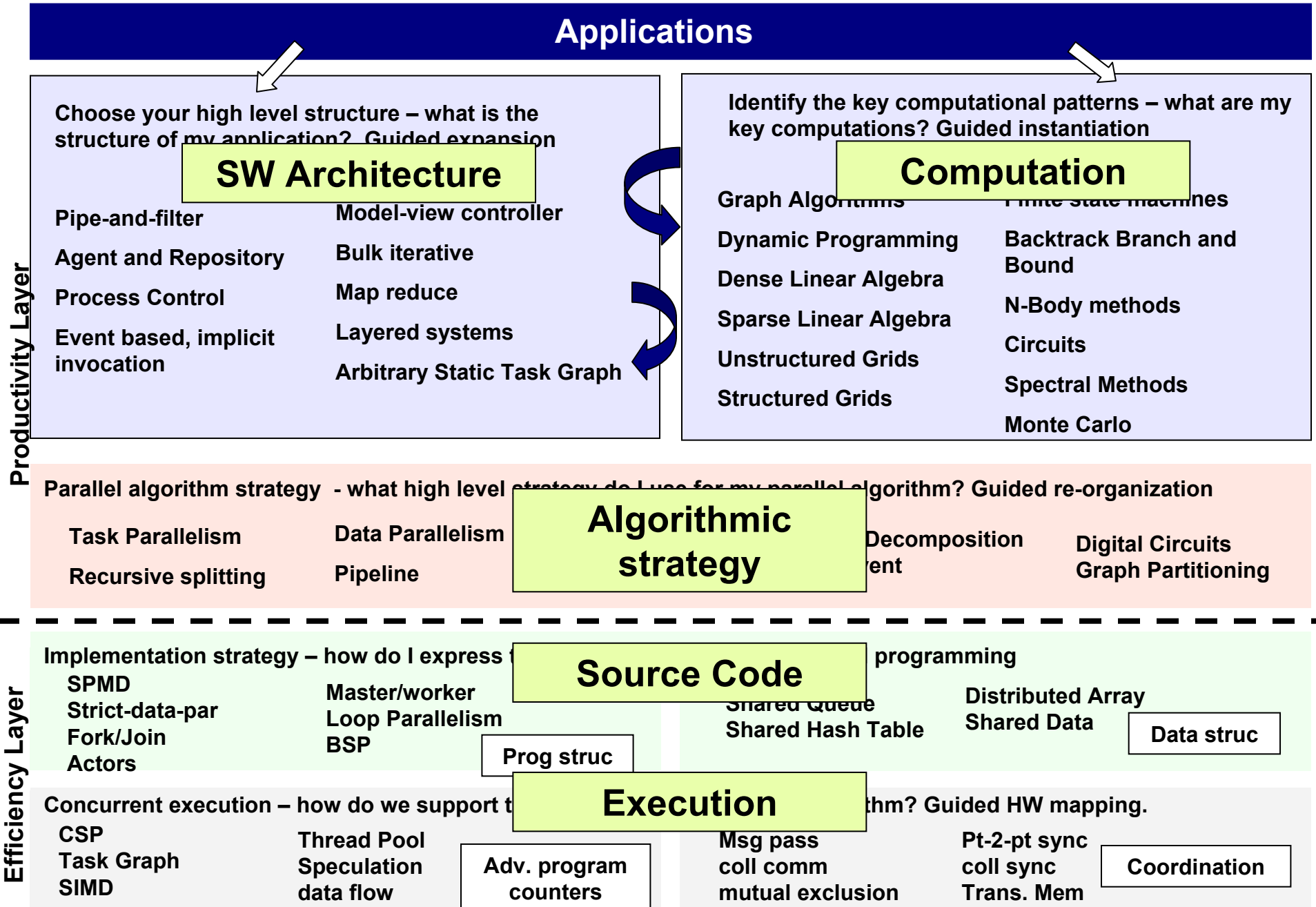
OPL Version 2.0 <http://parlab.eecs.berkeley.edu/wiki/patterns/patterns>

