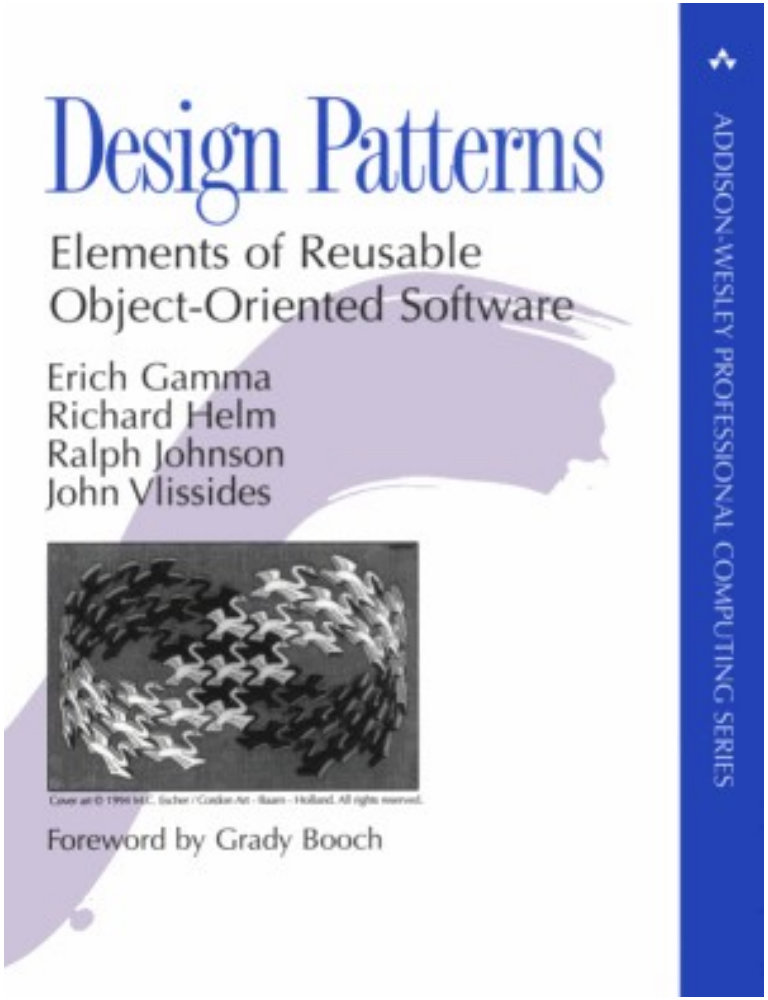
Parallel Programming Patterns: Data Parallelism

