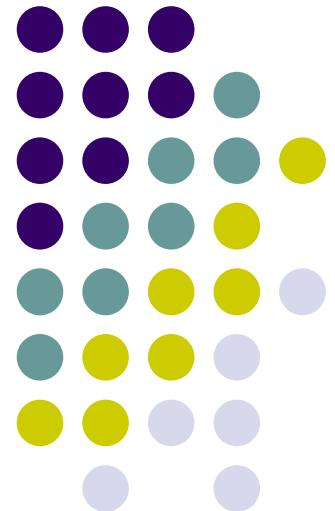


# Incrementally Developing Parallel Applications with AspectJ

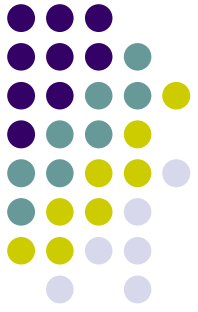
20th International Parallel & Distributed Processing Symposium (IPDPS'06)

João Luís Sobral  
Departamento de Informática  
Centro de Ciências e Tecnologias da Computação  
Universidade do Minho  
Braga - Portugal

Rhodes, 28 April 2006

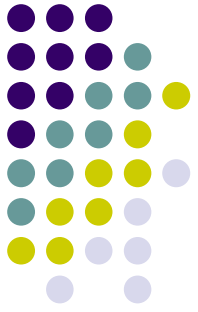


# Developing Parallel Applications with AspectJ



## Presentation Outline

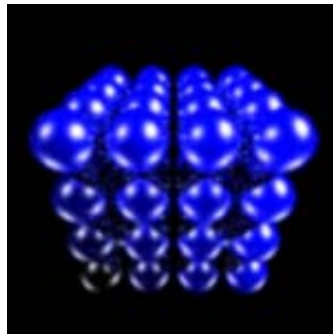
- Motivation for AOP in parallel computing
- AspectJ overview
- Developing modular parallel applications
- Performance results
- Developing reusable parallel modules
- Conclusions and future work



# Motivation for AOP in Parallel Computing

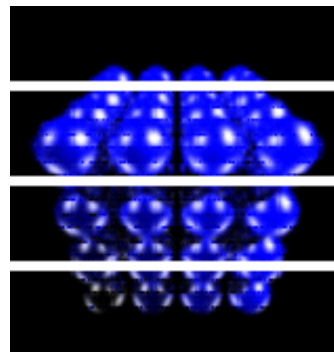
## Parallel application development

- JGF RayTracer implemented with Java concurrency/distribution mechanisms







Sequential

```
RayTracer rt = new RayTracer();  
Image result = rt.render(0,500);
```



Multicore/SMP

```
RayTracer rt[]=new RayTracer[4];  
  
for(i=0; i<4; i++)  
    rt[i] = new RayTracer();  
  
for(int i=0; i<4; i++)  
    new Thread() {  
        void run() {  
            res[i] = rt[i].render(/*sub-interval*/);  
        }  
    }.start();  
  
for(int i=0; i<4; i++) result = ...
```

CPU/core1   
CPU/core2   
CPU/core3   
CPU/core4 

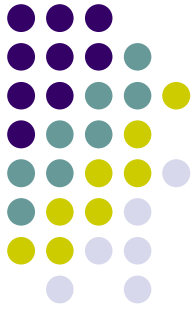


Machine1  
Machine2  
Machine3  
Machine4



MPP/Cluster/Grid

```
public interface IRayTracer extends Remote {  
    public int[] render(...) throws RemoteException;  
}  
IRayTracer rt[]=new IRayTracer[4];  
  
for(i=0; i<4; i++)  
    try { // rt[i] = new RayTracer();  
        rt[i] = (IRayTracer) PortableRemoteObject...  
    } catch(Exception ex) { ... }  
  
for(int i=0; i<4; i++)  
    new Thread() {  
        void run() {  
            try {  
                res[i] = rt[i].render(/*sub-interval*/);  
            } catch(Exception ex) { ... }  
        }  
    }
```



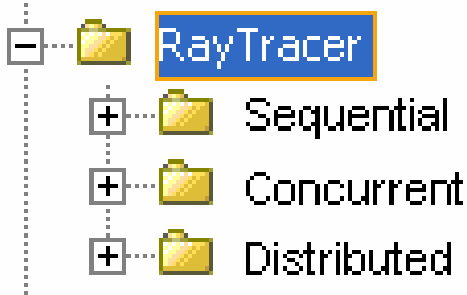
# Motivation for AOP in Parallel Computing