Applications



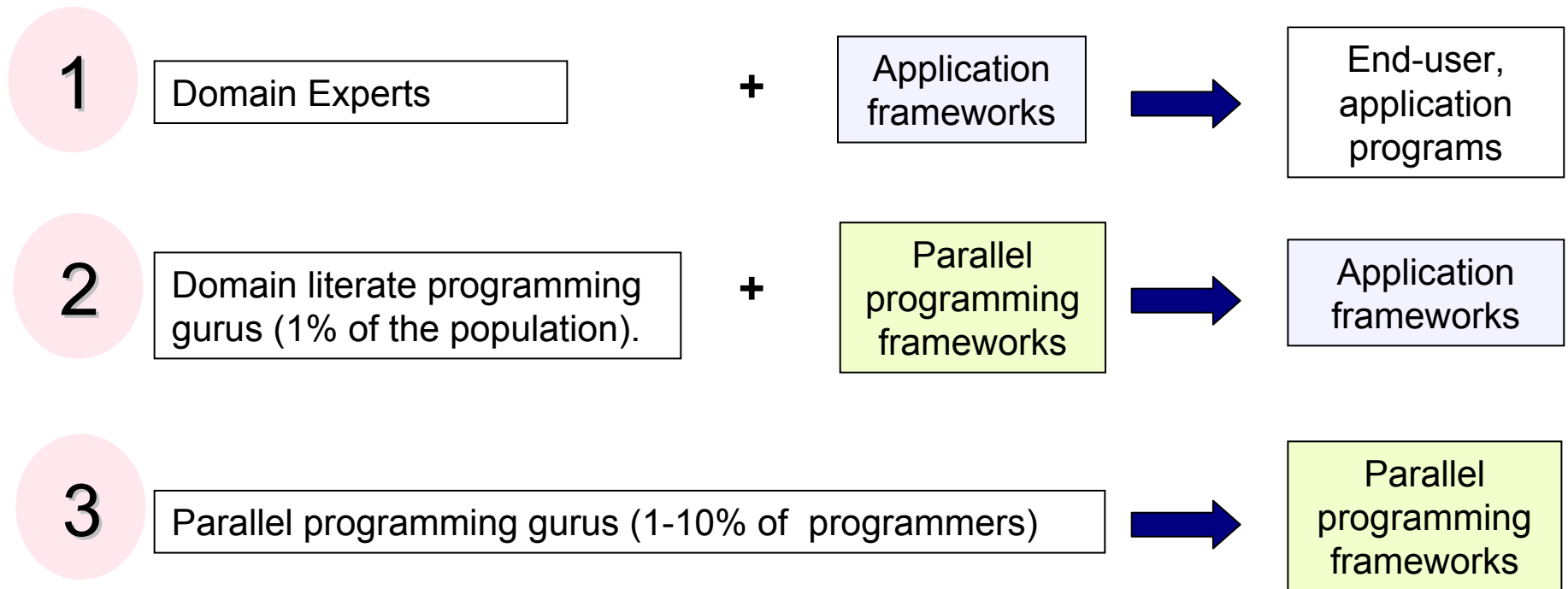
OPL Version 2.0 <http://parlab.eecs.berkeley.edu/wiki/patterns/patterns>

Applications



But patterns are not an end in themselves:

Design patterns inform creation of software frameworks



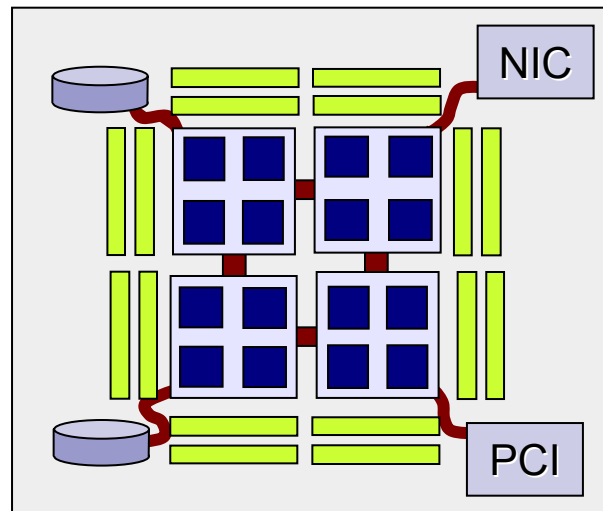
The hope is for Domain Experts to create parallel code with little or no understanding of parallel programming.

Leave hardcore “bare metal” efficiency layer programming to the parallel programming experts

To make our example concrete ...