Ralph Johnson

University of Illinois at Urbana-Champaign

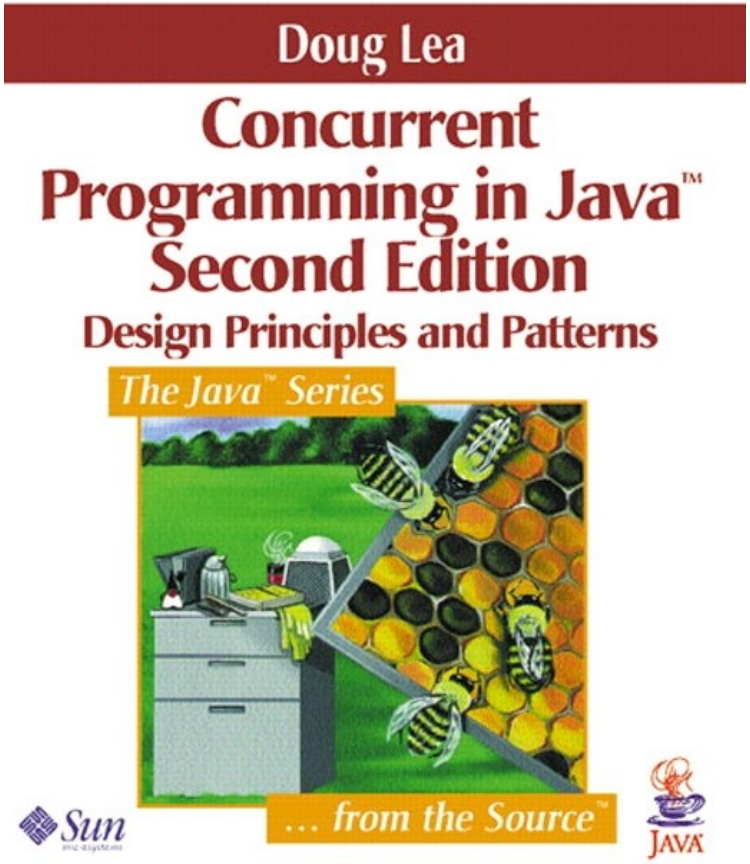
rjohnson@illinois.edu

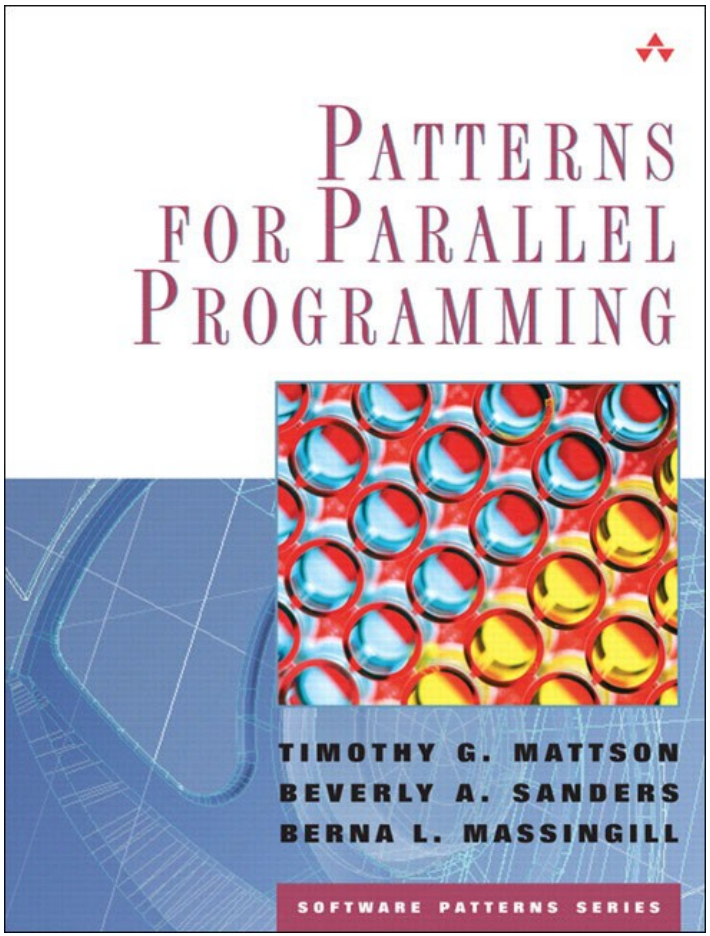
UPCRC Illinois
Universal Parallel Computing Research Center


Design Patterns
Elements of Reusable
Object-Oriented Software
Erich Gamma
Richard Helm
Ralph Johnson
John Vlissides
Foreword by Grady Booch
ADDISON-WESLEY PROFESSIONAL COMPUTING SERIES

Pattern language

- Set of patterns that an expert (or a community) uses
- Patterns are related (high level-low level)


Doug Lea
Concurrent Programming in Java™ Second Edition
Design Principles and Patterns
The Java™ Series
… from the Source™
Sun
JAVA


PATTERNS
FOR PARALLEL
PROGRAMMING
TIMOTHY G. MATTSON
BEVERLY A. SANDERS
BERNA L. MASSINGILL
SOFTWARE PATTERNS SERIES

Making a pattern language for parallelism is hard

- Parallel programming
  - comes in many styles
  - changes algorithms
  - is about performance

Our Pattern Language

- Universal Parallel Computing Research Center
- Making client applications (desktop, laptop, handheld) faster by using multicores