## Parallel application development

- (Non modular) support to parallel application development

**Goal:**

**Modularise RayTracer /  
concurrency / distribution**



```
RayTracer rt = new RayTracer();  
Image result = rt.render(0,500);
```

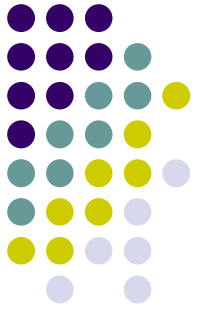


```
RayTracer rt[]=new RayTracer[4];  
  
for(i=0; i<4; i++)  
    rt[i] = new RayTracer();  
for(int i=0; i<4; i++)  
    new Thread() {  
        void run() {  
            res[i] = rt[i].render(/*sub-interval*/);  
        }  
    }.start();  
for(int i=0; i<4; i++) result = ...
```



```
public interface IRayTracer extends Remote {  
    public int[] render(...) throws RemoteException;  
}  
IRayTracer rt[]=new IRayTracer[4];  
for(i=0; i<4; i++)  
    try { // rt[i] = new RayTracer();  
        rt[i] = (IRayTracer) PortableRemoteObject...  
    } catch(Exception ex) { ... }  
for(int i=0; i<4; i++)  
    new Thread() {  
        void run() {  
            try {  
                res[i] = rt[i].render(/*sub-interval*/);  
            } catch(Exception ex) { ... }  
        }  
    }.start();
```

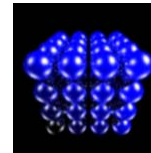
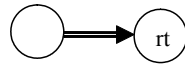




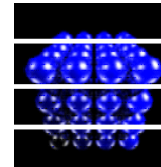
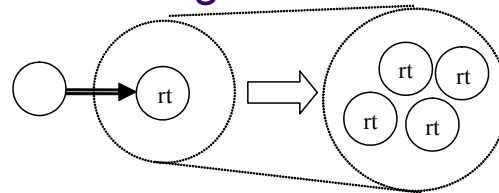
# Modular parallel application development

## An (OO) incremental approach to develop parallel applications

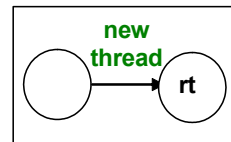
1. Develop core functionality (1 client - 1 server)



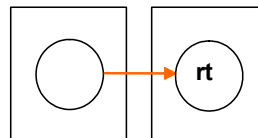
2. Partition work/data among several servers (1 client – several servers)

