An Interactive Approach to Parallelism

Danny Dig
University of Illinois

UPCRC Seminar – Oct 7th
Programming is Program Transformation

“Change is the only guaranteed constant”
Programming is Program Transformation

“Change is the only guaranteed constant”

Successful software undergoes constant change:
- add more features, fix bugs, improve performance

Programming is about change, and we need to change the way we program to better support change

Q1. What are the changes that occur most often in practice?
Q2. How can we automate them to improve programmer's productivity and software quality?
Overview of My Research on Automated Program Transformations

Software Upgrading
ICSE'08
TSE'08
ICSE'07
ECOOP'06
JSME'06
ICSM'05

Software Testing
TSE'10
ASE'09b
ISSTA'08
FSE'07

Sequential ➔ Parallel

OOPSLA'09
ASE'09a
ICSE'09
Impact of My Tools for Automated Program Transformations

ASTGen [FSE'07] – in the testing infrastructure at Sun NetBeans

RefactoringCrawler [ECOOP'06] used at dozens research & industry

Ship with official Eclipse:
- migrating Java 1.4 to 1.5
- record and replay of refactorings

Concurrencer [ICSE'09] ongoing into Eclipse

Software Upgrading

Software Testing

Sequential → Parallel

JavaRefactor (for Jedit): 17,000 downloads
Today's Talk

Sequential → Parallel

Software Upgrading
Software Testing

ReLooper
Immutator
Concurrencer
The Shift to Multicores Demands Work from Programmers

Increased transistor count will be used in coming years to put more cores on a chip

Use parallelism for performance or to enable new applications and services (better QoS, user experience)

Programmers must find and exploit parallelism

A major programming task:
retrofit sequential apps for parallelism

Cannot extract parallelism without user support
Fully Automatic vs. Interactive Parallelization

Good at fine-grained parallelizing (mostly dense-matrix)
- clueless at coarse-grained parallelism

Benefits:
- programmer has domain knowledge
- combine strengths of programmer and tool (search, remember, compute)

Challenges:
- efficient (keep programmer engaged)
- handle complex general-purpose programs

Programmer + Tool >> Tool
Outline

ReLooper: Refactoring to Loop Parallelism
- Transformations to introduce ParallelArray
- Safety analysis
- Evaluation

Immutator: Convert Mutable into Immutable Class

Future Work

Joint work:
C. Radoi,
M. Tarce,
M. Minea,
R. Johnson
Hasn't Loop Parallelism Been Solved a Long Time Ago?

<table>
<thead>
<tr>
<th>Interactive</th>
<th>ReLooper</th>
</tr>
</thead>
<tbody>
<tr>
<td>Scientific Fortran</td>
<td></td>
</tr>
<tr>
<td>Parascope Editor [Kennedy et al.'93], Tiny [Wolfe'91], Sigmacs [Shei and Ganon'90], PAT [Smith and Appelbe'88],</td>
<td></td>
</tr>
<tr>
<td>Intel's ICC/ICPC</td>
<td>Shape Analysis [Berdine et al.'07, Marron et al.'08]</td>
</tr>
<tr>
<td>Fortran Parallelizing Compilers</td>
<td>Points-to Analysis [Cheng et al.'00, Wu et al.'02]</td>
</tr>
<tr>
<td>SUIF [Lam et al. '93]</td>
<td>Memory annotations [Bocchino et al. '09, Hummel et al.'94, Choi et al.'93, Horwitz et al.'89]</td>
</tr>
<tr>
<td>PFC [Kennedy et al. '87], Ptran [Allen et al.'87], Parafrase [Kuck et al.'81]</td>
<td></td>
</tr>
</tbody>
</table>

Numerical | Business Apps |
## Anatomy of Loops in Numerical Apps vs. OO Business Apps

<table>
<thead>
<tr>
<th></th>
<th>Numerical App</th>
<th>OO Business App</th>
</tr>
</thead>
<tbody>
<tr>
<td>Array element</td>
<td>scalar</td>
<td>Heap-allocated objects, with fields to other objects</td>
</tr>
<tr>
<td>Array dimension</td>
<td>multi-dimensional</td>
<td>1-dimension</td>
</tr>
<tr>
<td>Loop nestedness</td>
<td>nested</td>
<td>single</td>
</tr>
<tr>
<td>Access per iteration</td>
<td>a[i], a[i-n], a[i+n]</td>
<td>a[i]</td>
</tr>
<tr>
<td>Analysis question</td>
<td>Do a[i] and a[j] refer same element (does i == j)</td>
<td>Can program reach the same memory location following references through fields?</td>
</tr>
</tbody>
</table>
A Mutable Particle

