Parallel Programming Principle and Practice

Lecture 11 — Case Study

Parallelizing Backtracking Algorithms

Backtracking Algorithms

- Incrementally builds candidates to solution(s)
  - Must be able to test if candidate is still viable
  - Abandons candidate when it cannot be a possible solution
- Faster than brute force enumeration
  - Can eliminate many invalid configurations quickly
- Solution technique for constraint satisfaction and combinatorial optimization problems
  - Sudoku, Kakuro, Akari, and other logic puzzles
  - Knight's Tour
  - Parsing
  - Knapsack Problem

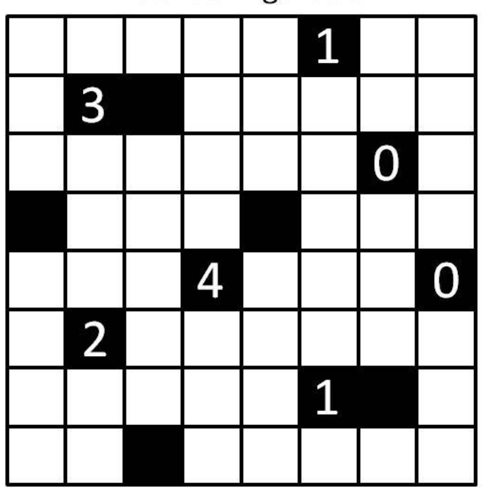
Pseudo-code for Backtracking Algorithm

```
function BT(c)
{
  if (DeadEnd(c)) return;
  if (Solution(c)) Output(c);
  else
    foreach (s = next moves from c) {
      BT(s);
    }
}
```

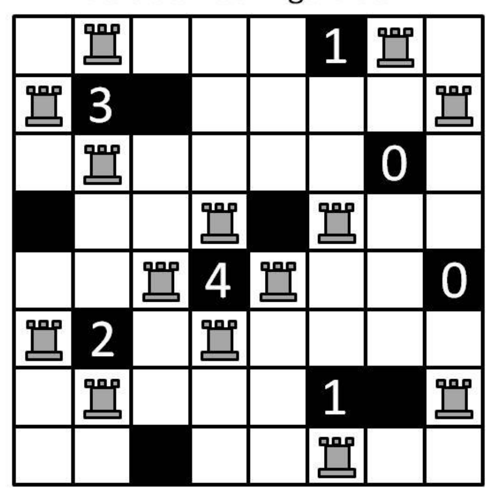
Case Study: Akari

- Logic puzzle from Nikoli
- Goal: Place chess rooks on open squares such that
  - No two rooks attack each other
  - Numbered squares surrounded by specified number of rooks
  - All open squares are "covered" by one or more rooks
  - Black squares block attack of rooks

Initial Configuration



Solution Configuration



http://www.nikoli.com/en/puzzles/bijutsukan/

Akari Algorithm

- **Input:** board size and list of number and black squares
- Sort numbered squares by value; plain black squares after numbered
- Place rooks around all “4” squares
- Using backtracking
  - Get next numbered square in list
  - Try all rook combinations around square, via recursive call
    - “3” square => 4 combinations
    - “2” square => 6 combinations
    - “1” square => 4 combinations
  - If no more numbered squares, compile list of all open squares
  - Using backtracking
    - Try rook in/out next open square from list
    - Solution reached when no more open squares

solveboard() method

- Method `board::solveboard()` implements recursive backtracking

```
void board::solveBoard(int **L)
{
   switch (L[0][2]) {
     case 4:
             placeFour(L); break;
     case 3:
             placeThree(L); break;
     case 2:
             placeTwo(L); break;
     case 1:
             placeOne(L); break;
     case 0:
     case -1: // plain black
             placeOthers(); break;
        }
}
```

placeThree() method

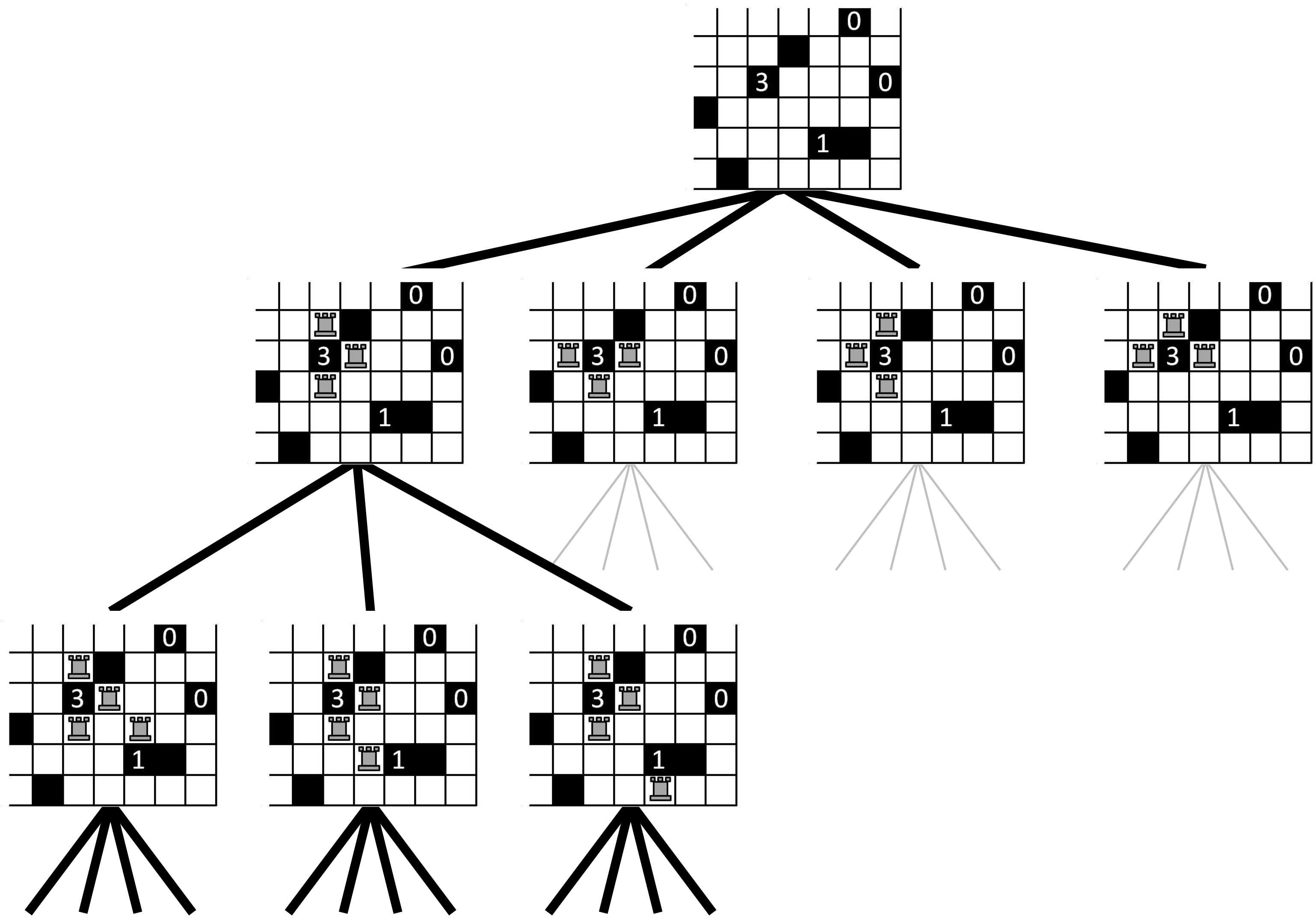
```
void board::placeThree(int **L)
{
//Test around NES faces of island
  board bNES(*this);
  if (bNES.placeNorthRook(L)) {      // add North Rook
    if (bNES.placeEastRook(L))  {    // add East Rook
      if (bNES.placeSouthRook(L)) { // add South Rook
        bNES.solveBoard(&L[1]);
      }
    }
  }

//Test around ESW faces of island
  board bESW(*this);
  if (bESW.placeEastRook(L)) {       // add East Rook
    if (bESW.placeSouthRook(L)) {    // add South Rook
      if (bESW.placeWestRook(L))  {  // add West Rook
        bESW.solveBoard(&L[1]);
      }
    }
  }
. . .
```

```
. . .
//Test around SWN faces of island
  board bSWN(*this);
  if (bSWN.placeSouthRook(L)) {      // add South Rook
    if (bSWN.placeWestRook(L))  {    // add West Rook
      if (bSWN.placeNorthRook(L)) { // add North Rook
        bSWN.solveBoard(&L[1]);
      }
    }
  }

//Test around WNE faces of island
  board bWNE(*this);
  if (bWNE.placeWestRook(L)) {       // add West Rook
    if (bWNE.placeNorthRook(L)) {    // add North Rook
      if (bWNE.placeEastRook(L))  { // add East Rook
        bWNE.solveBoard(&L[1]);
      }
    }
  }
}
```

