

# ReLooper: Refactoring for Loop Parallelism

**Danny Dig**  
**UPCRC Illinois**

**Collaborators:**

**Cosmin Radoi, Mihai Tarce, Marius Minea (UPT),  
Ralph Johnson (UIUC)**

**UPCRC Seminar – Oct 15<sup>th</sup>**



**UPCRC Illinois**  
Universal Parallel Computing  
Research Center

# Introducing Parallelism via Parallel Libraries

**Introduce parallelism for performance**

**Most frequent scenario: retrofit parallelism incrementally**

- **sequence of small steps (refactoring)**
- **always maintain a working, deployable version**

**Programmers often use parallel libraries and frameworks**

- **TPL, TBB, ForkJoinTask, ParallelArray**

**Introduce parallelism problem = introduce parallel library**

# Problems with Using Parallel Libraries

1. Parallel constructs through the library are more verbose than through programming language
2. Library assumes non-interference in parallel constructs

Analysis is **non-trivial**, transformations are **tedious and error-prone**

- Task Parallelism: 5 refactorings => 300 LOC changed, 150 minutes
- Data Parallelism: average of 10 changes per loop

**Goal:** implement tools for **interactive** and **incremental refactoring**

# 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;
    }
}
```

# A Client Class Working with Particles

```
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 = ...y      .....      y .....
            cm = new Particle(cmx, cmy, cmm);
        }
    }
}
```

ParticleComputation.java - TestProject/src/parallelArray

- Update imports
- ParticleComputation
  - Change declaration type
- compute()
  - Replace array assignment
  - Replace sequential loop with parallel operation: replaceWithGeneratedValue
  - Replace sequential loop with parallel operation: apply
  - Replace sequential loop with parallel operation: reduce

ParticleComputation.java



Original Source

```
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 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);
        }
    }
}
```

Refactored Source

```
public class ParticleComputation {
    ParallelArray<Particle> bodies;

    void compute() {
        bodies = ParallelArray.create(10000000, Particle.class, ParallelArr
            .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 cmm = cm.m + elt.m;
                double cmx = (cm.x * cm.m + elt.x * elt.m)/cmm;
                double cmy = (cm.y * cm.m + elt.y * elt.m)/cmm;
                cm = new Particle(cmx, cmy, cmm);
                return cm;
            }
        }, cm);
    }
}
```

# Outline

**ReLooper: Refactoring to Loop Parallelism**

**- ParallelArray framework**

**Transformations to introduce ParallelArray**

**Safety analysis**

**Evaluation**

# ParallelArray in Java

**ParallelArray provides parallel operations (e.g., apply, reduce)**

**Fine-grained parallelism**

- operations are applied on parallel on the elements**
- pool of worker threads**
- load-balanced**

**ParallelArray coming in next version of Java**

# The Refactoring Process using ReLooper

**The programmer selects a target array**

**ReLooper analyzes whether the loops intended for parallelism can be safely parallelized**

- reports potential problems (e.g., data races)
- user can:
  - cancel refactoring, fix problem, re-run ReLooper
  - ignore warnings, continue with transformations

**Programmer confirms changes:**

- replace loop with a parallel operation
- leave loop sequential, replace accesses to indexed elements

**Programmer can override the ReLooper's default**

# Outline

**ReLooper: Refactoring to Loop Parallelism  
- ParallelArray framework**

**Transformations to introduce ParallelArray**

**Safety analysis**

**Evaluation**

# Transformations for Convert to ParallelArray

## Change type declaration:

```
Particle[] bodies
```



```
ParallelArray<Particle> bodies
```

## Change initializer:

```
bodies = new Particle[10000000]
```



```
bodies = ParallelArray.create(10000000, Particle.class,  
                             ParallelArray.defaultExecutor())
```

# Loop Transformations for Convert to ParallelArray

#1. Infer **parallel operation**

#2. Create **element operator**

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



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

# Other transformations

Simple accesses to indexed elements (e.g., `a[i]` → `a.get(i)`)

Other loop styles (e.g., `foreach`)

Loops over `Vector`

- traversal with iterator (`hasNext()`, `next()`)
- most API methods are syntactic sugar for array accesses
- loops with `addElement(obj, i)` changed into `replaceWithMappedIndex` which uses an `AtomicInteger`

# Outline

**ReLooper: Refactoring to Loop Parallelism  
- ParallelArray framework**

**Transformations to introduce ParallelArray**

**Safety analysis**

**Evaluation**

# Safety Analysis

**ParallelArray does not provide any synchronization => potential races**

**- trust but verify**

**1. Check that the loop iterates over **all** elements**

**- first to last element, no skipping**

**2. Loop iterations do not have **conflicting memory accesses****

**- updates to shared & mutable state are not conflicting**

**3. Loops do not contain blocking IO**

# Check that Loop Iterates over All Elements

A parallel operation applies to **all** elements, but the original loop might not iterate all => different semantics

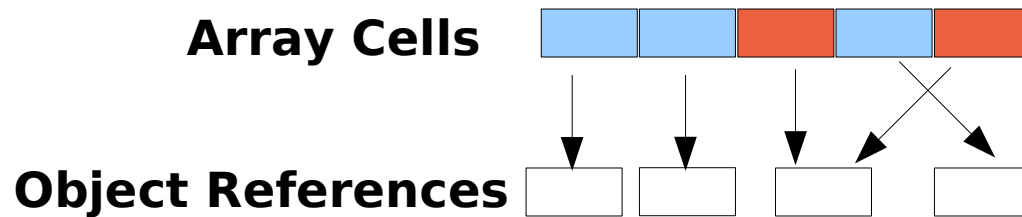
The common case: the loop index variable is also the array index variable

- variable traverses  $0 \rightarrow a.length$
- incremented by one
- each iteration accesses only the loop-indexed element
- also allow reversed order

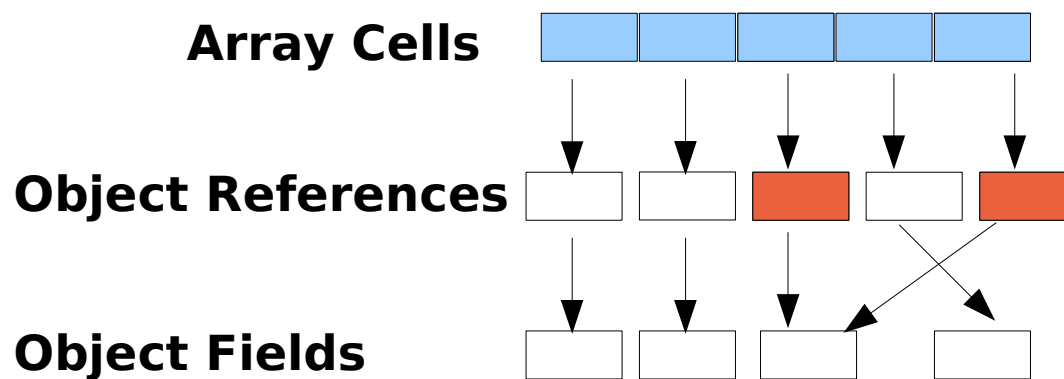
Check that array traversal does not abort intentionally

- loop does not contain `return`, `break`, or `throw`
- we allow `continue`

# Checking whether Loops contain Conflicting Updates



If the array contained aliased references, parallel updates could be conflicting



# Determine Conflicting Memory Accesses

**Parallel section** = instructions in a loop intended for parallelization

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

**Goal: determine updates to shared objects = writes to fields of shared objects**

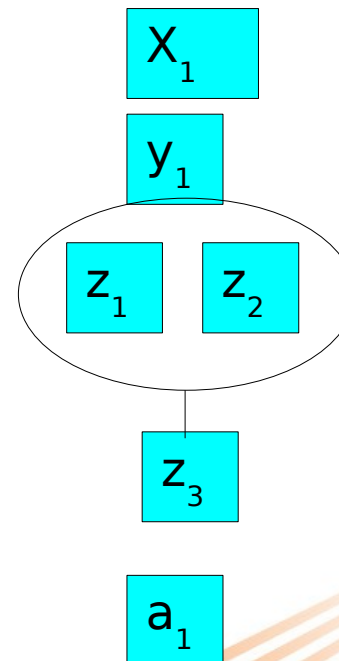
**Analysis has two subgoals:**

1. determine **shared** objects
2. detect **writes** to their fields

# Modeling the Heap

- an object is a memory location modeled by allocation sites and field dereferences
- a variable can only be defined once (like in SSA form)
- collapse all objects that could be stored in a variable into an equivalence class
- model a whole array as one single object