class Particle {
    double x, y, m;

    public Particle(double x, double y, double m) {
        this.x = x;
        this.y = y;
        this.m = m;
    }

    static Particle createRandom() {
        return new Particle(Math.random(), Math.random(),
                             Math.random() * 100);
    }

    void moveBy(double dx, double dy) {
        this.x = x + dx;
        this.y = y + dy;
    }
}
class ParticleComputation {
    Particle[] bodies;

    void compute() {
        bodies = new Particle[10000000];

        for (int i = 0; i < bodies.length; i++) {
            bodies[i] = Particle.createRandom();
        }

        for (int i = 0; i < bodies.length; i++) {
            bodies[i].moveBy(1, 7);
        }

        Particle cm = new Particle(0,0,0);
        for (int i = 0; i < bodies.length; i++) {
            double cmm = cm.m + bodies[i].m;
            double cmx = (cm.x * cm.m + bodies[i].x * bodies[i].m)/cmm;
            double cmy = (cm.y * cm.m + bodies[i].y * bodies[i].m)/cmm;
            cm = new Particle(cmx, cmy, cmm);
        }
    }
}
Auto-parallelization fails.

Particle.cpp (col. 3): remark: loop was not parallelized: existence of parallel dependence.
Particle.cpp (col. 17): remark: parallel dependence: assumed OUTPUT dependence between (unknown) line 33 and (unknown) line 33.
Particle.cpp (col. 17): remark: parallel dependence: assumed OUTPUT dependence between (unknown) line 33 and (unknown) line 33.
Particle.cpp (col. 17): remark: parallel dependence: assumed OUTPUT dependence between (unknown) line 33 and (unknown) line 33.
Particle.cpp (col. 17): remark: parallel dependence: assumed OUTPUT dependence between (unknown) line 33 and (unknown) line 33.
Particle.cpp (col. 17): remark: parallel dependence: assumed OUTPUT dependence between (unknown) line 33 and (unknown) line 33.
Particle.cpp (col. 17): remark: parallel dependence: assumed OUTPUT dependence between (unknown) line 33 and (unknown) line 33.
Particle.cpp (col. 17): remark: parallel dependence: assumed OUTPUT dependence between (unknown) line 33 and (unknown) line 33.
Particle.cpp (col. 17): remark: parallel dependence: assumed OUTPUT dependence between (unknown) line 33 and (unknown) line 33.
Particle.cpp (col. 17): remark: parallel dependence: assumed OUTPUT dependence between (unknown) line 33 and (unknown) line 33.
Particle.cpp (col. 17): remark: parallel dependence: assumed OUTPUT dependence between (unknown) line 33 and (unknown) line 33.
Particle.cpp (col. 17): remark: parallel dependence: assumed OUTPUT dependence between (unknown) line 33 and (unknown) line 33.
Particle.cpp (col. 17): remark: parallel dependence: assumed OUTPUT dependence between (unknown) line 33 and (unknown) line 33.
Particle.cpp (col. 17): remark: parallel dependence: assumed OUTPUT dependence between (unknown) line 33 and (unknown) line 33.
Particle.cpp (col. 17): remark: parallel dependence: assumed OUTPUT dependence between (unknown) line 33 and (unknown) line 33.
Particle.cpp (col. 17): remark: parallel dependence: assumed OUTPUT dependence between (unknown) line 33 and (unknown) line 33.
Particle.cpp (col. 17): remark: parallel dependence: assumed OUTPUT dependence between (unknown) line 33 and (unknown) line 33.
Particle.cpp (col. 17): remark: parallel dependence: assumed OUTPUT dependence between (unknown) line 33 and (unknown) line 33.
Particle.cpp (col. 17): remark: parallel dependence: assumed OUTPUT dependence between (unknown) line 33 and (unknown) line 33.
Particle.cpp (col. 17): remark: parallel dependence: assumed OUTPUT dependence between (unknown) line 33 and (unknown) line 33.
Particle.cpp (col. 17): remark: parallel dependence: assumed OUTPUT dependence between (unknown) line 33 and (unknown) line 33.
Particle.cpp (col. 16): remark: loop was not parallelized: existence of parallel dependence.
Particle.cpp (col. 16): remark: parallel dependence: assumed ANTI dependence between this line 37 and this line 37.
Particle.cpp (col. 16): remark: parallel dependence: assumed FLOW dependence between this line 37 and this line 37.
Particle.cpp (col. 16): remark: parallel dependence: assumed FLOW dependence between this line 37 and this line 37.
Particle.cpp (col. 16): remark: parallel dependence: assumed ANTI dependence between this line 37 and this line 37.
Particle.cpp (col. 16): remark: parallel dependence: assumed ANTI dependence between this line 37 and this line 37.
Particle.cpp (col. 16): remark: parallel dependence: assumed FLOW dependence between this line 37 and this line 37.
Particle.cpp (col. 16): remark: parallel dependence: assumed FLOW dependence between this line 37 and this line 37.
Particle.cpp (col. 16): remark: parallel dependence: assumed FLOW dependence between this line 37 and this line 37.
Particle.cpp (col. 16): remark: parallel dependence: assumed FLOW dependence between this line 37 and this line 37.
Particle.cpp (col. 16): remark: parallel dependence: assumed ANTI dependence between this line 37 and this line 37.
Particle.cpp (col. 16): remark: parallel dependence: assumed ANTI dependence between this line 37 and this line 37.
public class ParticleComputation {
    Particle[] bodies;