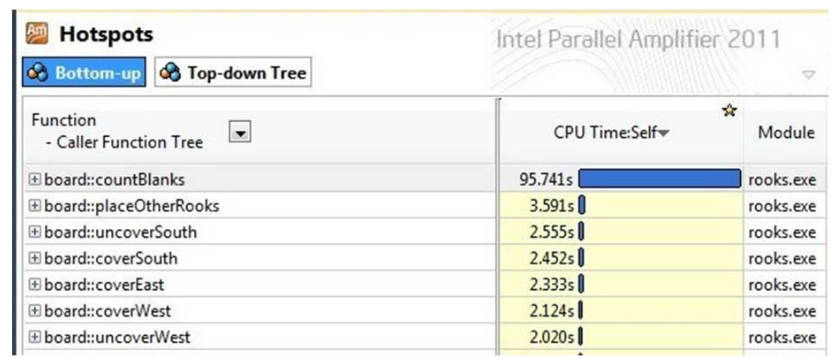
Search Tree – placeThree() example



Where to Parallelize?

- What might hotspot analysis show?
  - If move generation is fast, very little time in BT()
  - Likely DeadEnd() or Solution() (at leafs of search tree) would report most execution time
- Typically not much parallelism to be exploited in checking partial solutions
- Example: Akari solver
  - countBlanks() is 80% of serial time
  - Simple for-loop over board squares
    - 360 squares for dataset used

```
function BT(c)
{
  if (DeadEnd(c)) return;
  if (Solution(c)) Output(c);
  else
    foreach (s = next moves from c) {
      BT(s);
    }
}
```



Akari Execution

- Puzzle (workload) specification
  - 24 x 15 board size
  - 84 black squares (1-“4”, 4-“3”, 14-“2”, 20-“1”, 4-“0”)
  - 21 unique solutions

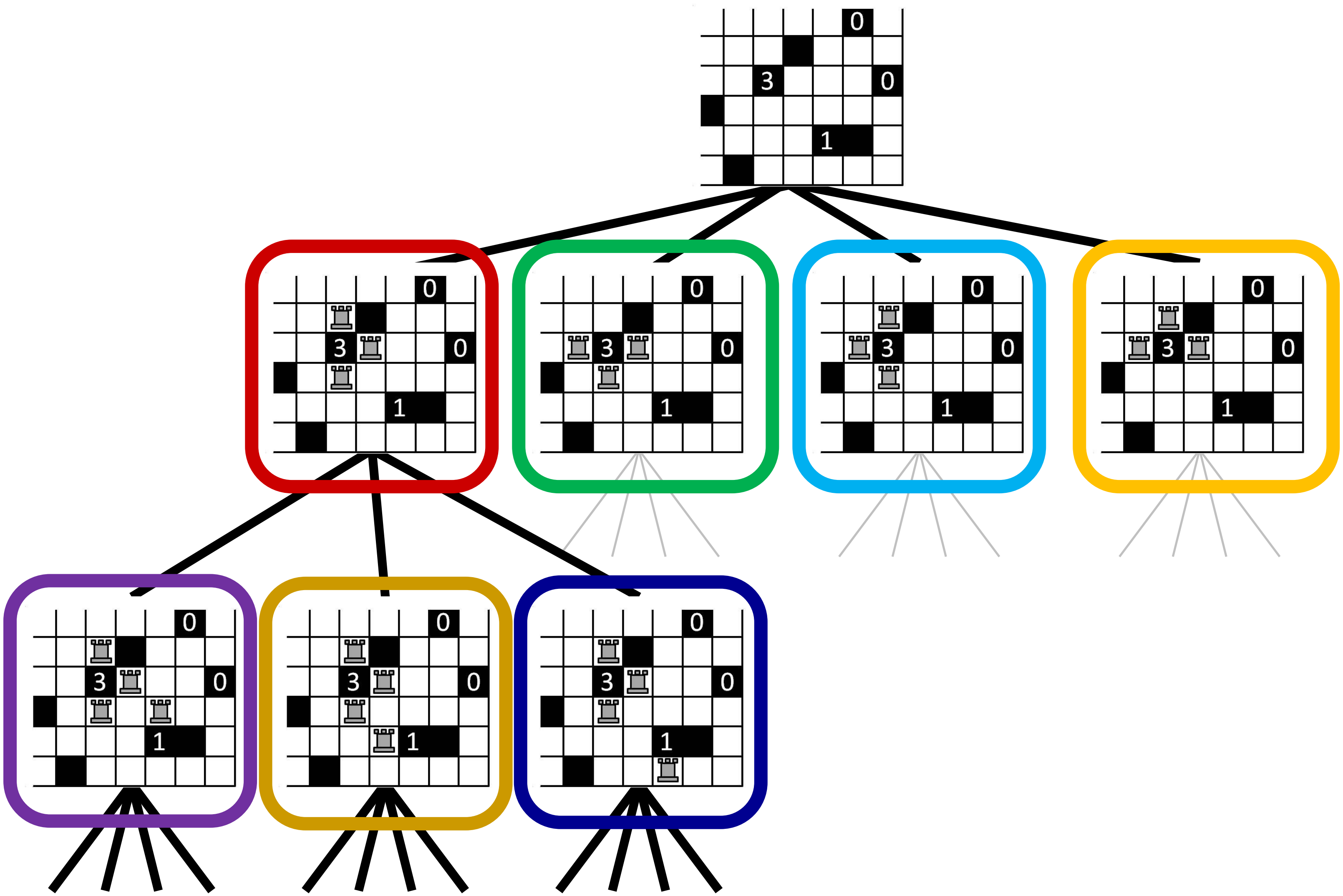
Akari Execution – Hotspot Parallelism

```
int board::countBlanks()
{
// find total number of blanks and non-covered squares
  int i, c = 0;

#pragma omp parallel for reduction(+:c)
  for (i = 0; i < rows*cols; ++i)
    if (B[0][i] == ' ' || B[0][i] == 'n' || B[0][i] == 'N' )
      ++c;
  return c;
}
```