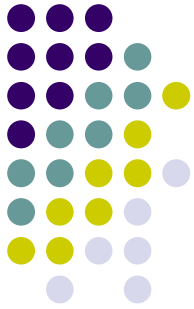


3. Include asynchronous method invocations (1 thread per method invocation)



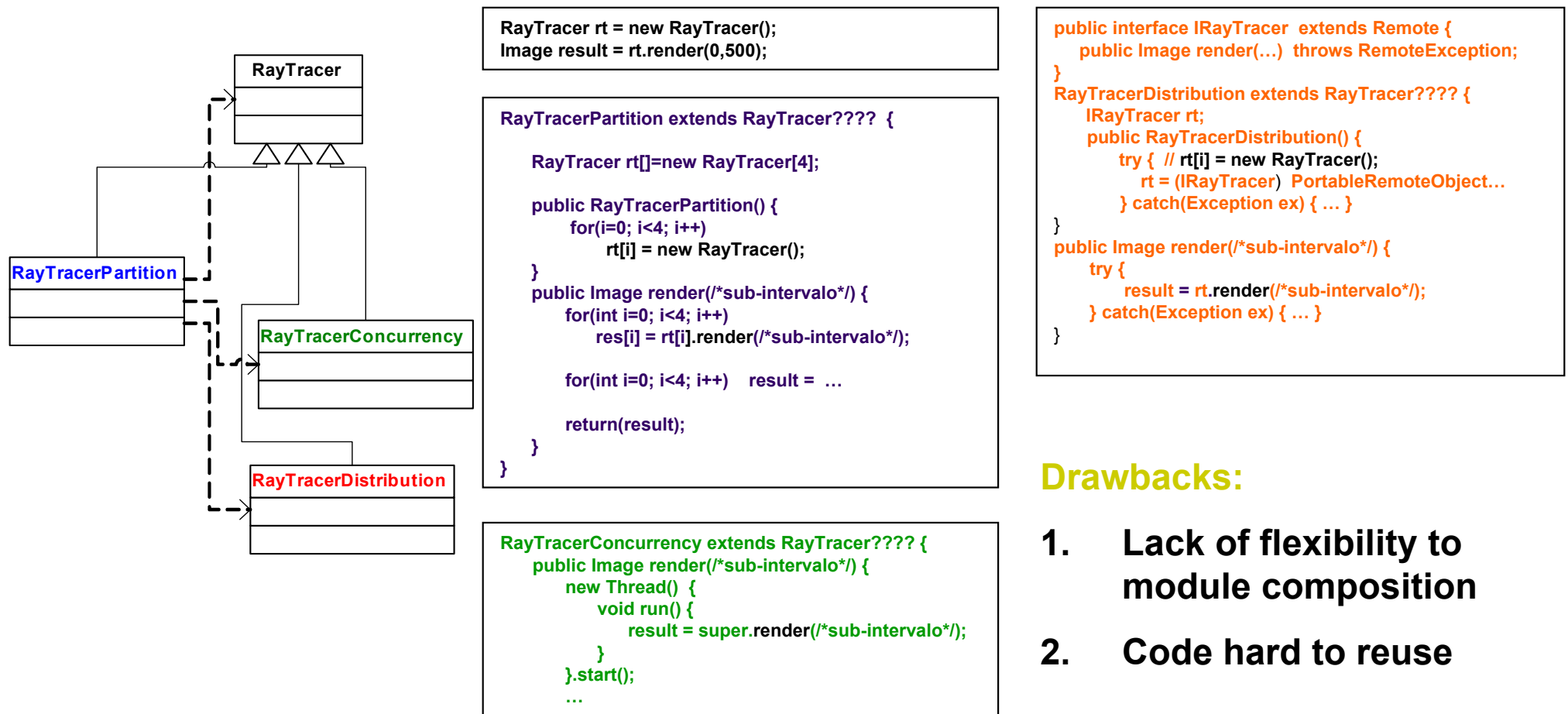
4. Distribute servers among available resources (e.g., RMI middleware)

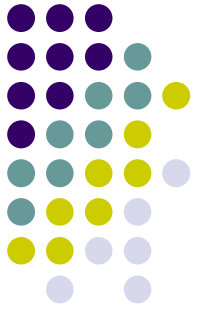




# Modular parallel application development

## Implementing the approach with (OO) inheritance/composition





# AO Support for Modular Parallel Computing

## Benefits from modularising core functionality and parallelisation strategy

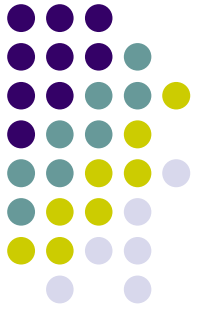
- Core functionality and parallelisation strategy easier to understand
- Core functionality and parallel code can evolve simultaneously
- Ability to plug or unplug parallel code for profiling and debugging

## Benefits from using a fine-grained decomposition

- More incremental application development
- Ability to swap/unplug parallelisation modules
- Increased reuse potential

## Motivation for using AOP

- Parallelisation concerns are typically crosscutting
- Enables new ways to build modular parallelisation concerns, promising to attain higher reuse than traditional OOP

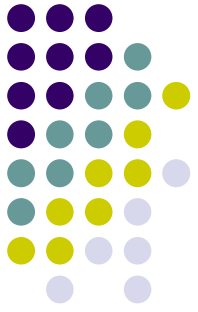


# AO Support for Modular Parallel Computing

## AspectJ Overview

- Extension to Java designed to deal with crosscutting concerns
  - **Static crosscutting**
    - Introduce a method or an instance variable into a class
    - Implement a subclass or an interface
    - ```
public aspect Static {
```
    - ```
    declare parents: Point implements Serializable;
```
    - ```
    public void Point.migrate(String node) { System.out.println("Migrate to node" + node); }
```
    - ```
}
```
  - **Dynamic crosscutting**
    - Intercept an object creation, method call or instance variable access and specify a new behaviour, before, after or instead (around)
    - ```
public aspect Logging {
```
    - ```
    void around(Point obj, int disp) : call(* *.move*(..) && target(obj) && args(disp) {
```
    - ```
        System.out.println("Move called: target object = " + obj + " Displacement " + disp);
```
    - ```
        proceed(obj,disp);
```
    - ```
    }
```
    - ```
}
```





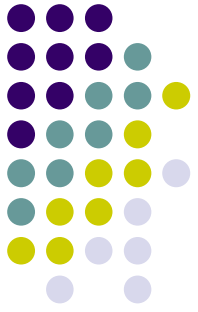
# AO Support for Modular Parallel Computing

## Developing modular parallel applications

- Use one or several modules to implement the parallelisation strategy?
- How to deal with intrinsically parallel applications?
- What are parallelisation concerns?
- How to reuse parallelisation concerns?