    void compute() {
        bodies = new Particle[10000000];

        for (int i = 0; i < bodies.length; i++) {
            bodies[i] = Particle.createRandom();
        }

        for (int i = 0; i < bodies.length; i++) {
            bodies[i].moveBy(1, 7);
        }

        Particle cm = new Particle(0,0,0);
        for (int i = 0; i < bodies.length; i++) {
            double cmx = cm.x + bodies[i].x;
            double cmy = cm.y + bodies[i].y;
            cm = new Particle(cmx, cmy, cm);
        }
    }
}

public class ParticleComputation {
    ParallelArray<Particle> bodies;

    void compute() {
        bodies = ParallelArray.create(10000000, Particle.class, ParallelArray.defaultExecutor());

        bodies.replaceWithGeneratedValue(new Ops.Generator<Particle>() {
            public Particle op() {
                Particle elt;
                elt = Particle.createRandom();
                return elt;
            }
        });

        bodies.apply(new Ops.Procedure<Particle>() {
            public void op(Particle elt) {
                elt.moveBy(1, 7);
            }
        });

        Particle cm = new Particle(0,0,0);
        cm = bodies.reduce(new Ops.Reducer<Particle>() {
            public Particle op(Particle cm, Particle elt) {
                double cmx = cm.m + elt.m;
                double cmy = cm.y + elt.y * elt.m/cmm;
                double cmy = cm.y + elt.y * elt.m/cmm;
                cm = new Particle(cmx, cmy, cm);
            }
        }, cm);
The Refactoring Process using ReLooper

Particle[] bodies;

ReLooper analyzes loops

ReLooper safe?

no

fix race

ReLooper

yes

Rewrites

Loop → Parallel Operator

Rewrites

Replace Array Accesses

Rewrites

ReLooper

Leave loop sequential

ignore warning

fix race

ReLooper
Outline

ReLooper: Refactoring to Loop Parallelism
- Transformations to introduce ParallelArray
- Safety analysis
- Evaluation

Immutator: Converting Mutable into Immutable Class

Future Work
Transformations for Convert to `ParallelArray`

`ParallelArray` provides parallel operations (e.g., `apply`, `reduce`)
- operations are applied in parallel on the elements
- pool of worker threads, dynamically load-balanced

Change type declaration:

```
Particle[] bodies
```

```
ParallelArray<Particle> bodies
```
Loop Transformations for Convert to ParallelArray

#1. Infer parallel operation
#2. Create element operator

```java
for (int i = 0; i < bodies.length; i++) {
    bodies[i].moveBy(1, 7);
}
```

```java
bodies.apply(new Ops.Procedure<Particle>() {
    public void op(Particle elt) {
        elt.moveBy(1, 7);
    }
});
```
Outline

ReLooper: Refactoring to Loop Parallelism
- Transformations to introduce ParallelArray
  - Safety analysis
- Evaluation

Immutator: Converting Mutable into Immutable Class

Future Work
Safety Analysis

ParallelArray does not provide any synchronization => potential races
  - trust programmer, but verify

1. Check that the loop iterates over all elements of array
   - ParallelArray iterates over all elements, does not guarantee ordering