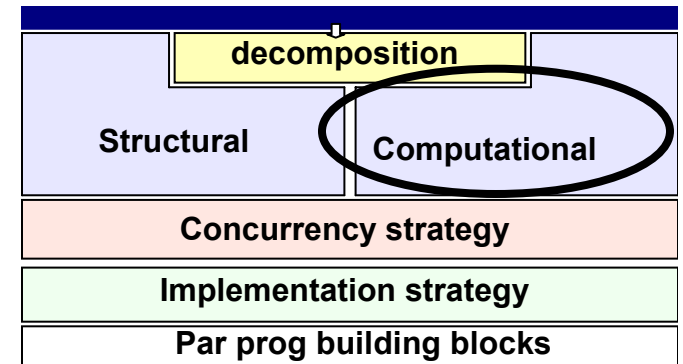
■ Reference Platform:

- a multi-socket system with 4 sockets each of which has a quad-core CPU.
 - Assume cores in a socket share a last level cache.
 - No shared address space between sockets (message passing)
- Expect that over the lifetime of the software, the number of sockets will stay fixed, but the number of cores will increase to 64/socket.



■ Efficiency layer

- Distributed memory between sockets
 - MPI
- Shared Address space on a socket
 - Pthreads
 - OpenMP



Spectral Methods computational pattern

- Name:
 - Spectral Methods
- Problem:
 - Some problems are easier to solve if you first transform the representation ... e.g. from the time domain to the frequency (spectral) domain. How do we solve these problems in parallel?
- Context:
 - PDE or image processing problem that is easier to solve following a transform.
- Solution:
 - Apply a discrete transform to the PDE turning differential operators into algebraic operators.
 - Solve the resulting system of algebraic or ordinary differential equations.
 - Inverse transform the solution to return to the original domain.

Spectral method: example

- Solve Poisson's equation:

$$\left(\frac{\partial^2}{\partial x^2} + \frac{\partial^2}{\partial y^2}\right)f(x, y) = g(x, y)$$

- Apply the 2D Discrete Fourier transforms:

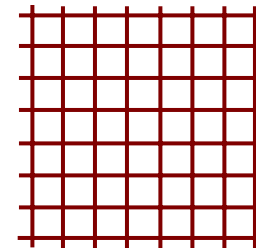
$$f = \sum a_{j,k} e^{2\pi i(jx+ky)} \quad g = \sum b_{j,k} e^{2\pi i(jx+ky)}$$

- Differential equation becomes an algebraic equation:

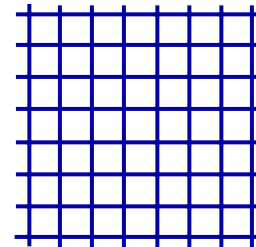
$$4\pi^2 \sum -a_{j,k} (j^2 + k^2) e^{2\pi i(jx+ky)} = \sum b_{j,k} e^{2\pi i(jx+ky)}$$