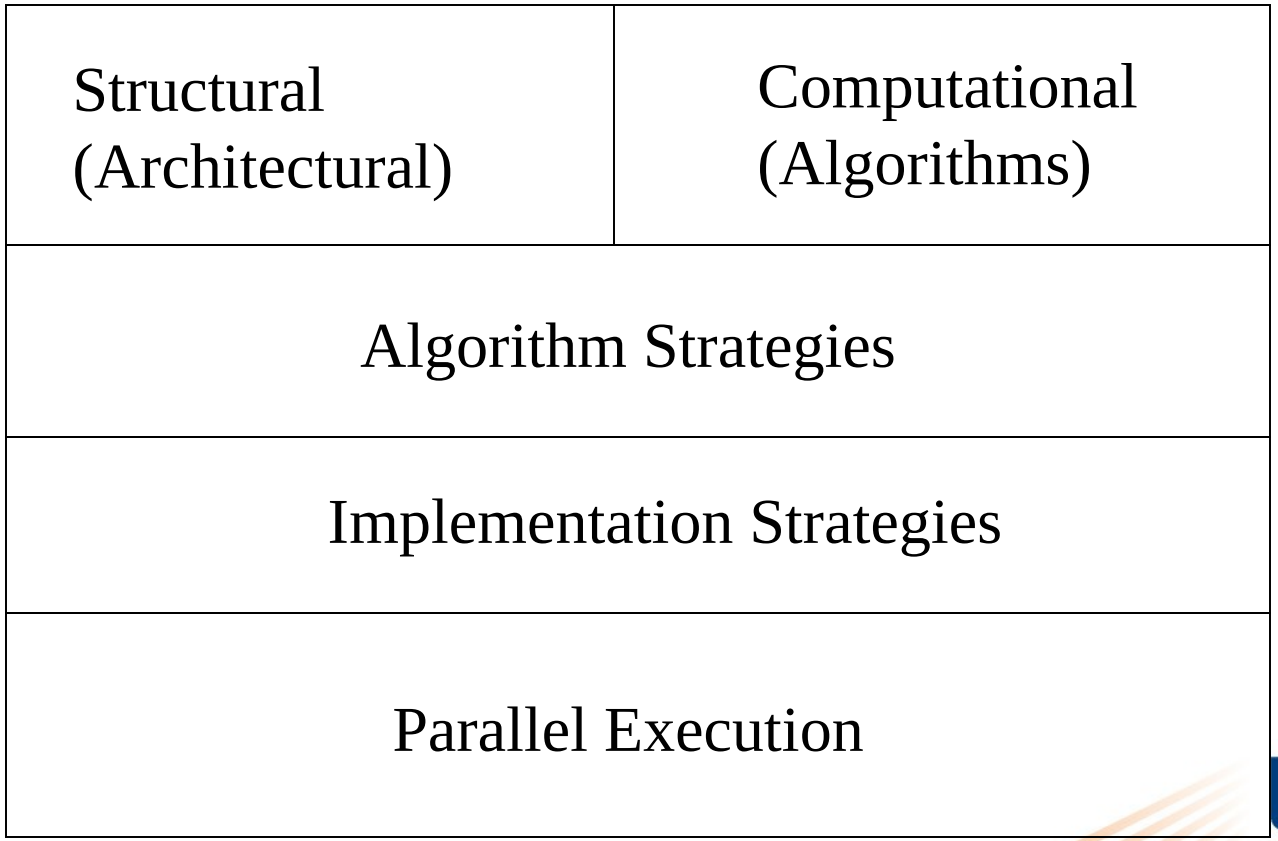
- Kurt Keutzer - Berkeley
- Tim Mattson - Intel

- http://parlab.eecs.berkeley.edu/wiki/patterns
- Comments to rjohnson@illinois.edu