## Supporting the approach with AOP

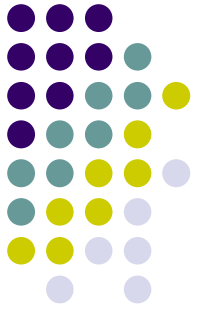
1. Implement core functionality with OOP techniques (i.e., domain specific code)
2. Implement **partition**, **concurrency** and **distribution** concerns with AOP



# AO Support for Modular Parallel Computing

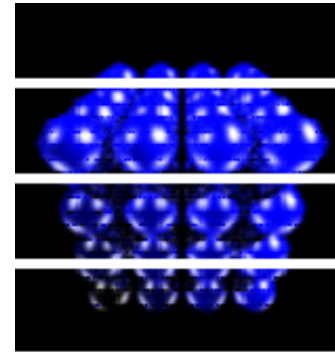
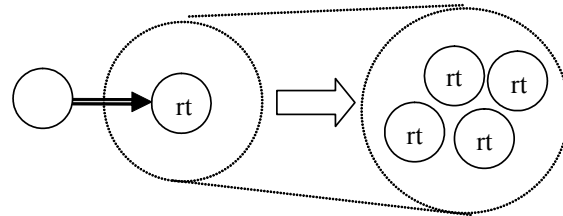
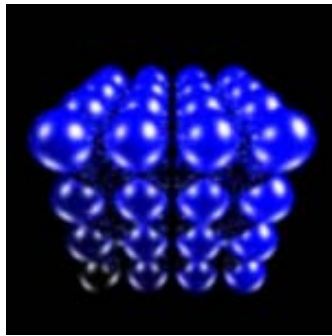
## Supporting the approach with AOP

- **Partition** modules (i.e., aspects) transparently replicate objects, distribute data among these objects and manage method calls to these objects
- **Concurrency** modules implement asynchronous method calls and synchronisation requirements
- **Distribution** modules implement object distribution and remote method invocations
- **Benefits**
  - More flexibility to compose these modules
  - Higher reuse potential



# AO Support for Modular Parallel Computing

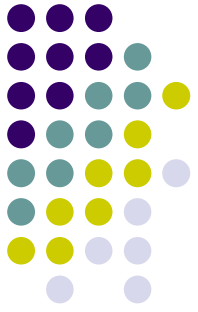
JGF RayTracer which aspects for **partition**, **concurrency** and **distribution**



```

RayTracer rt = new RayTracer();
RayTracer farm[] = new RayTracer[4];
RayTracer around() : call (RayTracer.new()) {
    for(i=0; i<4; i++)
        farm[i] = new RayTracer();
    return(farm[0]);
}
Image result = rt.render(0,500);
Image around(/* ... */) : call (*.render(..)) {
    for(int i=0; i<4; i++)
        res[i] = farm[i].render(/* subinterval* */);
    ... //join sub-images saved in res array
    return(/*merged subimages* */);
}
RayTracer around() : call (RayTracer.new()) {
    ... // request object creation to remote factory
    ... // associate remote object to local fake
    return(/*fake local object* */);
}
Image around() : call (RayTracer.render()) {
    (new Thread() {
        public void run() {
            ... // redirect to remote node
            around() : call (*.render(...)) {
                ... // redirect to remote node
                return(/*rendered image* */);
            }
        }
    }).start();
}
    
```