$$a_{j,k} = -\frac{b_{j,k}}{4\pi^2 (j^2 + k^2)}$$

See lecture 6 on meshes for more

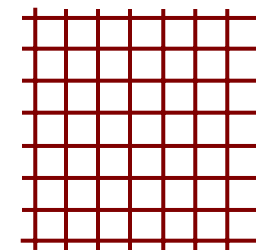


PDE System
defined on a
2D spatial grid



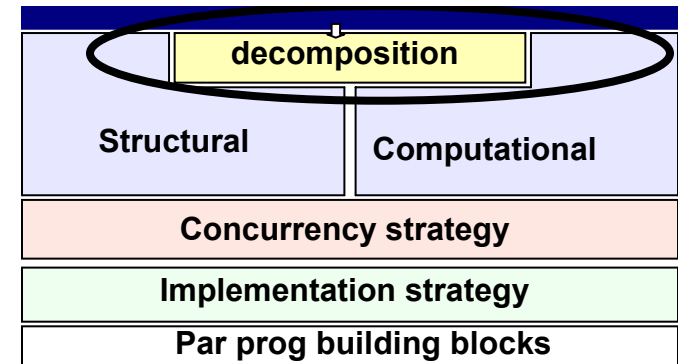
Transform
onto a 2D
frequency grid

Solve algebraically



Transform
back into
spatial domain
for final
answer

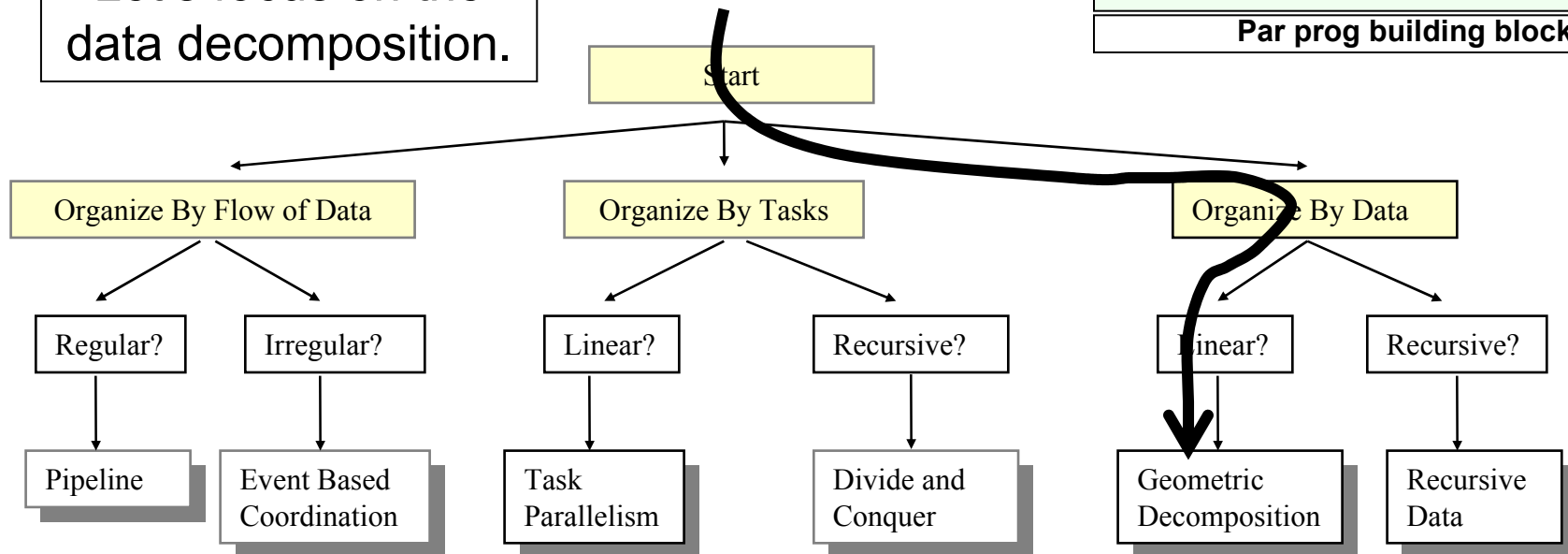
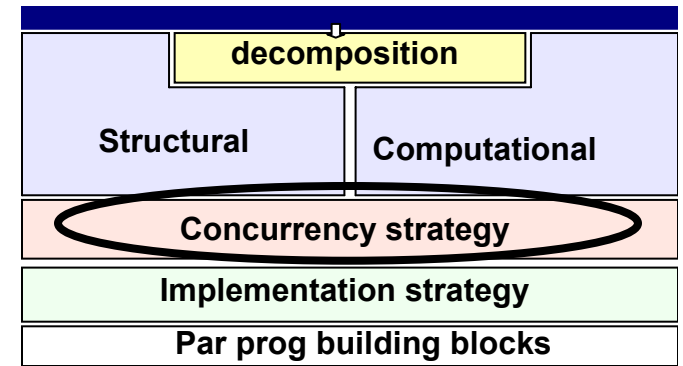
PLPP Patterns: Finding Concurrency



- Task Decomposition:
 - 1D FFTS making up the 2D FFT
 - Computing $a_{j,k}$ from $b_{j,k}$
- Data Decomposition
 - Column based decomposition to support FFT tasks
 - Any decomposition will do for $a_{j,k}$ computation
- Grouping tasks
 - Each phase of 2D FFT
 - $a_{j,k}$ update
- Data dependency
 - Transpose (all to all) between phases of 2D FFT
- Design evaluation
 - May want to parallelize 1D FFT if more concurrency is needed
 - Not needed for modest core count, but as number of cores increase, this could be very important.