2. Loop iterations do not have conflicting memory accesses
   - sharing through the Heap

3. Loops do not contain blocking IO
   - writing to a “shared” environment,
   - poor parallel performance
Determine Conflicting Memory Accesses

**Shared Object** = an object that can be reached from different iterations of a loop (transitive definition)

Goal: determine updates to shared objects = writes to their fields

Modeling the heap: object is a memory location modeled by allocation sites and field dereferences

**Data-flow analysis** computes set of shared objects at each statement

Interprocedural analysis propagates the sharing information through function calls

Analysis works on bytecodes – handles library calls
Outline

ReLooper: Refactoring to Loop Parallelism
- Transformations to introduce ParallelArray
- Safety analysis
- Evaluation

Immutator: Converting Mutable into Immutable Class

Future Work
Evaluation

Q1: Does the analysis find problems? Is it fast?

Q2: Does ReLooper save rewriting effort?

Q3: What is the speedup of the refactored code?

Methodology:
- took 7 real-world programs
- used ReLooper to parallelize the computationally intensive loops
- for all the problems reported:
- we checked carefully whether they were genuine and tool did not miss,
- we fixed the races, then re-ran ReLooper to rewrite code
## Results

<table>
<thead>
<tr>
<th>Analysis</th>
<th>Size SLOC</th>
<th>Warnings Race(Real)</th>
<th>I/O</th>
<th>#Analyzed Methods</th>
<th>Time [sec]</th>
<th>Transformation</th>
<th>Speedup</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Changed LOC</td>
<td>Par Loops</td>
</tr>
<tr>
<td>POS Tagger</td>
<td>35K</td>
<td>8(5)</td>
<td>6</td>
<td>995</td>
<td>35</td>
<td>12</td>
<td>1</td>
</tr>
<tr>
<td>Coref</td>
<td>117K</td>
<td>3(0)</td>
<td>7</td>
<td>1147</td>
<td>38</td>
<td>16</td>
<td>1</td>
</tr>
<tr>
<td>Monte Carlo</td>
<td>1127</td>
<td>9(9)</td>
<td>0</td>
<td>106</td>
<td>13</td>
<td>15</td>
<td>1</td>
</tr>
<tr>
<td>Barnes -Hut</td>
<td>540</td>
<td>2(0)</td>
<td>0</td>
<td>57</td>
<td>4</td>
<td>7</td>
<td>1</td>
</tr>
<tr>
<td>Em3d</td>
<td>190</td>
<td>2(0)</td>
<td>6</td>
<td>23</td>
<td>6</td>
<td>16</td>
<td>2</td>
</tr>
<tr>
<td>Lucene</td>
<td>51K</td>
<td>10(9)</td>
<td>0</td>
<td>484</td>
<td>17</td>
<td>44</td>
<td>2</td>
</tr>
<tr>
<td>JUnit</td>
<td>5.7K</td>
<td>2(0)</td>
<td>0</td>
<td>128</td>
<td>12</td>
<td>10</td>
<td>1</td>
</tr>
</tbody>
</table>
Doug Lea (architect of ParallelArray) writes on the Java concurrency mailing list:

“I was very impressed when Danny demoed ReLooper a few weeks ago at OOPSLA. Some of the sample refactored programs are realistic enough that I expect it will be useful to just about anyone interested in exploring these forms of parallelization”

Download:
http://refactoring.info/tools/ReLooper
Outline

ReLooper: Refactoring to Loop Parallelism
- Transformations to introduce ParallelArray
- Safety analysis
- Evaluation

Immutator: Converting Mutable into Immutable Class

Future Work
Immutability Simplifies Parallel Programming

Immutability = transitive state of an object can not be changed after construction

If threads can’t change the state of an object, they can share it without any synchronization:
• No races, no locks, no deadlocks (for that class) = no headaches

Immutable classes are embarrassingly thread-safe

Use immutable classes when programming threads, unless you need mutability
Make Class Immutable

1. Make all fields final
2. Convert mutator methods into factory methods
3. Clone all entering and escaping objects
Adapt Client Code to Use Immutable Class

Class Client{
    ...
    circle.moveTo(new Point(1,2));
    ...
}

Class Client{
    ...
    circle = circle.moveTo(new Point(1,2));
    ...
}

Solution #2:
Adapting the Target Class
- no changes to client code

void moveTo(Point p) {
    value = value.moveTo(p);
}
Immutator's under the Hood Program Analysis