The problem

- Multicores (free ride is over)
- GPUs
- Caches
- Vector processing

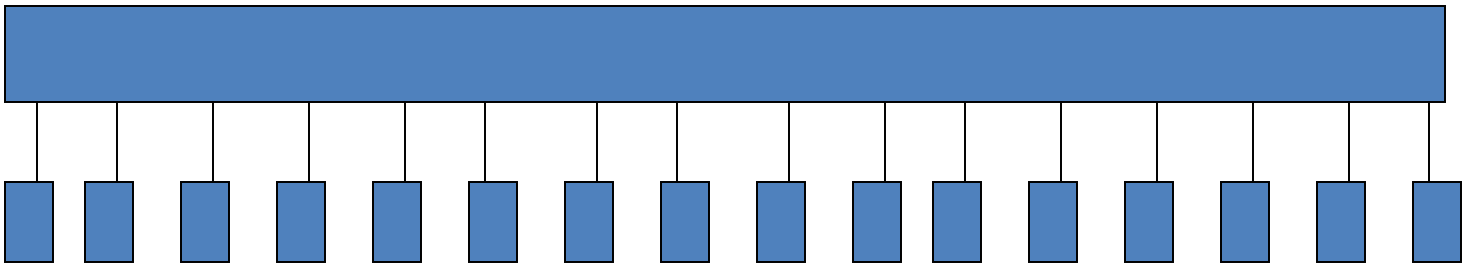
Our Pattern Language



Algorithm Strategies

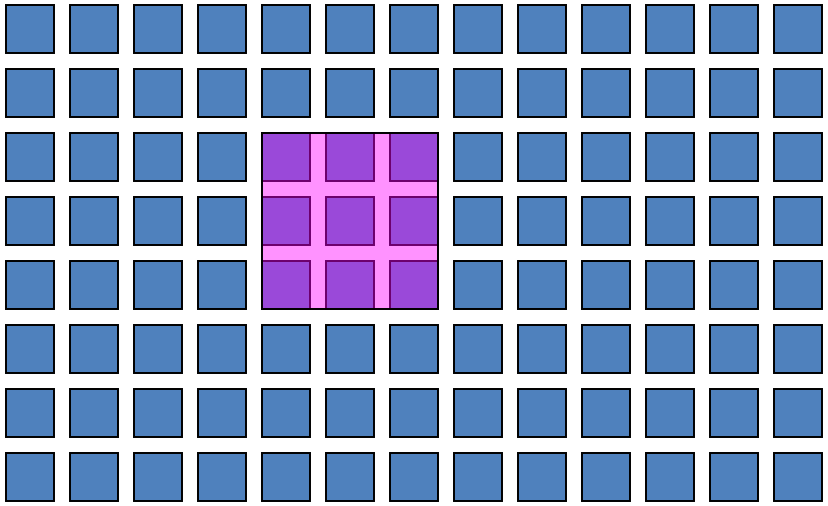
- Task parallelism
- Geometric decomposition
- Recursive splitting
- Pipelining

Task Parallelism



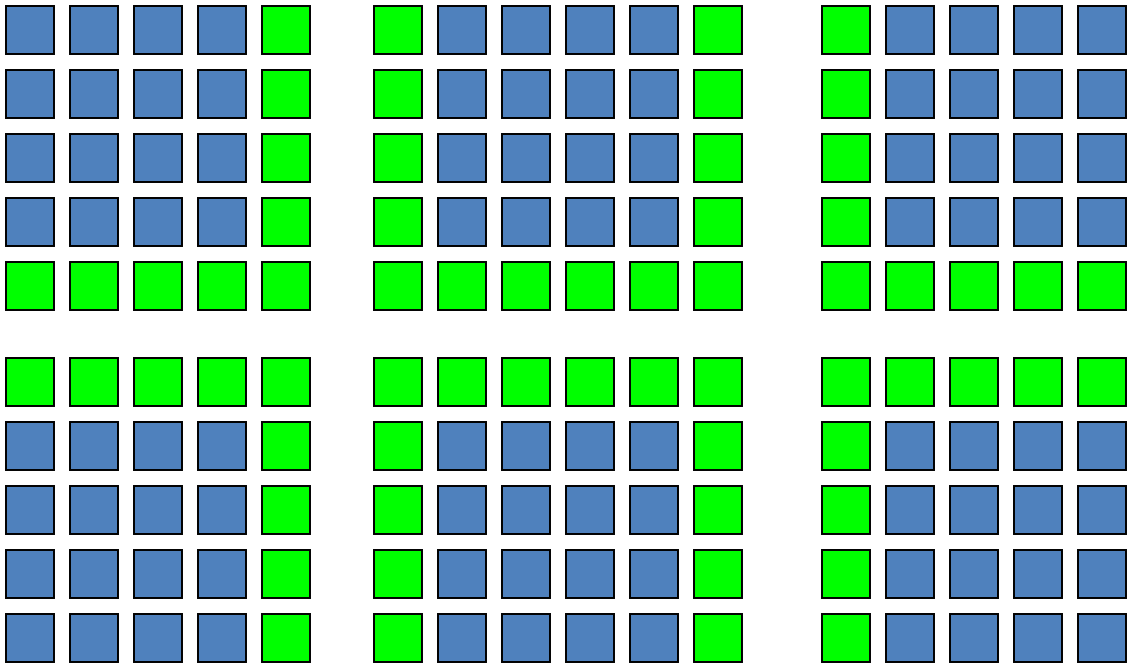
- Communication? As little as possible.
- Task size? Not too big, not too small.
  - Overdecomposition – more than number of cores
- Scheduling? Keep neighbors on same core.