new thread

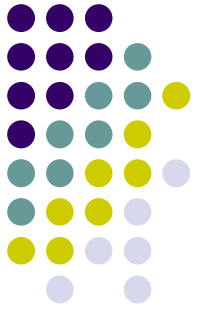


# AO Support for Modular Parallel Computing

## Supporting the approach with AOP

- Examples of deployable parallel applications

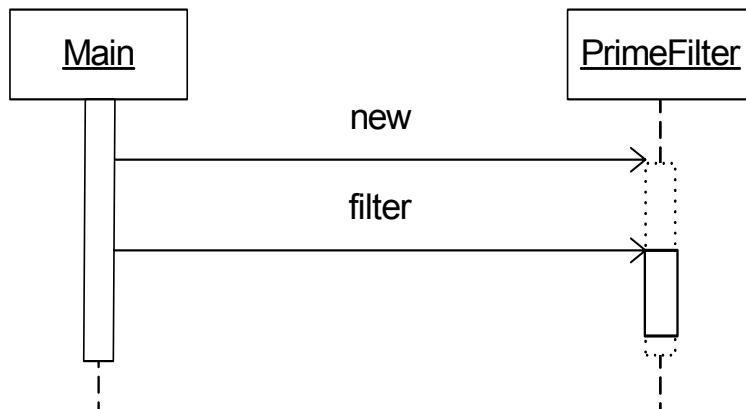
Partition module	Concurrency module	Distribution module	Purpose
No	No	No	Tidy up core functionality, debugging, single processor machines
Yes	No	No	Tidy up partition strategy, debugging
No / Yes	Yes	No	Shared memory parallel machines (SMP/Multi-core)
Yes	Yes	Yes	Distributed memory machines/Grids
No	No / Yes	Yes	Distributed application



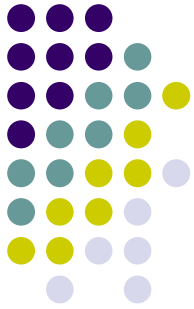
# Developing Parallel Applications with AspectJ

## Example: Prime Filter – to find all prime numbers up to a pre-defined limit

- Core Functionality



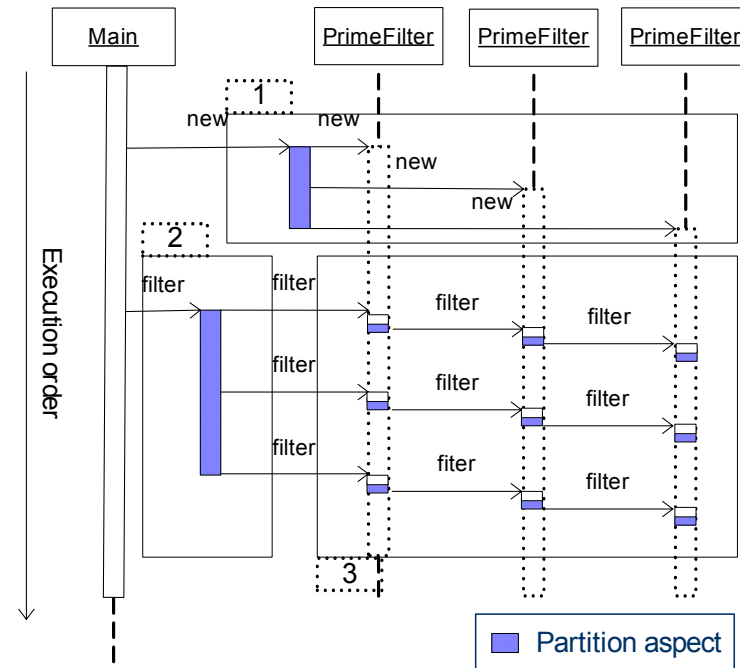
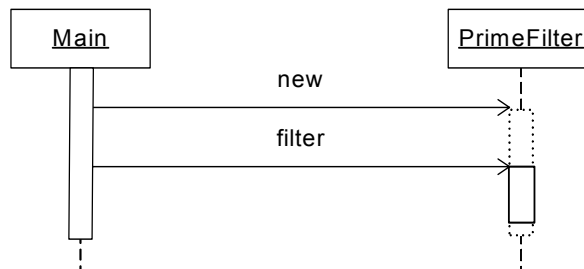
```
public class PrimeFilter {  
  
    // calculates primes between [pmin,pmax]  
    public PrimeFilter(int pmin, int pmax);  
  
    // remove non-primes from num list  
    public void filter(int num[]);  
}  
  
void public static void main(String[] arg) {  
  
    int list[] = ... // place numbers in the list  
  
    PrimeFilter p = new PrimeFilter(2, sqrt(Max) );  
  
    p.filter(list);           // filters the list  
}
```

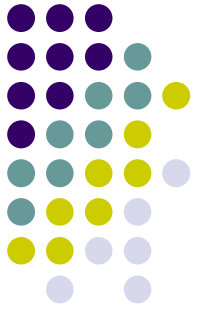


# Developing Parallel Applications with AspectJ

## Example: Prime Filter (cont. 1)

- Partition Aspect (pipeline)
  1. Create several prime filters
  2. Split numbers to filter into tasks
  3. Forward calls among elements

