Parallelisation Service in the AspectGrid Framework

Jorge Pinho, **Matheus Almeida**, Miguel Rocha, João L. Sobral

Ibergrid 2010

Braga, 27 May 2010

Outline

- Motivation
- What is Gridification?
- AspectGrid Framework Overview
- Parallelisation Service
- Case Study: JECoLi Gridification
- Conclusion and Future Work

Motivation

- *Gridify* existing scientific codes
  - *Gridification*: process of enabling an application to (efficiently) run on Grid environments
- Limitations of current approaches:
  - Impose additional burden
    - Require invasive modifications to the base code
    - Applications become dependent on the Grid
  - Trade-off: non-invasive vs fine grained
  - Discourage application-specific enhancements when running in Grid environments

Common Gridification Types

- Compose existing services into a new one
- Deploy an application as a Grid service
- Transparent access to remote resources

- Enable an application to run on (multiple) remote resources to improve performance
  - Fine or coarse grain

Gridification Taxonomy

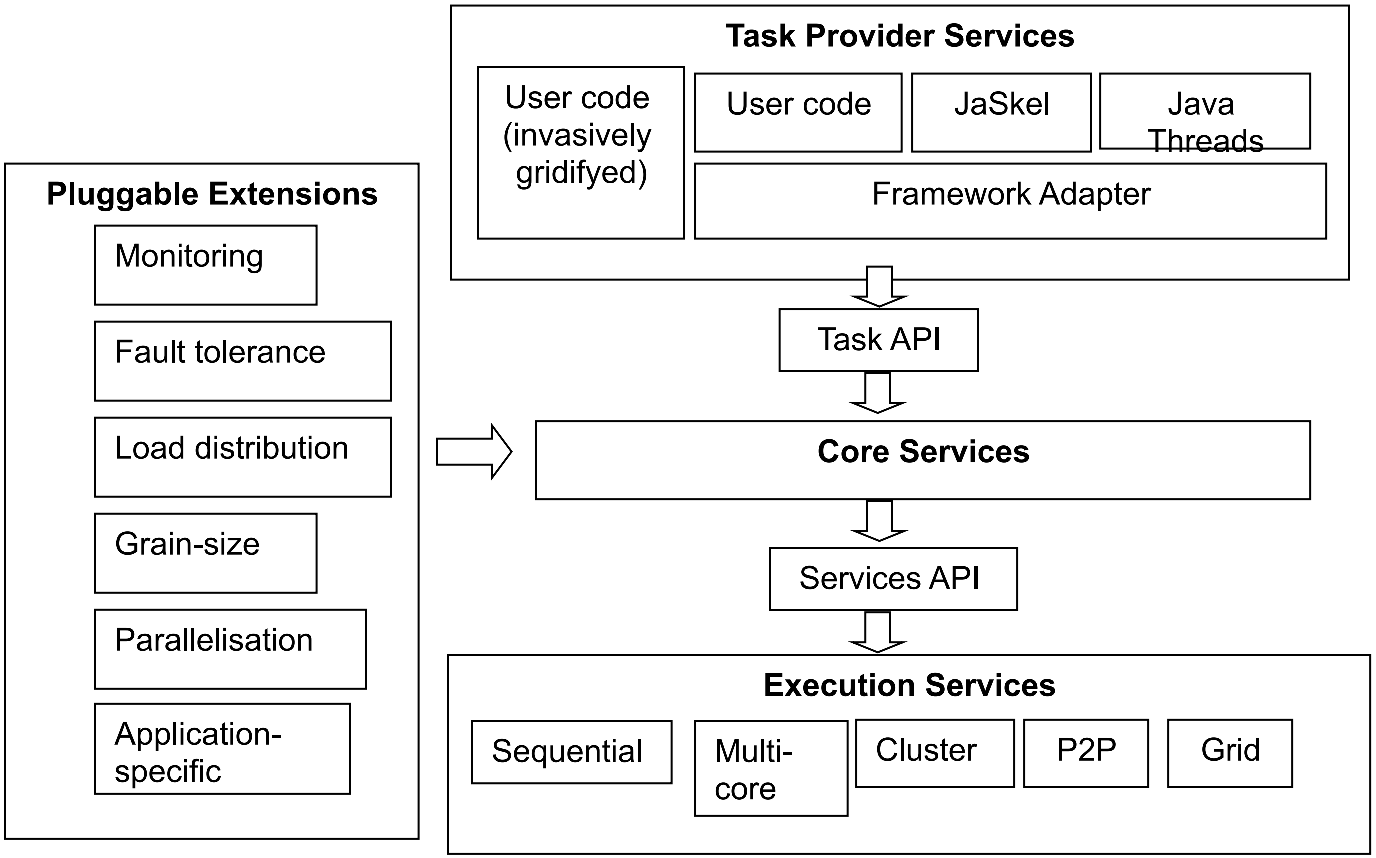
	Coarse-Grained	**Fine-Grained**
Invasive	ProActive, PAGIS, …	Ibis (Satin), …
Non-Invasive	GEMLA, GRASG, …	***AspectGrid***

AspectGrid Framework Overview

- *Non-Invasive* and *Fine-grained* Gridification
  - Three stage gridification
- Gridification issues addressed
  - Remote execution, monitoring, data access
  - Fault-tolerance and self-adaptability
- Explores new programming tools to deliver:
  - (un)pluggable and composable services
    - each service is implemented as a separate *concern*
  - Application specific enhancements