Code version	Time (seconds)	Speedup
Serial	29.557	1.0
countBlanks	36.841	0.8

Backtracking Search – Independent Work

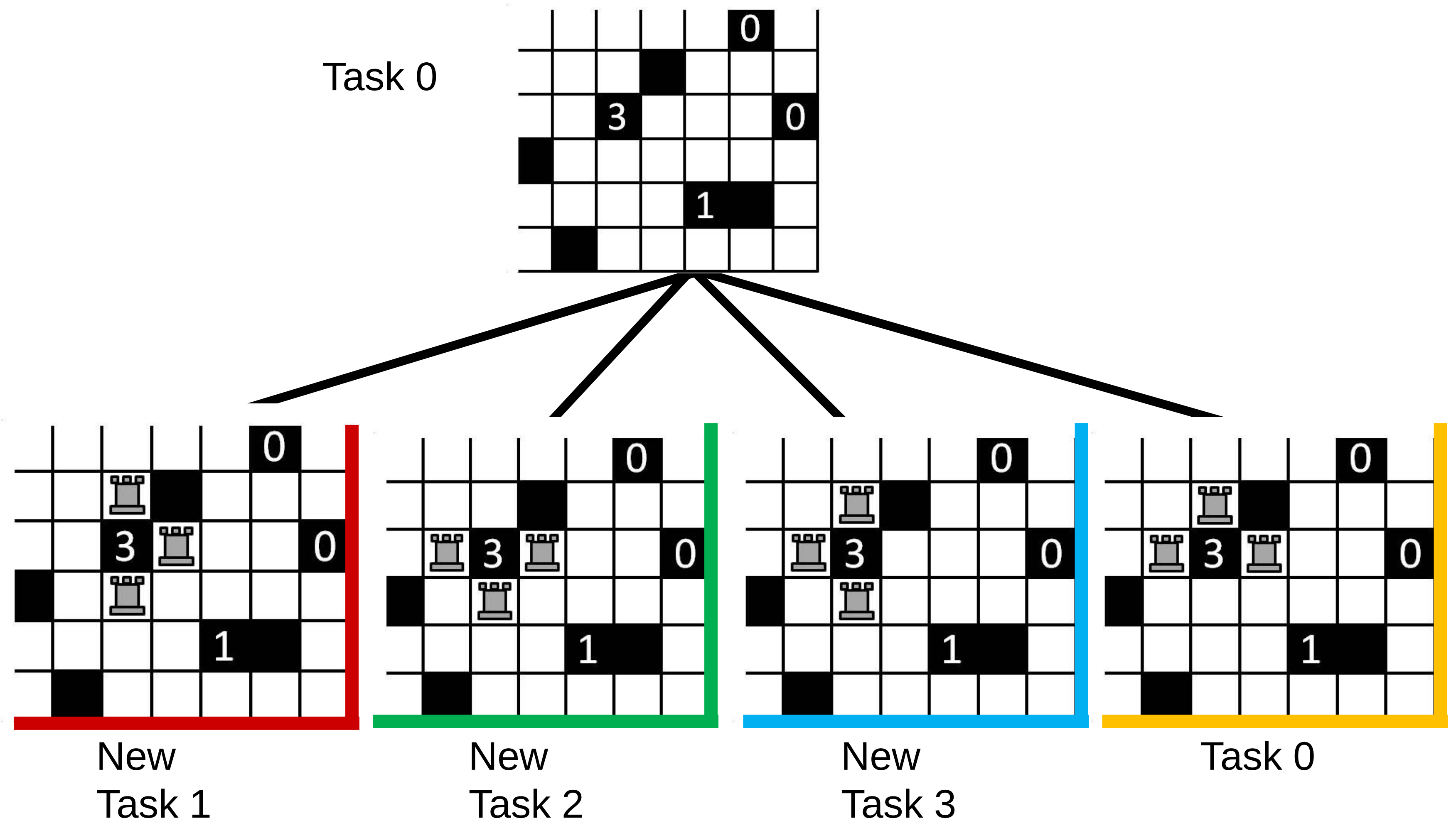


Design #1 for Parallel Backtracking

- Create tasks to explore different parts of the search tree simultaneously

- If a node has children
  - The task generates child nodes (next move)
  - New task created for every child node
  - Generating task could explore one child node itself

Design #1 for Parallel Backtracking



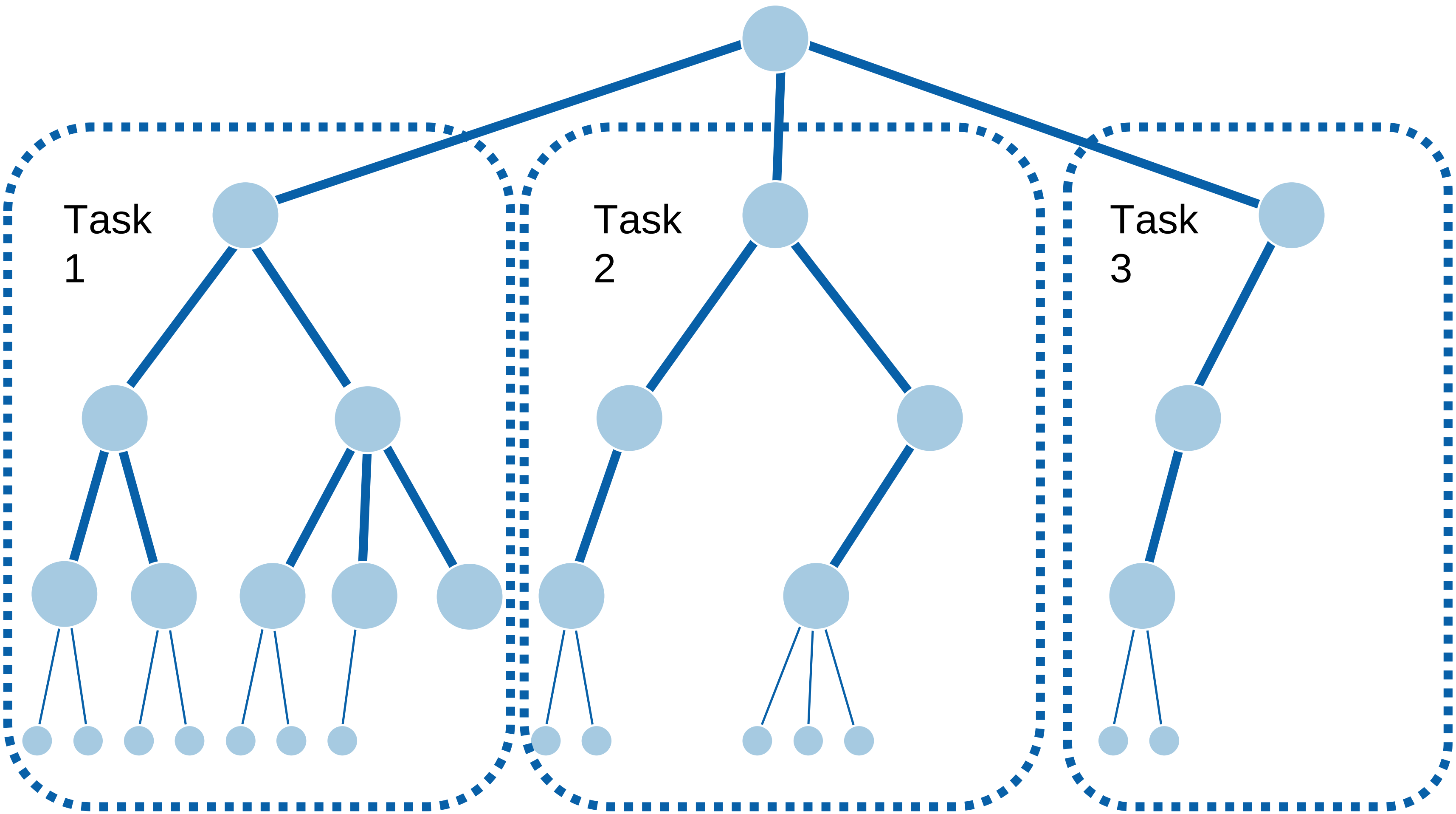
Pros and Cons of Design #1

- Pros
  - Simple design, easy to implement
  - Balances work among tasks
- Cons
  - Too many tasks created
  - Work assigned to task (generate next moves from current state) is small
  - Overhead costs too high
  - Memory usage can be high since a copy of current state needed for each task