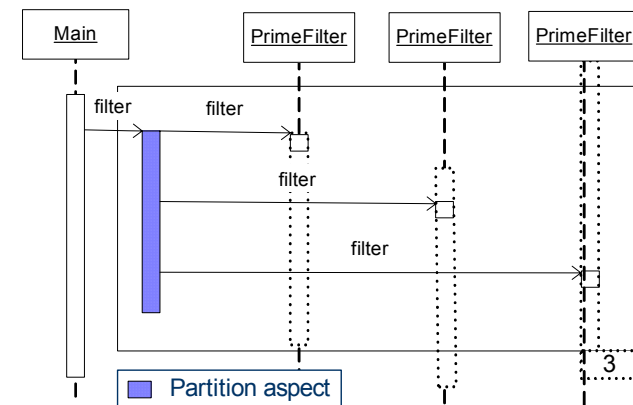
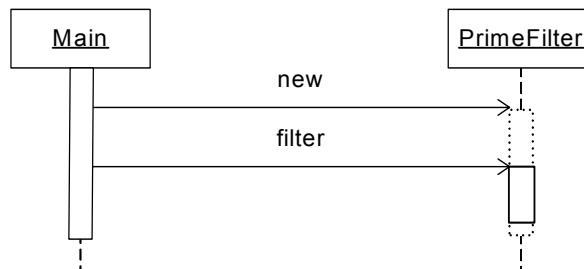


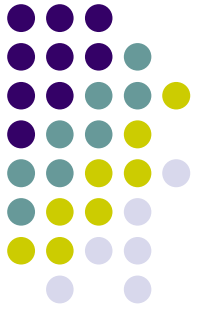


# Developing Parallel Applications with AspectJ

## Example: Prime Filter (cont. 2)

- Partition Aspect (Farm)
- 1. Create several prime filters
- 2. Split numbers to filter into tasks
- 3. Forward each call to one element

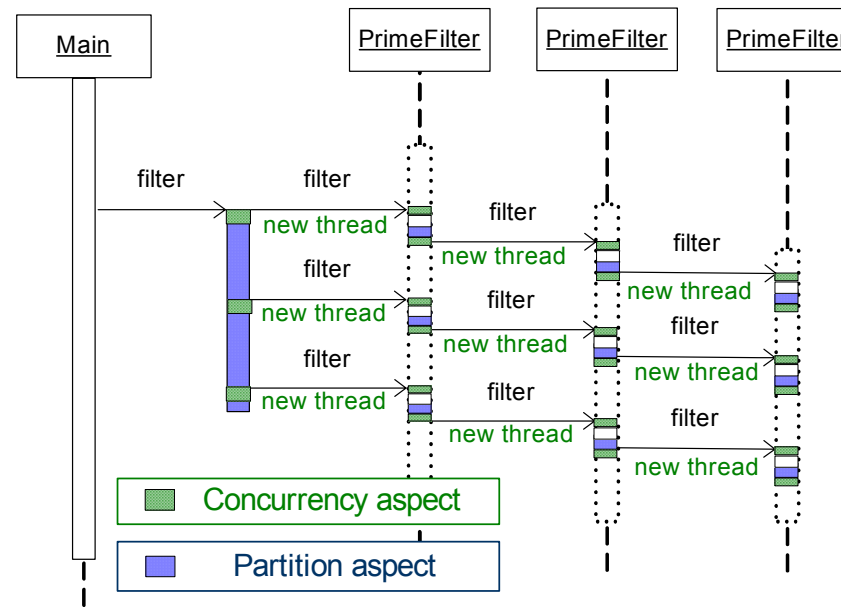




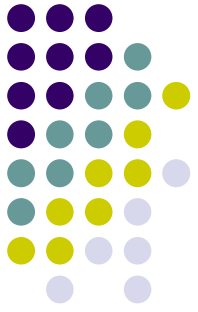
# Developing Parallel Applications with AspectJ

## Example: Prime Filter (cont. 3)

- **Concurrency Aspect**
  1. **Spawn a new thread for each call**
  2. **Synchronise access to prime filters**