The Algorithm Structure Design Space

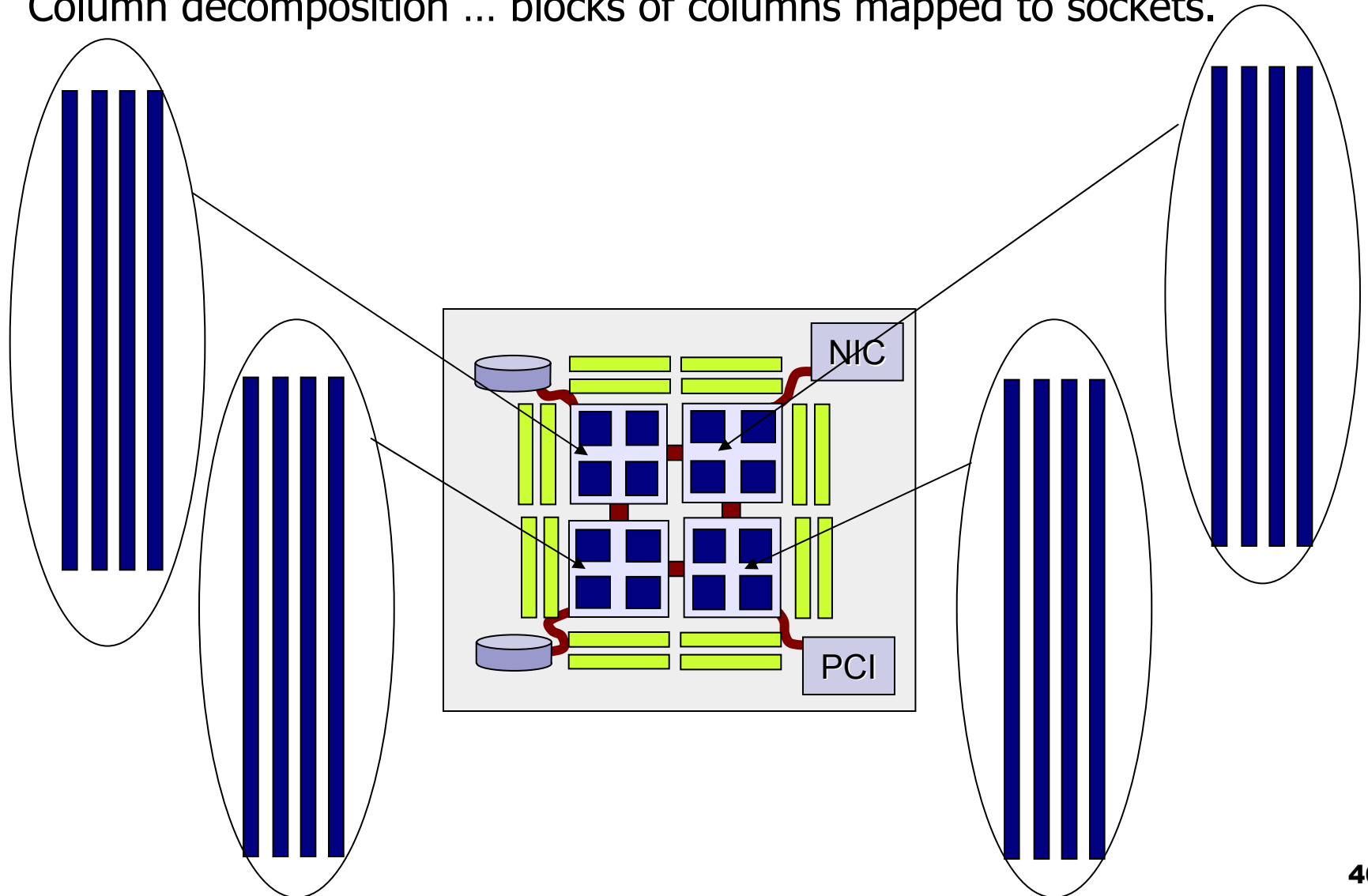
Let's focus on the data decomposition.



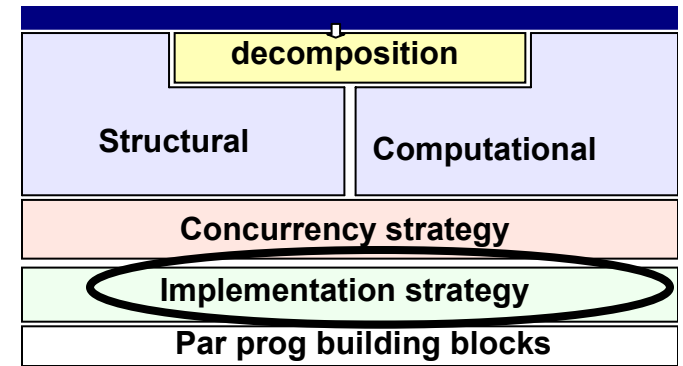
- Name:
 - Geometric Decomposition
- Problem:
 - How can an algorithm be organized around a data structure that has been decomposed into concurrently updatable "chunks"?