Design #2 for Parallel Backtracking

- One task created for each subtree rooted at a particular depth
- Each task sequentially explores its subtree

Design #2 in Action



Pros and Cons of Design #2

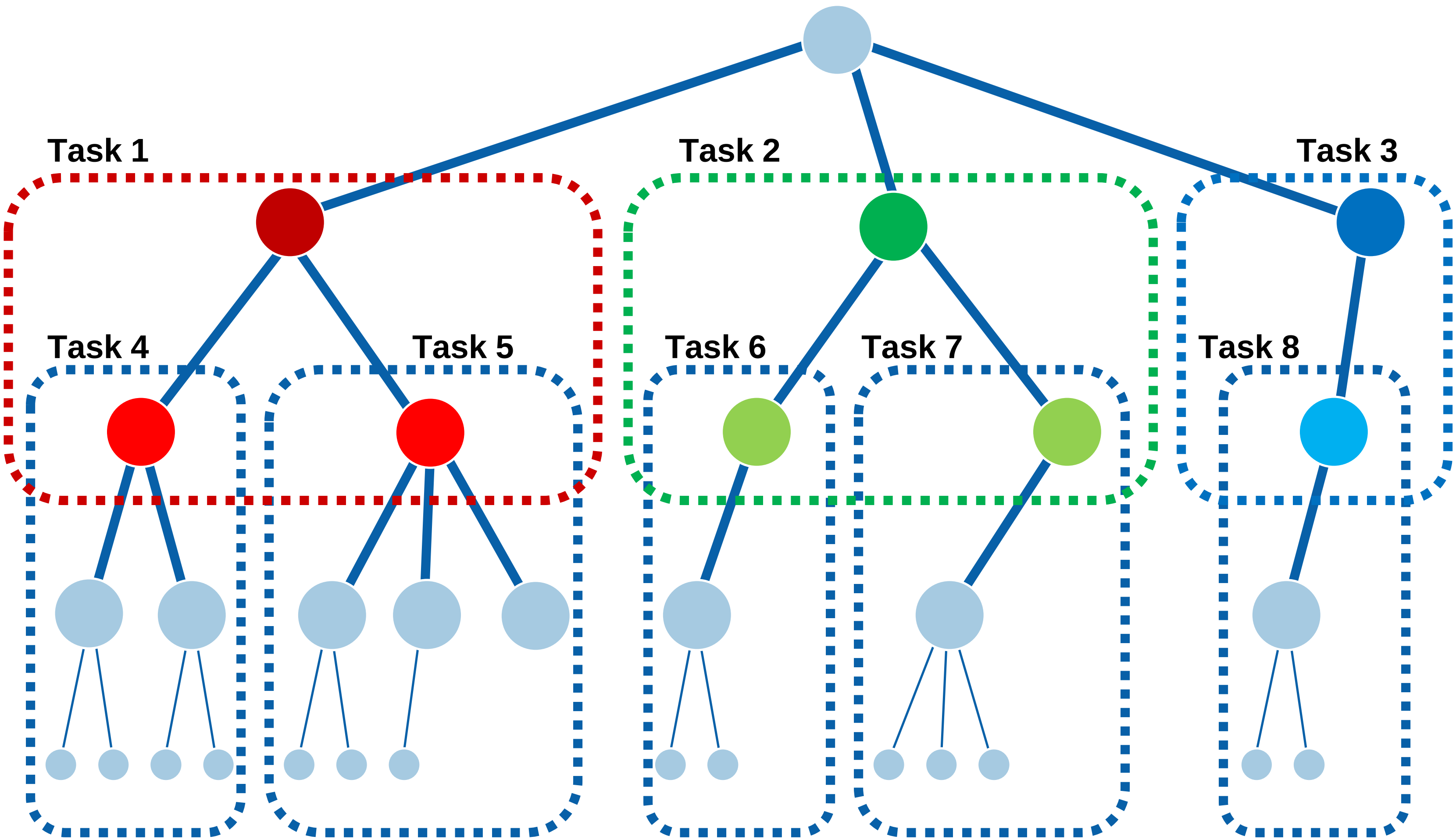
- Pros
  - Task creation/assignment time minimized
  - Minimal memory usage
- Cons
  - Subtree sizes may vary dramatically
  - Some tasks may finish long before others
  - Imbalanced workloads lower efficiency
  - Poor scalability

Design #3 for Parallel Backtracking

- Compromise between the first two designs

- Create a new task for each child node, but only to a certain depth

Design #3 in Action



Pros and Cons of Strategy #3

- Pros
  - Task creation/termination time minimized
  - Workload balance better than strategy #2
- Cons
  - Harder algorithm to code
  - “Best” level of tree to halt new task spawning will depend on workload and number of threads
- Conclusion

  Good compromise between designs 1 and 2

Akari Execution – Design 3

- A) Only "3" square combinations create new tasks
  - Four "3" squares in test case; potentially 81 newly spawned tasks
- B) Use "3", "2", and "1" square combinations to create new tasks
  - 38 squares in test case; potentially ~1.723 sextillion (10^{21}) tasks
  - 19066 actual tasks created with 7296 calling solveBoard()

Code version	**Time (seconds)**	**Speedup**
Design 1	9.925	2.97
Design 2	30.99	0.95
Design 3A	15.015	1.97
Design 3B	1.31	22.56
Serial	29.557	1.0
countBlanks	36.841	0.8