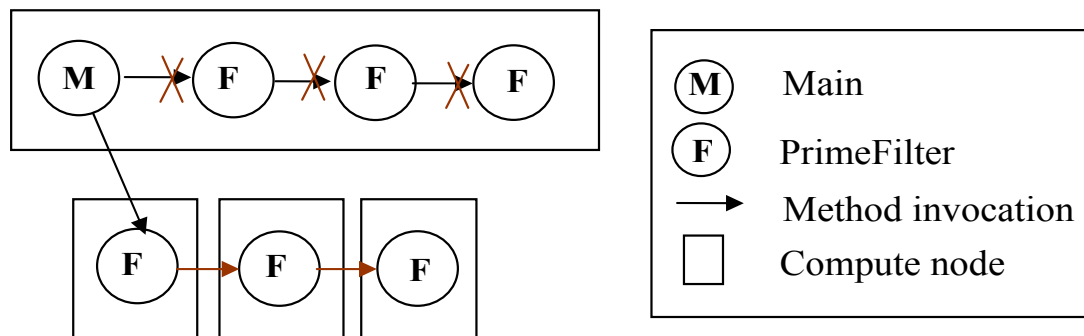


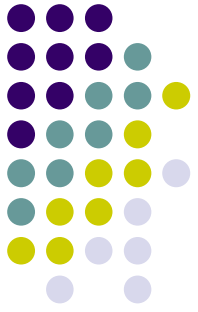


# Developing Parallel Applications with AspectJ

## Example: Prime Filter (cont. 3)

- **Distribution Aspect**
  1. Remotely instantiate prime filters
  2. Redirect method call to remote nodes

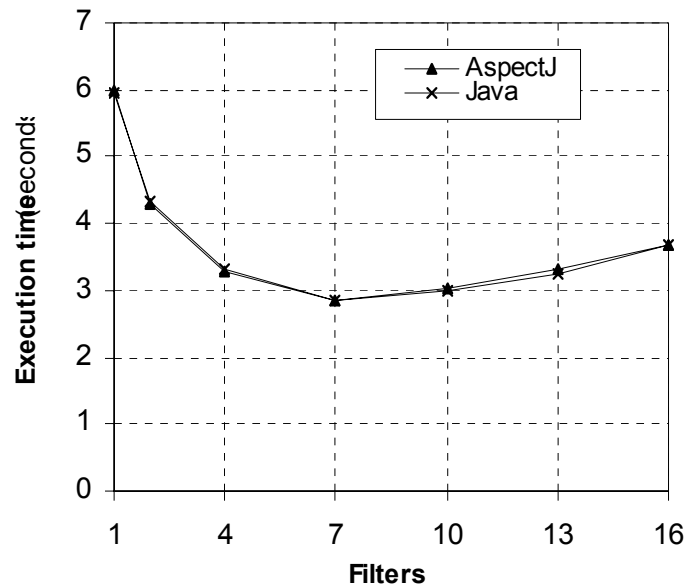


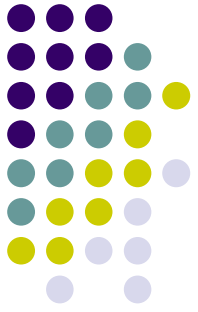


# Developing Parallel Applications with AspectJ

## Performance Results (Prime Sieve)

- Less than 5% overhead in this application



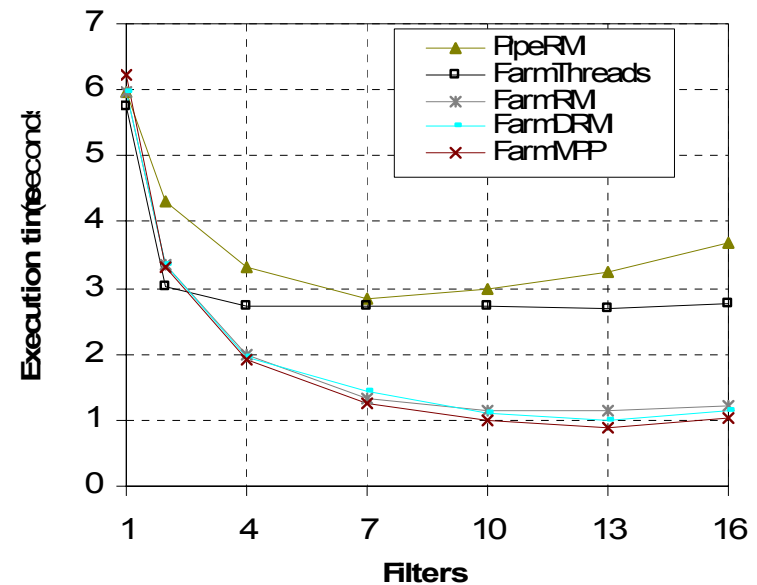


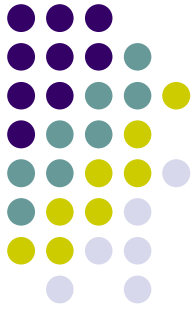
# Developing Parallel Applications with AspectJ