```
x = new Object();  
y = o.f;  
if (condition)  
    z = new ObjectFoo();  
else  
    z = new ObjectBar();  
  
for( ... ){  
    a[i] = new Particle();  
}
```



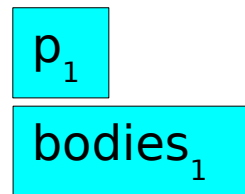
# Data-flow Analysis to Determine Shared Objects

Flow-sensitive (but path-insensitive), context-sensitive,  
field-insensitive (if  $o.f$  is marked as shared  $\Rightarrow$   $o$  marked as shared)

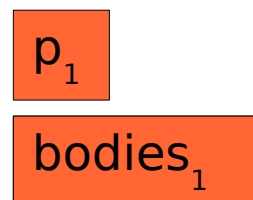
## Intraprocedural part:

- visit instructions in the order of CFG
- evaluate **different** transfer functions for instructions in parallel/sequential sections

```
for (...) //parallel section  
  p = new Particle();  
  bodies[i] = p;
```



```
p = new Particle();  
for (...) //parallel section  
  bodies[i] = p;
```



# Interprocedural Analysis to Determine Shared Objects

Every method has its set of shared objects,  $S_m$

For InvokeMethod node:

- bind actual and formal arguments (initialize the method's set of shared object)
- invoke the intraprocedural data-flow analysis for method body
- propagate the set of shared objects back into the calling context

Analysis **memoizes** the result for a method with a particular sharing pattern of the input parameters

```

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

```

```

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

    [[
      bodies[i]=Particle.createRandom();
    ]]


    this.moveAll(bodies);

    Particle cropper = new Particle();
    [[
      if (i % 13)
        bodies[i] = cropper;
    ]]

    this.moveAll(bodies);
  }
}

```

```

void moveAll(Particle[] arr) {
  [[
     arr[i].moveBy(0,7);
  ]]
}

```

```

this.moveBy{}
  {}
  {}

```

```

this.compute()
{this}

```

```

Particle.createRandom
{this}

```

```

this.moveAll(bodies)
{this}

```

```

{this,cropper}

```

```

{this,cropper,bodies}
{this,cropper,bodies}

```

```

this.moveAll(bodies)
{this,cropper,bodies}

```

```

this.moveAll(arr)

```

```

arr[i].moveBy(0,7)

```

```

this.moveBy{}
  this.x =Conflict
  this.y =Conflict

```

```

this.compute()
{this}

```

```

Particle.createRandom
{this}

```

```

this.moveAll(bodies)
{this}

```

```

{this,cropper}

```

```

{this,cropper,bodies}
{this,cropper,bodies}

```

```

this.moveAll(bodies)
{this,cropper,bodies}

```

```

this.moveAll(arr)

```

```

arr[i].moveBy(0,7)

```

# Detecting IO operations

**Loops that contain blocking IO result in poor performance**

- documentation warns, but the framework does not check

**ParallelArray framework is implemented using lightweight tasks**

- element operations are encapsulated as tasks
- tasks are mapped to worker threads
- worker threads are mapped to cores (threads = # of cores)
- a blocked task => blocked thread => core stays lazy

**As it visits the CFG/CG in parallel sections, the analysis catches method invokes from a black-list of Java IO**

# Outline

**ReLooper: Refactoring to Loop Parallelism**

**Transformations to introduce ParallelArray**

**Safety analysis**

**Evaluation**

# 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 5 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

	Size SLOC	Analysis				Transformation			SpeedUp	
		Warnings	#Analyzed IO	Time [sec]	Changed LOC	#Loops Parallel	Seq	1-core	2-core	
		Mem(Genuine)								Methods
POSTagger	35810	4(4)	8	354	25	12	1	1	0.97	1.09
Coref	117660	14(0)	6	257	21	16	1	2	0.97	1.32
MonteCarlo	1127	3(3)	1	83	8.6	15	1	1	0.99	1.42
Barnes-Hut	540	2(2)	0	14	1.7	13	1	1	0.98	1.7
Em3d	189	1(1)	0	22	6.1	52	6	0	0.98	1.35

# Conclusions

**Introducing concurrency is hard**

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

**Automated refactoring is more effective than manual refactoring**

**<http://refactoring.info/tools/Concurrencer>**

**<http://refactoring.info/tools/ReLooper>**

**Future work:**

- support more refactorings,  
e.g., introduce pipeline parallelism**