Many Core Challenge

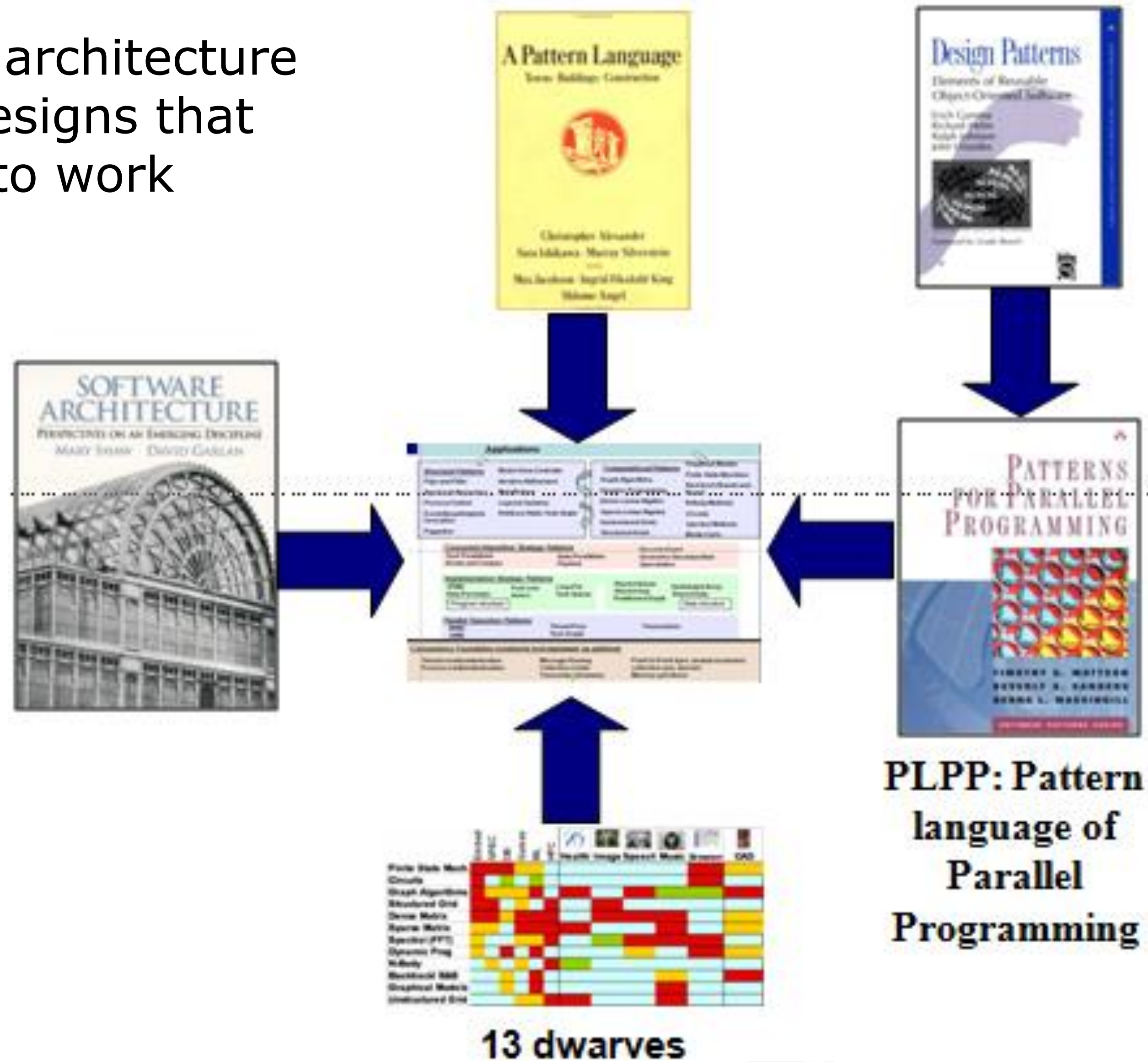
- A harsh assessment
  - We have turned to multi-core chips not because of the success of our parallel software but because of our failure to continually increase CPU frequency

- Result: a fundamental and dangerous (for the computer industry) mismatch
  - Parallel hardware is ubiquitous
  - Parallel software is rare

- Many core challenge
  - Parallel software must become as common as parallel hardware
  - Programmers need to make the best use of <u>all</u> the available resources from within a <u>single</u> program
    - One program that runs close to “hand-tuned” optimal performance on a heterogeneous platform

How Do We Fix Parallel Programming Problem?

- Focus on software architecture and time-tested designs that have been shown to work
- Given a good design, a programmer can create quality software regardless of the language
- Design patterns are a way to put these approaches into writing



Our Pattern Language

<u>Structural Patterns</u>

Pipe and Filter

Agent and Repository

Process Control

Event Based, Implicit Invocation

Puppeteer

Model View Controller

Iterative Refinement

Map Reduce

Layered Systems

Arbitrary Static Task Graph

<u>Computational Patterns</u>

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

<u>Algorithm Strategy Patterns</u>

Task Parallelism

Recursive splitting

Data Parallelism

Pipeline

Discrete Event

Geometric Decomposition

Speculation

<u>Implementation Strategy Patterns</u>

SPMD

Strict data par

Fork/Join

Actors

Master/worker

Graph partitioning

Loop-Par.

BSP

Task-Queue

Program structure

Shared-Queue

Shared Hash Table

Distributed-Array

Shared-Data

Data structure

<u>Parallel Execution Patterns</u>

MIMD

SIMD

Thread Pool

Task Graph

Task-Graph

Data Flow

Digital Circuits

Advancing "program counters"