Geometric Decomposition



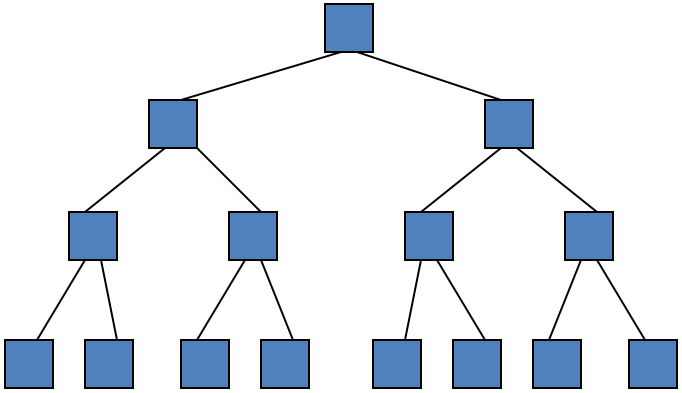
- Stencil

Geometric Decomposition



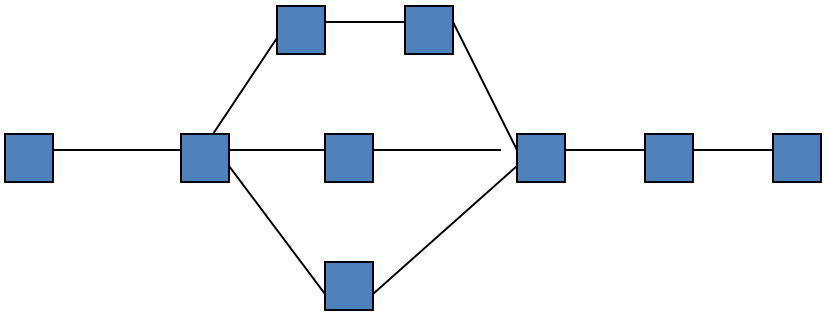
- Ghost cells

Recursive Splitting



- How small to split?

Pipelining



- Bottleneck
- Throughput vs. response time

Styles of parallel programming

- Threads and locks
- Asynchronous messaging – no sharing (actors)
- Transactional memory
- Deterministic shared memory
- Fork-join tasks
- Data parallelism

Fork-join Tasks

- Tasks are objects with behavior "execute"
- Each thread has a queue of tasks
- Tasks run to completion unless they wait for others to complete
- No I/O. No locks.

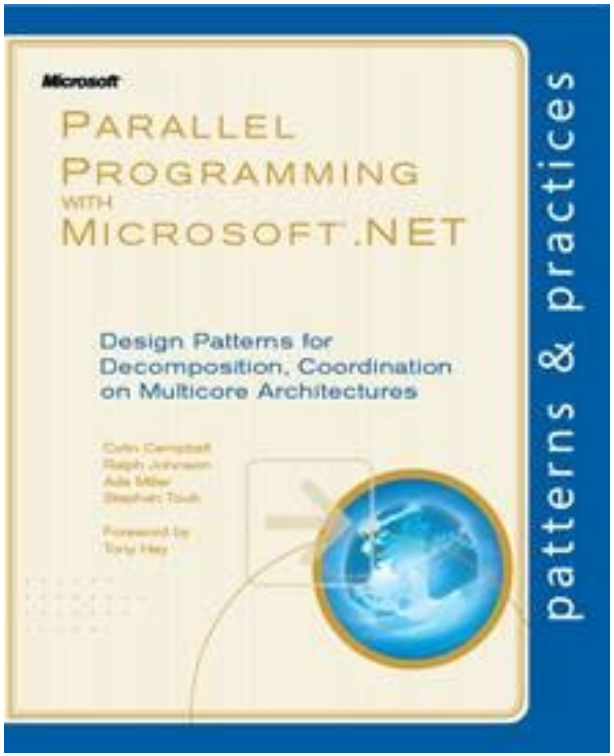
```
void tracerays(Scene *world) {
   for (size_t  i = 0, i>WIDTH, i++) {
   for (size_t  j = 0,  j>HEIGHT, j++) {
   image[i][j] = traceray(i,j,world);
   }
   }
}
```

```
#include “tbb/parallel_for.h”
#include “tbb/blocked_range2d.h”
using namespace tbb;

class TraceRays {
   Scene *my_world;
Public:
   void operator() (const
   blocked_range2d<size_t>& r) {
   …
    }
   TraceRays(Scene *world) {
        my_world = world;
   }
}
```

```
void operator() (const blocked_range2d<size_t>& r) {
   for (size_t  i = r.rows().begin(), i != r.rows().end(), i+
   +) {
     for (size_t  j = j.cols().begin(),  j!=r.cols().end(), j++)
   {
   output[i][j] = traceray(i,j,world);
     }
   }
}

void tracerays(Scene *world) {
parallel_for(blocked_range2d<size_t>(0,WIDTH,8,0,HEI
   GHT,8), TraceRays(world);
}
```