Spectral Method: Algorithm Structure

- Column decomposition ... blocks of columns mapped to sockets.

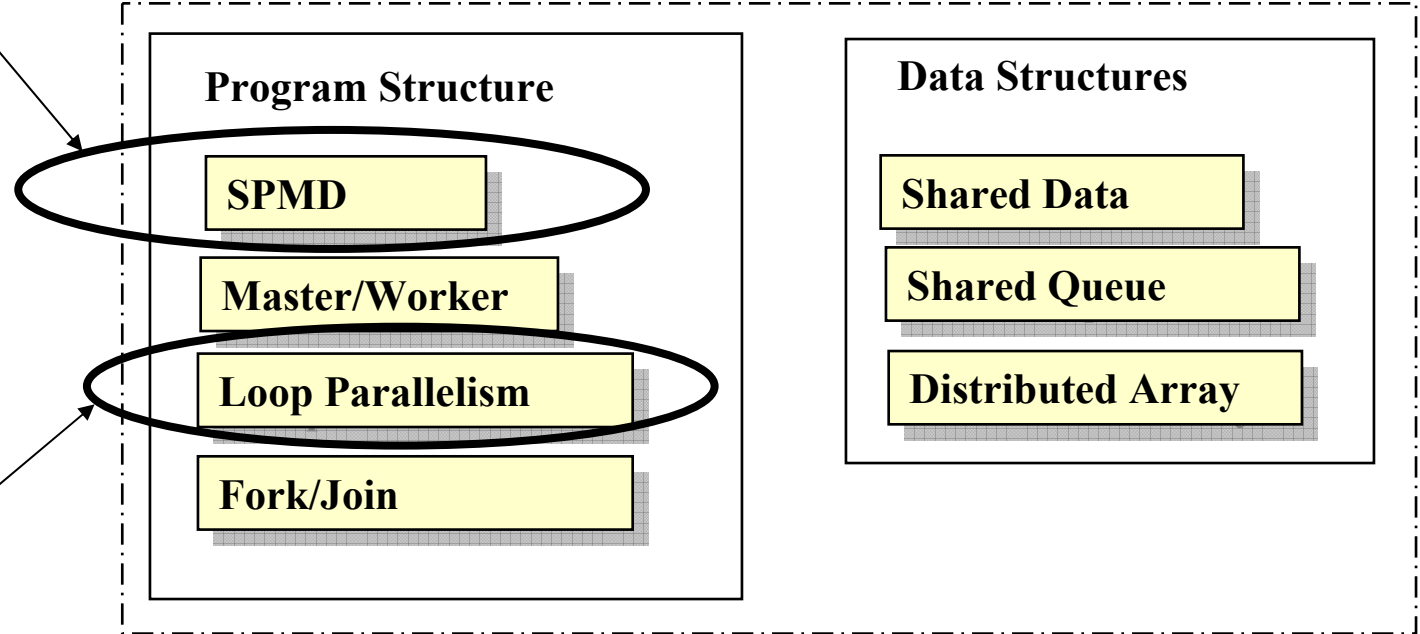


The Supporting Structures Design Space

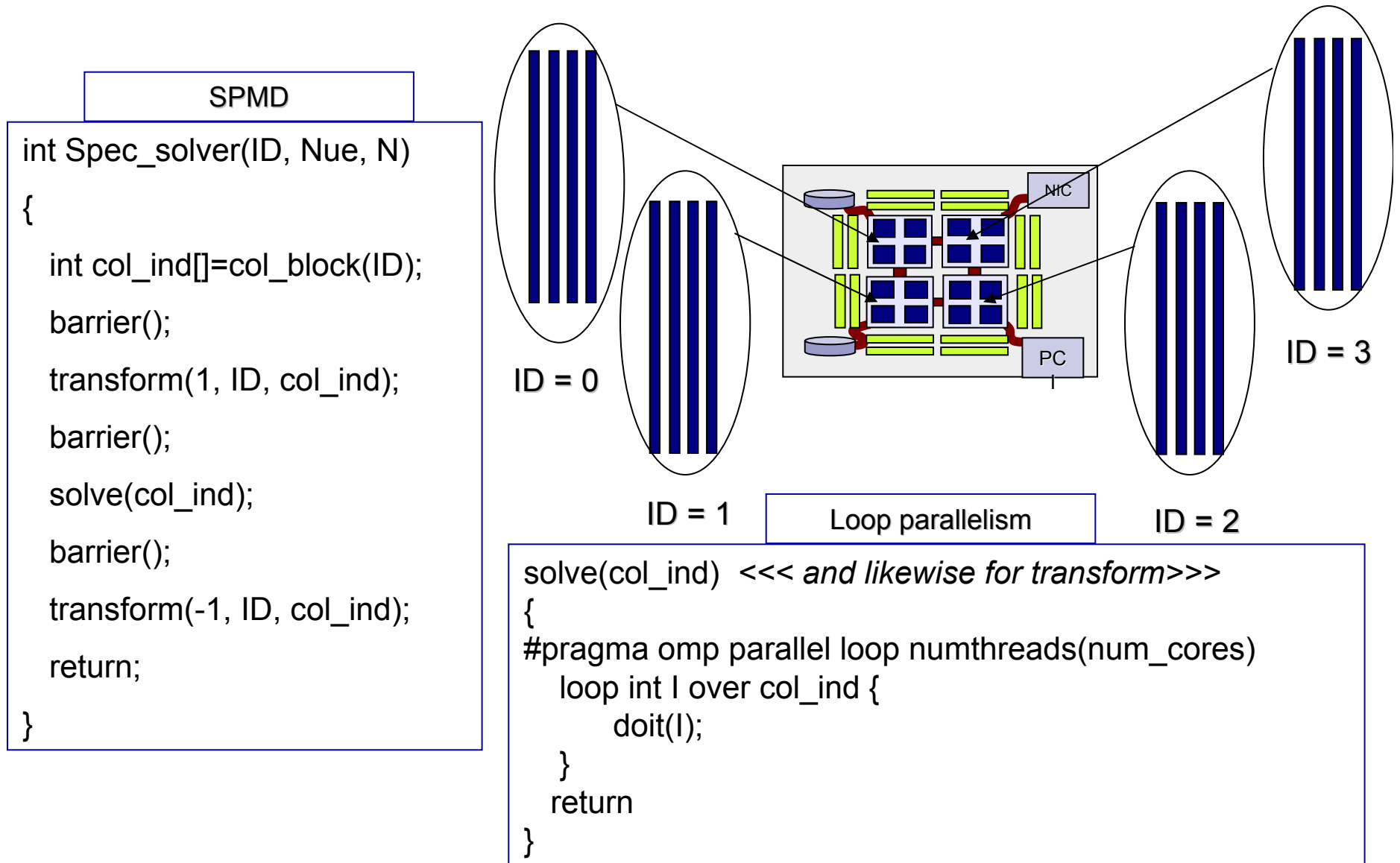


Large column blocks per process managed by MPI ... one process per socket

Exploit cores on a socket with OpenMP

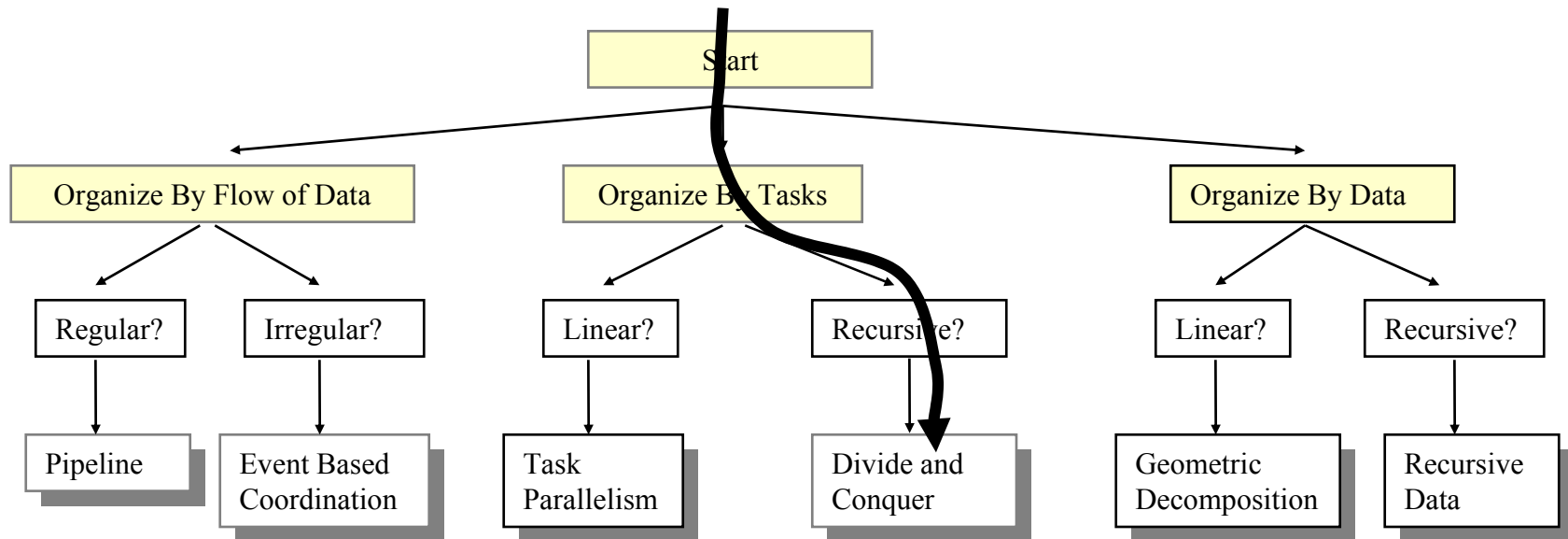


Spectral Method: Supporting Structure



As core counts grow ...

- As core counts grow, you may want to parallelize 1D FFTs.



Win32 Program: Fork/join pattern

```
#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);
}
```

Fork/join:

(1) Package concurrent tasks in a function, (2) fork threads to run each function, (3) join when done, and (4) manage dependencies.

```
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);
}
```

Summary

- Programming education needs to focus on the human angle ... teach people how to “think parallel”.
 - Major research universities should have joint computer science /psychology faculty.
- We are designing our research program around a design pattern language to help us keep this human connection explicit.