Detecting direct and transitive mutators
- interprocedural method-purity analysis for detecting side-effects on the transitive state of the target class

Detecting entering and escaping objects
- interprocedural class escape analysis for detecting entering/escaping objects that are part of the transitive state of the target class

Analysis works on bytecodes and correctly accounts for library code
Immutator Evaluation

1. Ran Immutator on 346 classes from open-source projects:
   - refactoring is **widely applicable** (33% of classes meet preconditions)
   - refactoring with Immutator is **fast** (average 2.3 sec/refactoring)
   - **improves productivity**: saves programmer from analyzing 84 methods/refactoring and rewriting 42 lines per refactored class

2. Compared Immutator with 11 manual refactorings performed by developers from 6 open-source projects
   - manual refactorings had 24 bugs (forgot to clone subtle entering/escaping objects)
   - confirmed with the developers these were real bugs
   - developers of JDigraph applied our patch
Other Interactive Transformations for Parallelism

Concurrencer [Dig et al. - ICSE'09] supports three refactorings:
- convert int to AtomicInteger
- convert HashMap to ConcurrentHashMap
- parallelize sequential divide-and-conquer via ForkJoinTask framework

Evaluation on 6 widely used OSS shows Concurrencer:
- applies transformations that developers overlooked
- refactored code exhibits good speedup

Currently working to integrate Concurrencer into official Eclipse
## Summary of Current Refactoring Toolset

<table>
<thead>
<tr>
<th></th>
<th>Concurrencrer</th>
<th>ReLooper</th>
<th>Immutator</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Thread-safety</strong></td>
<td>- Convert primitive to Atomic*</td>
<td></td>
<td>Make class Immutable</td>
</tr>
<tr>
<td></td>
<td>- Use ConcurrentHashMap</td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>Throughput</strong></td>
<td>Recursive fork-join parallelism</td>
<td>Loop parallelism</td>
<td></td>
</tr>
<tr>
<td><strong>Scalability</strong></td>
<td>- Convert primitive to Atomic*</td>
<td></td>
<td>Make class Immutable</td>
</tr>
<tr>
<td></td>
<td>- Use ConcurrentHashMap</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

* Atomic* denotes atomic variables or operations that are thread-safe.
Outline

ReLooper: Refactoring to Loop Parallelism

Immutator: Converting Mutable into Immutable Class

Future Work
Future Work

Refactorings for parallelism
- empower average programmer to safely parallelize programs
- new transformations inspired by problems in industry
- porting tools toward tomorrow's paradigm: shared-memory, distributed memory, cloud

Challenges in research on || programming:
- programmer productivity
- program portability
- program performance

My vision:
treat parallel transformations as first-class citizens
Improving Programmer Productivity by Improving Software Maintenance of Parallel Code

**Record** transformations in the IDE

**Explicit documentation** for parallelization
Refactorings leave annotations in code

The IDE can provide two views of the same code
- a simple view for **program understanding**
- a more complex view for tuning
Challenges on Providing Multi-Views

Programmer uses:
- **sequential view** for code understanding, debugging
- **performance view** for performance debugging, profiling

How to maintain consistency between the two views?

How to seamlessly collect and understand edits?

Do edits & transformations in the two views commute?

What kind of annotations are suitable for representing transformations?

How to represent and store programs?
First-class Program Transformations Improves Program Performance

Compose transformations, assemble them in different combinations

Transformations can provide explicit “knobs” for autotuners
Challenges on Improving Program Performance

How to help programmer parallelize code with conflicting memory updates?

- R1: privatize variable
- R2: protect with locks
- R3: use an Atomic* variable
- R4: split loop
- R5: custom transformation

loop with a memory dependency

How to search efficiently in the space of optimization vs. safety, and provide best recommendation?

How to interact seamlessly with other performance tools and the compiler?
First-class Program Transformations Improves Portability

Easy porting to new platforms
- same transformation has several platform-specific implementations
  (e.g., ParallelArray, CloudArray, GPUArray)

  - keep the portable code separate from the platform-specific transformations

Challenges:
- some transformations for a platform can require conflicting restructuring of the code
Conclusions

“Change is the only guaranteed constant”

Interactive program transformations can incorporate various change requirements into existing applications (e.g., parallelism)

Convert “introduce parallelism” into “introduce parallel library”
- still tedious, error- and omission-prone

Automated refactoring is more effective than manual refactoring

Today's brand new sequential programs are tomorrow's legacy programs
- evolution becomes the primary paradigm of software development