- Parallel reduction
- Lock-free atomic types
- Locks (sigh!)

- TBB: http://threadedbuildingblocks.org
- Java concurrency: http://g.oswego.edu/
- Microsoft TPL and PPL: http://msdn.microsoft.com/concurrency



http://parallelpatterns.codeplex.com/

Common Strategy

- Measure performance
- Parallelize expensive loops
- Add synchronization to fix data races
- Eliminate bottlenecks by
  - Privatizing variables
  - Using lock-free data structures

Data Parallelism

- Single thread of control – program looks sequential and is deterministic
- Operates on collections (arrays, sets, …)
- Instead of looping over a collection, perform “single operation” on it
- No side effects

- APL, Lisp, Smalltalk did something similar for ease of use, not parallelism.

Data Parallelism

- Easy to understand
- Simple performance model
- Doesn’t fit all problems

Operations

- Map – apply a function to each element of a collection, producing a new collection
- Map – apply a function with N arguments to N collections, producing a new collection

Operations

- Reduce – apply a binary, associative function to each element in succession, producing a single element
- Select – apply a predicate to each element of a collection, returning collection of elements for which predicate is true

Operations

- Gather – given collection of indices and an indexable collection, produce collection of values at indices
- Scatter – given two collections, i'th element is element of second collection whose matching element in first has value "i"
- Divide – divide collection into pieces

N-body

Body has variables position, velocity, force, mass

```
for time = 1, 1000000 {
  for b = 1, numberOfBodies {
  bodies[b].computeForces(bodies);
  bodies[b].move();
  }
}
```

```
computeForces(Body *bodies) {
  force = 0;
  for  i = 1, numberOf Bodies {
      force =+ forceFrom(bodies[i])
  }
}
```

```
forceFromBody(Body body) {
 return mass * body.mass * G /
   distance(location, body.location) ^ 2
}
```

```
move() {
  velocity =+ timeIncrement * force /
  mass
  position =+ timeIncrement * velocity
}
```

Data Parallel

computeForces

- map forceFrom to produce a collection of forces
- reduce with + to produce sum

Data parallel

N-body

- map computeForces to produce forces
- map velocity + timeIncrement * force / mass to produce velocities
- map position + timeIncrement * velocity to produce positions
- scatter velocities into body.velocity
- scatter positions into body.position

TBB/java.util.concurrent/TPL

- Each map becomes a parallel loop
- In C++ without closures, each parallel loop requires a class to define operator
- In Java, large library of operators, else you have to define class

Messy, why bother?

- Data parallelism really *is* easier
- Compiler can vectorize easier
- Maps to GPU better

- Better support in other languages
- Will be better support for C++ in the near future
  - Intel Array Building Blocks

Parallel Programming Style

- Data parallism
  - Deterministic semantics, easy, efficient, no I/O
- Fork-join tasking - shared memory
  - Hopefully deterministic semantics, no I/O
- Actors - asynchronous message passing - no shared memory
  - Nondeterministic, good for I/O