Msg Pass

Collective Comm

Mutual exclusion

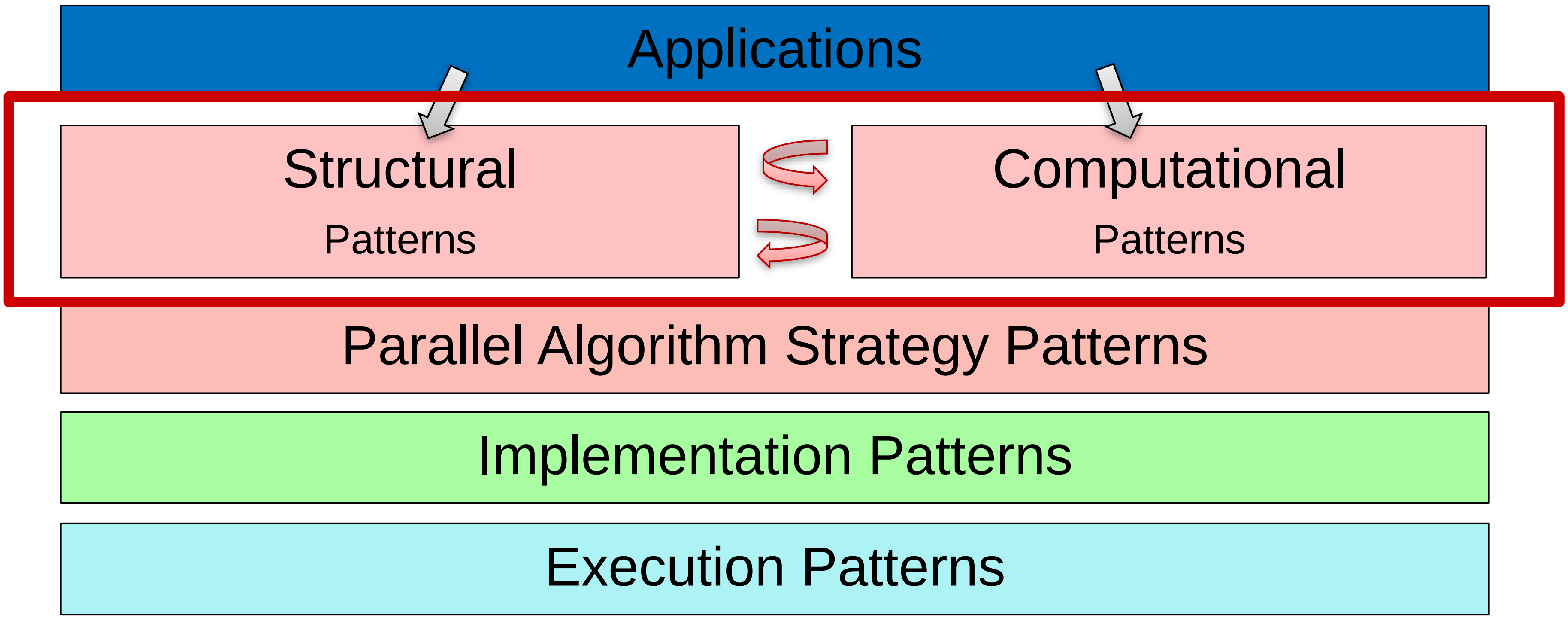
Pt-2-pt sync

Collective sync

Trans Mem

Coordination

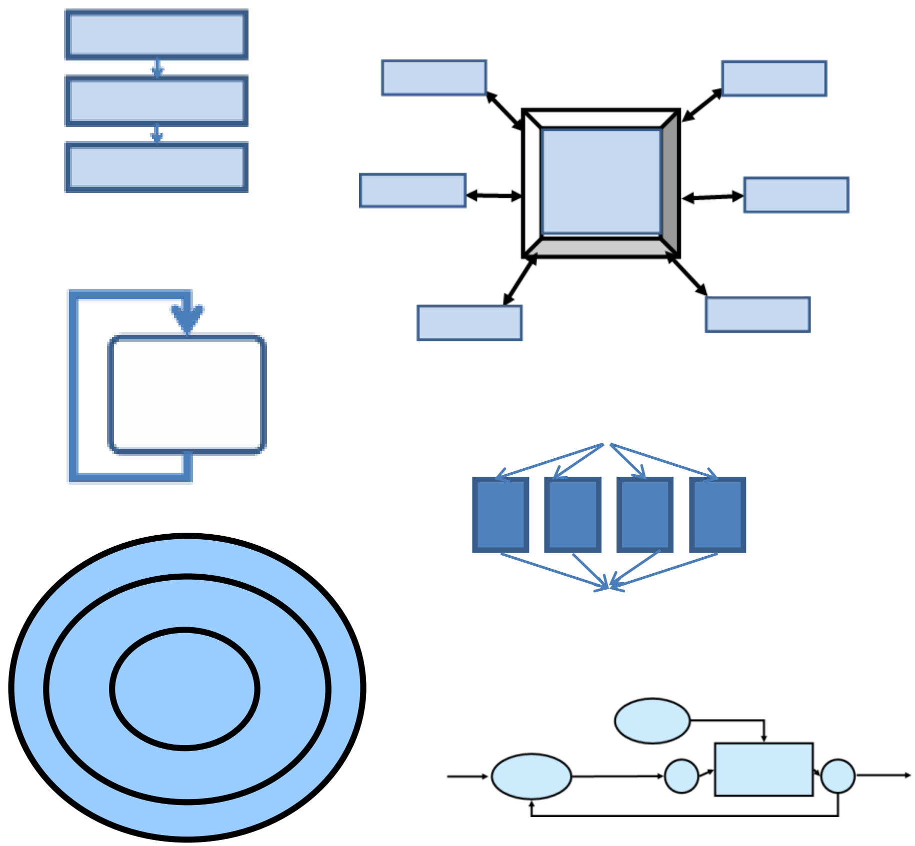
Our Pattern Language



Identify the SW Structure

Structural Patterns

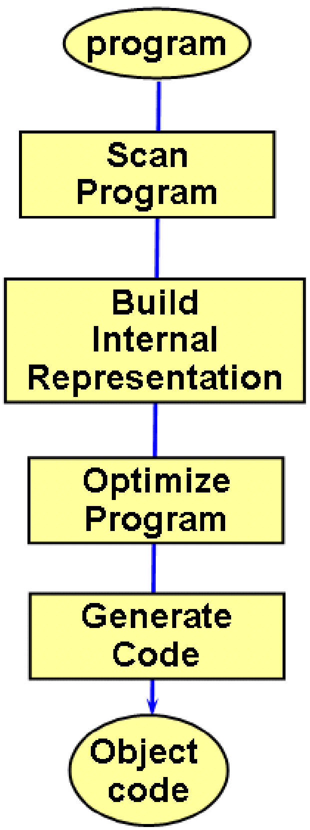
- Pipe-and-Filter
- Agent-and-Repository
- Process-Control
- Event-Based/Implicit-Invocation
- Puppeteer
- Model-View-Controller
- Iterative-Refinement
- Map-Reduce
- Layered-Systems
- Arbitrary-Static-Task-Graph



These define the software structure but *do not describe* what is computed

Example: Pipe and Filter

- Almost every large software program has a pipe and filter structure at the highest level



Compiler

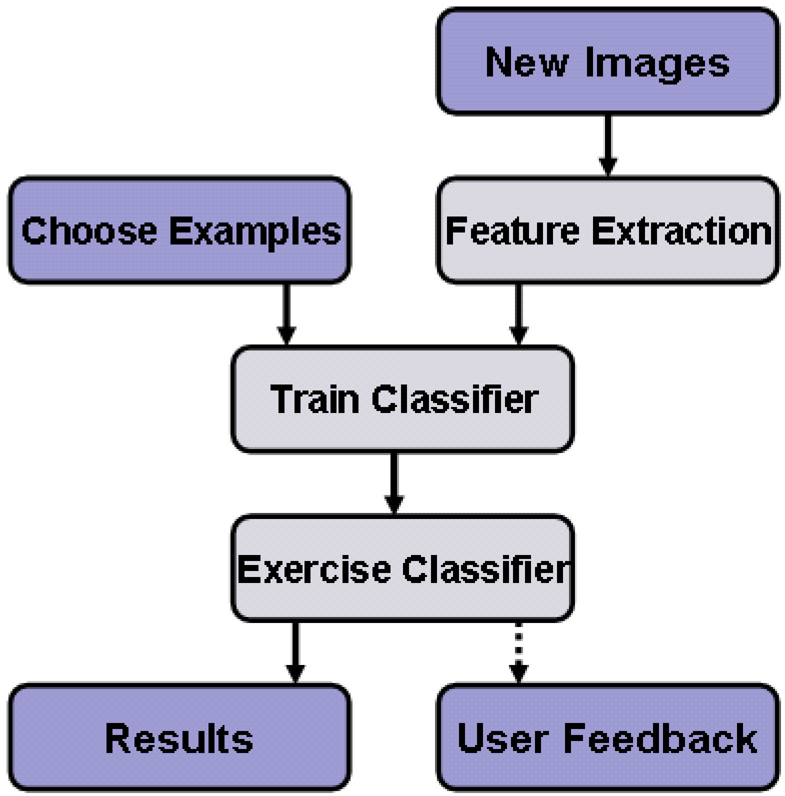
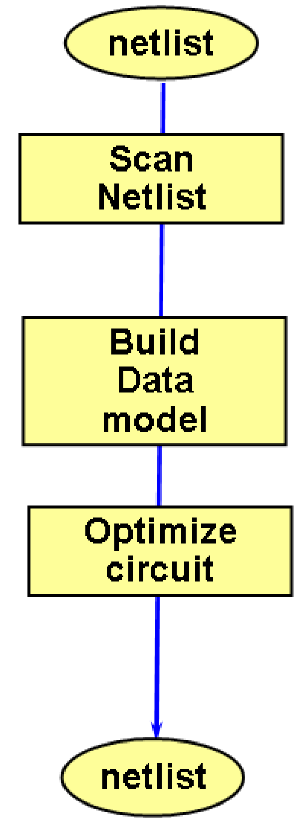
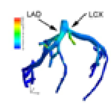
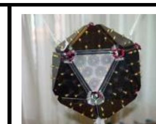