- Much work remains to be done:
 - Review the pattern language and help it evolve so it represents a more broad consensus view.
 - Work with us to test this pattern language on real programmers to confirm that “its right”.
 - Derive programming frameworks based on this pattern language ... to help programmers turn their thinking into software.

Join us in working on the pattern language



- ... a workshop for pattern writers and people eager to help them to build a consensus pattern language of parallel programming
 - <http://www.upcrc.illinois.edu/workshops/paraplop09>

References

- [Brooks83] R. Brooks, "Towards a theory of the comprehension of computer programs", *International Journal of Man-Machine Studies*, vol. 18, pp. 543-554, 1983.
- [Guindom90] R. Guindon, "Knowledge exploited by experts during software system design", *Int. J. Man-machine Studies*, vol. 33, pp. 279-304, 1990
- [Hoc90] J.-M. Hoc, T.R.G. Green, R. Samurcay and D.J. Gilmore (eds.), *Psychology of Programming*, Academic Press Ltd., 1990.
- [Petre88] M. Petre and R.L. Winder, "Issues governing the suitability of programming languages for programming tasks. "People and Computers IV: Proceedings of HCI-88, Cambridge University Press, 1988.
- [Petre90] M. Petre, "Expert Programmers and Programming Languages", in [Hoc90], p. 103, 1990.
- [Rist86] R.S. Rist, "Plans in programming: definition, demonstration and development" in E. Soloway and S. Iyengar (Eds.), *Empirical Studies of Programmers*, Norweed, NF, Ablex, 1986.
- [Rist90] R.S. Rist, "Variability in program design: the interaction of process with knowledge", *International Journal of Man-Machine Studies*, Vol. 33, pp. 305-322, 1990.
- [Robertson90] S. P. Robertson and C Yu, "Common cognitive representations of program code across tasks and languages", *int. J. Man-machine Studies*, vol. 33, pp. 343-360, 1990.
- [Wiedenbeck89] S. Wiedenbeck and J. Scholtz, "Beacons: a knowledge structure in program comprehension", In G. Salvendy and M.J. Smith (eds.) *Designing and Using Human Computer interfaces and Knowledge-based systems*, Amsterdam: Elsevier, 1989.