## Performance Results (Prime Sieve – cont.)

- Tested combinations of modules

	Partition	Concurrency	Distribution
<b>FarmThreads</b>	Farm	Yes	No
<b>PipeRMI</b>	Pipeline	Yes	RMI
<b>FarmRMI</b>	Farm	Yes	RMI
<b>FarmDRMI</b>	Dynamic Farm		RMI
<b>FarmMPP</b>	Farm	Yes	MPP

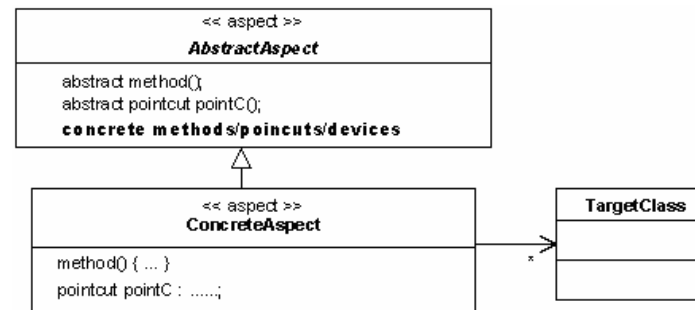




# Developing reusable modules

## How to implement reusable modules?

- Marker interfaces
- Abstract pointcuts



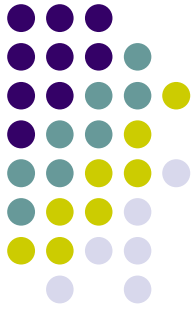
## Concurrency modules

- Common concurrency mechanism and patterns (oneway, futures, barrier, synchronisation, read/write lock, etc)

- Example: 

```
public abstract aspect OnewayProtocol {
    pointcut onewayMethodExecution();
```

```
void around(): onewayMethodExecution() {
    Thread t = new Thread() {
        public void run(){
            proceed();
        }
    };
    t.start();
}
```



# Developing reusable modules

## Partition modules

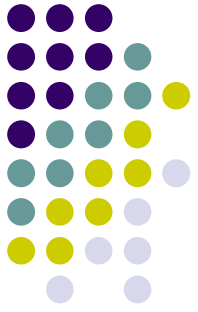
- Marker interfaces specify objects to replicate
- Abstract pointcuts to specify method calls to broadcast, scatter, reduce among replicated objects (a.k. GMI and Concurrent Aggregates)
- Abstract methods to provide call-specific parameters

```
public abstract aspect GridProtocol {
    public interface Grid {}
    Vector elem = new Vector();
    Object around() : call(Grid+.new()) {
        for(int i=0; i<workers; i++) {
            elem.add(ceed());
        }
        return(elem.get(0));
    }
}
```

## JGF RayTracer with reusable modules

<pre>RayTracer rt = new RayTracer(); Image result = rt.render(0,500);</pre>	<pre>aspect Partition extends GridProtocol {     declare parents: RayTracer implements Grid;     // calculates parameters of each scatterCall     Vector scatter(Object arg) {         Vector v = new Vector();         ... // splits args into sub-intervals         return(v);     }     pointcut scatterCall(..) : call (RayTracer.render(..)); }</pre>	<pre>public aspect Future extends FutureProtocol {     protected pointcut futureMethodExecution() :         (execution(* RayTracer.render(..))); }</pre>
---	--	--

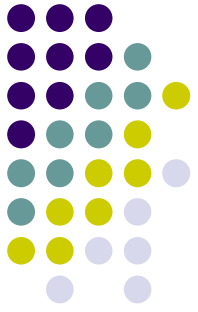
# AO Support for Modular Parallel Computing



## Conclusions

- AOP offers an alternative to modularise parallel applications
- Successfully modularised several types of parallel algorithms
  - **Heartbeat, farming, pipeline and divide and conquer**
- Parallel applications can be developed by composing parallelisation modules
- Easy to swap the parallelisation (e.g, farm, pipeline) and the middleware (e.g., MPI, RMI, Grid)
- Higher reuse potential, but more design effort required
- Performance is close to OOP (when weaving can achieve *similar* parallel code)
- Composing reusable modules requires more fine-grained control over the scope of an aspect

# AO Support for Modular Parallel Computing



## Current and Future Work

- Apply the approach “in the large”
- Develop reusable modules for distribution and optimisation on distributed systems
- Develop a model of composition of reusable aspects