Image Retrieval System



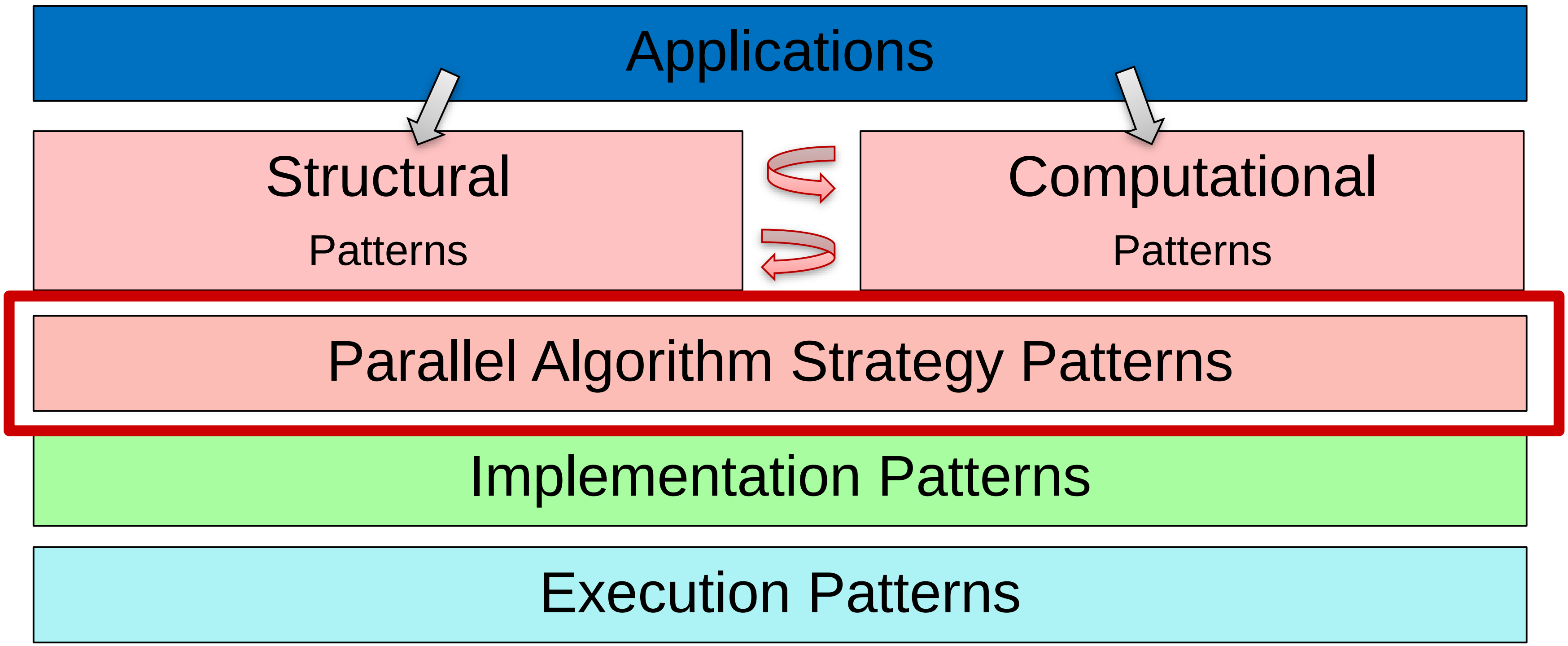
Logic optimizer

Identify Key Computations

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

These define the key computations, but *do not describe* how they are implemented

Parallel Algorithm Strategy



Parallel Algorithm Strategy Patterns

- Parallel Algorithm strategies
  - These patterns define high-level strategies to exploit concurrency within a computation for execution on a parallel computer
  - They address the different ways concurrency is naturally expressed within a problem/application

How does the software architecture map onto parallel algorithms?

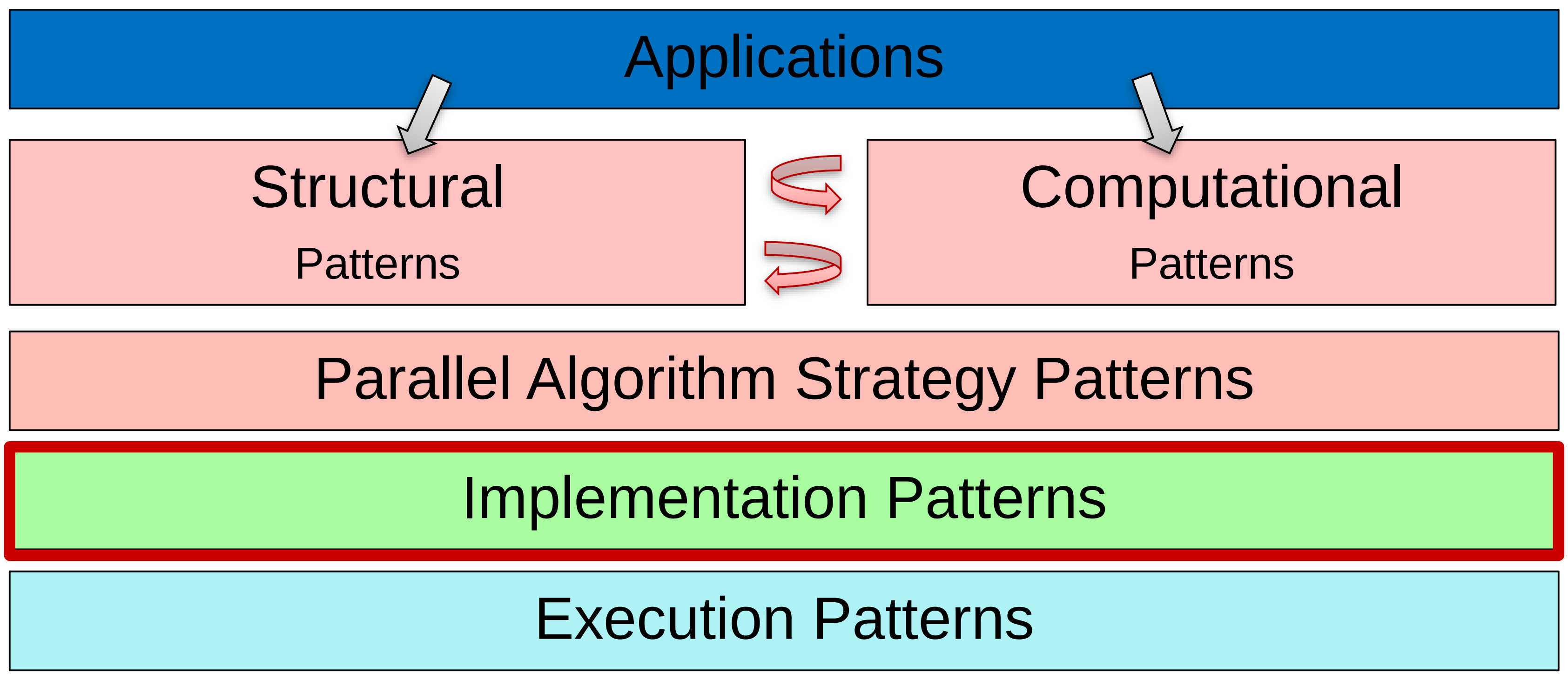
Algorithm Strategy Patterns

Task Parallelism
Recursive splitting

Data Parallelism
Pipeline

Discrete Event
Geometric Decomposition
Speculation

Implementation Strategy



Implementation Strategy Patterns

- Implementation strategies
  - These are the structures that are realized in source code to support (a) how the program itself is organized and (b) common data structures specific to parallel programming

How do parallel algorithms map onto source code in a parallel programming language?

Implementation Strategy Patterns

SPMD
Strict data par

Fork/Join
Actors
Master/worker
Graph partitioning

Loop-Par.
BSP
Task-Queue

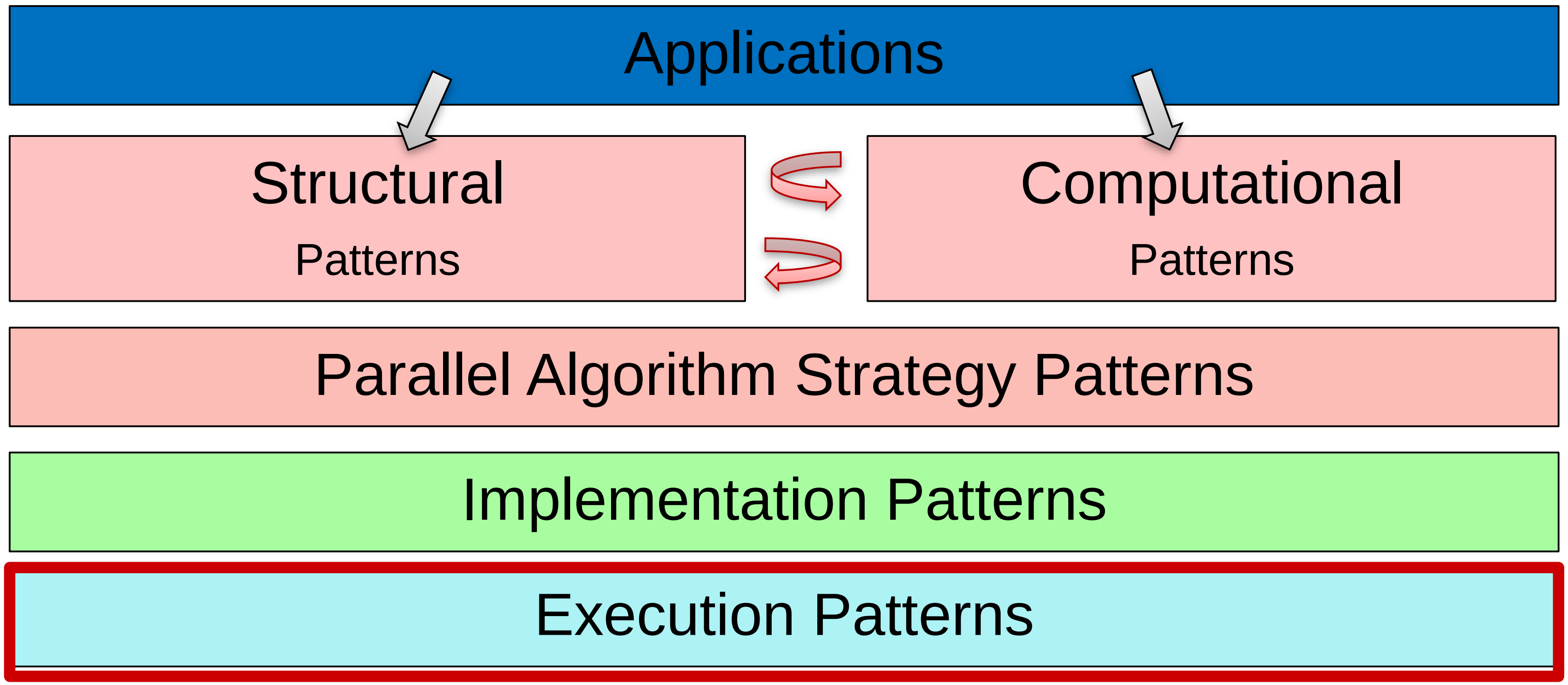
Program structure

Shared-Queue
Shared Hash Table

Distributed-Array
Shared-Data

Data structure

Execution Strategy



Parallel Execution Patterns

- Parallel Execution Patterns
  - These are the approaches often embodied in a runtime system that supports the execution of a parallel program

How is the source code realized as an executing program running on the target parallel processor?

Parallel Execution Patterns

MIMD
SIMD

Thread Pool
Task Graph

Task-Graph
Data Flow
Digital Circuits

Advancing "program counters"

Msg Pass
Collective Comm
Mutual exclusion

Pt-2-pt sync
Collective sync
Trans Mem

Coordination

Compelling Application: Fast, Robust Pediatric MRI

- Pediatric MRI is difficult
  - Children cannot sit still, breathhold
  - Low tolerance for long exams
  - Anesthesia is costly and risky
- Like to accelerate MRI acquisition
  - Advanced MRI techniques exist, but require data- and compute- intense algorithms for image reconstruction
- Reconstruction must be fast, or time saved in accelerated acquisition is lost in computing reconstruction
  - Slow reconstruction times are a non-starter for clinical use

SW Architecture of Image Reconstruction

Pipe and Filter

- Data Parallelism / Fourier Transforms
- Fork-Join
  - Linear Alg. • • • Linear Alg.
- Data Parallelism / Fourier Transforms
- Fork-Join
- Data Parallelism / Fourier Transforms

Iterative POCS Algorithm:

1. Apply SPIRiT Operator:
   $$x_c \leftarrow \sum_j g_{cj} * x_j$$
2. Wavelet Soft-Thresholding
   $$x \leftarrow W S_\lambda \{W^* x\}$$
3. Fourier-space projection
   $$x \leftarrow F(P^T y + P_c^T P_c F^* x)$$

Iter. Refinement / Spectral Method

- Data Parallelism / Convolutions
- Data Parallelism / Wavelet xforms
- Data Parallelism / Fourier xforms

Game-Changing Speedup

- 100X faster reconstruction
- Higher-quality, faster MRI
- This image: 8 month-old patient with cancerous mass in liver
  - 256 x 84 x 154 x 8 data size
  - Serial Recon: 1 hour
  - Parallel Recon: 1 minute
- Fast enough for clinical use
  - Software currently deployed at Lucile Packard Children's Hospital for clinical study of the reconstruction technique

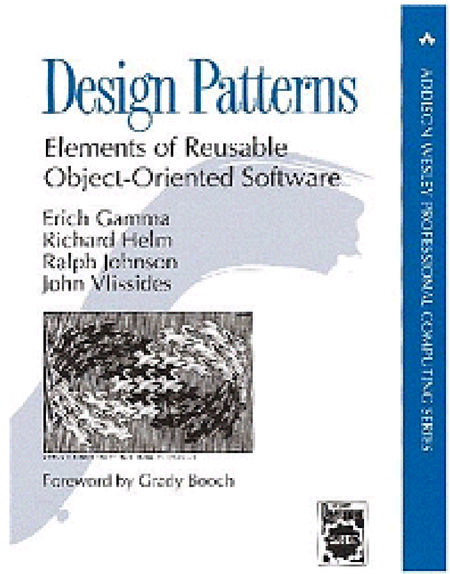
Software Design Patterns in Education: Lessons from History?

Early days OO

Perception: Object oriented? Isn't that just an academic thing?

Usage: specialists only. Mainstream regards with indifference or anxiety.

Performance: not so good.



1994

Now

Perception: OO=programming

Isn't this how it was always done?

Usage: cosmetically widespread, some key concepts actually deployed.

Performance: so-so, masked by CPU advances until now.

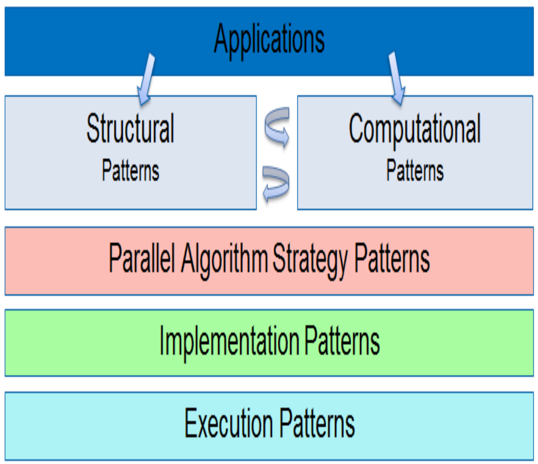
Software Design Patterns in Education: Lessons from History?

Now

Perception: Parallel programming? Isn't that just an HPC thing?

Usage: specialists only. Mainstream regards with indifference or anxiety.

Performance: very good, for the specialists.



Now

Future

Perception: PP=programming

Isn't this how it was always done?

Usage: widespread, key concepts actually deployed.

Performance: broadly sufficient. Application domains greatly expanded.

References

- The content expressed in this chapter comes from
  - Michael Wrinn, Intel Manager, Innovative Software Education