AspectGrid Framework Overview

- User code (domain specific) is rewritten to obtain Grid-enabled version(s)



Parallelisation Service

- Invasive Parallelisation
  - parallel red-black version of the Successive Over-Relation method

```
public class SOR {
  public static int nprocess;
  public static int rank;
  protected static double[][] G;
  protected static double[][] p_G;
  public static void SORrun(double omega, int num_iterations) {
    int M= p_G.length;
    int N= p_G[0].length;
    double omega_over_four=omega*0.25;
    double one_minus_omega=1.0-omega;
    ... /* send/receive matrix to/from process 0 */
    MPI.COMM_WORLD.Barrier();
    for(int p=0; p<2*num_iterations; p++) {
      do_iteration(p%2, M-1, N-1, omega_over_four, one_minus_omega);
      /* exchange matrix borders */
      if(rank!=nprocess-1){
        MPI.COMM_WORLD.Sendrecv(p_G[p_G.length-2],0,/*...*/);
      }
      if(JGFSORBench.rank!=0){
        MPI.COMM_WORLD.Sendrecv(p_G[1],0,/*...*/);
      }
    }
    ... /* send/receive matrix to/from process 0 */
  }
  static void do_iteration(int odd, int Mm1, int Nm1, double oof, double omo) {
    /* ...*/
  }
}
```

Parallelisation Service

- Modular Parallelisation with OO inheritance
  - Include parallelisation code by extending base classes

```
public class SOR {
  /* ... */
  public static void SORrun(double omega, int num_iterations)
   /* ...*/
 }
 static void do_iteration(int odd, int Mm1, int Nm1, double oof, dou
   /* ...*/
 }
}
```

```
public class MPISOR extends SOR {

 static int nprocess, rank;
 static double[][] p_G;

 public static void SORrun(double omega, int num_iterations) {
   G = p_G;  // swap matrix

   ... /* send/receive matrix -/from process 0 */
   MPI.COMM_WORLD.Barrier();
   SOR.SORrun(omega, num_iterations);
   * send/receive matrix to/- process 0 */

 void do_iteration(int a, int Mm1, int Nm1, double oof, double omo) {

 R.do_iteration(a, Mm1, Nm1, oof, omo);
```

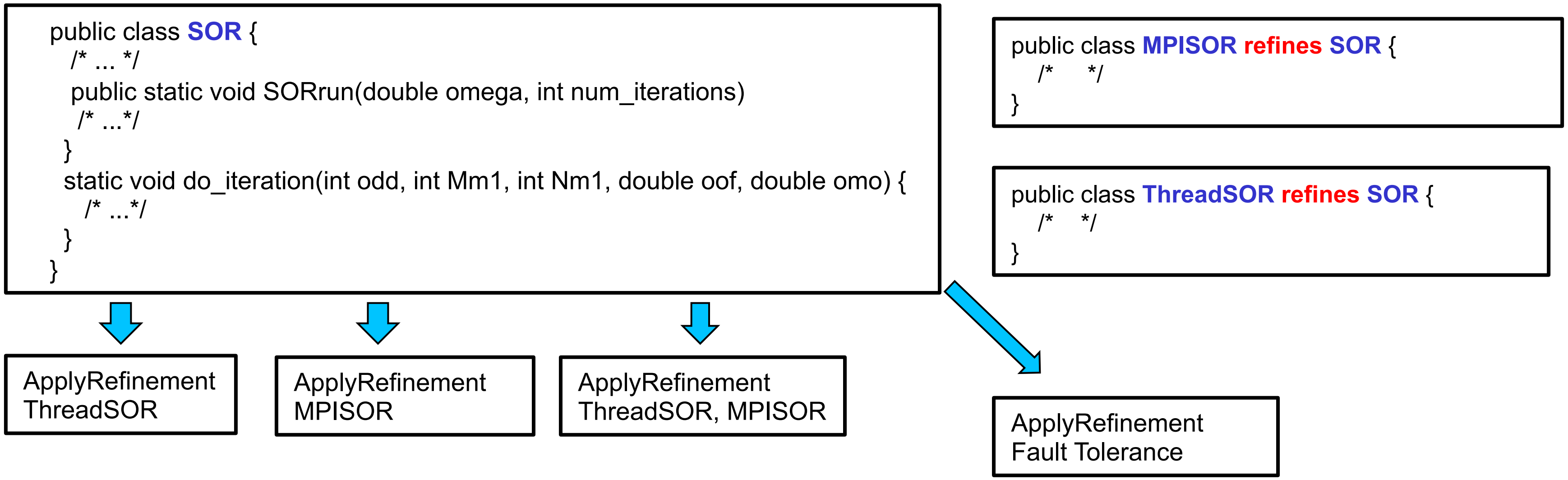
```
public class ThreadSOR extends SOR {

 static void do_iteration(int a, int Mm1, int Nm1, double oof, double omo) {

   Thread t = new Thread() {
     void run() {

  }
}
```

But: it requires invasive changes to the base code

- Derived classes must be explicitly referenced in the base code (e.g., ThreadSOR or MPISOR instead SOR)
- Does not scale if multiple services and /or alternative mappings should be "plugged"

Parallelisation Service

- Non-invasive parallelisation with class refinement
  - Derived classes **rewrite** the base code



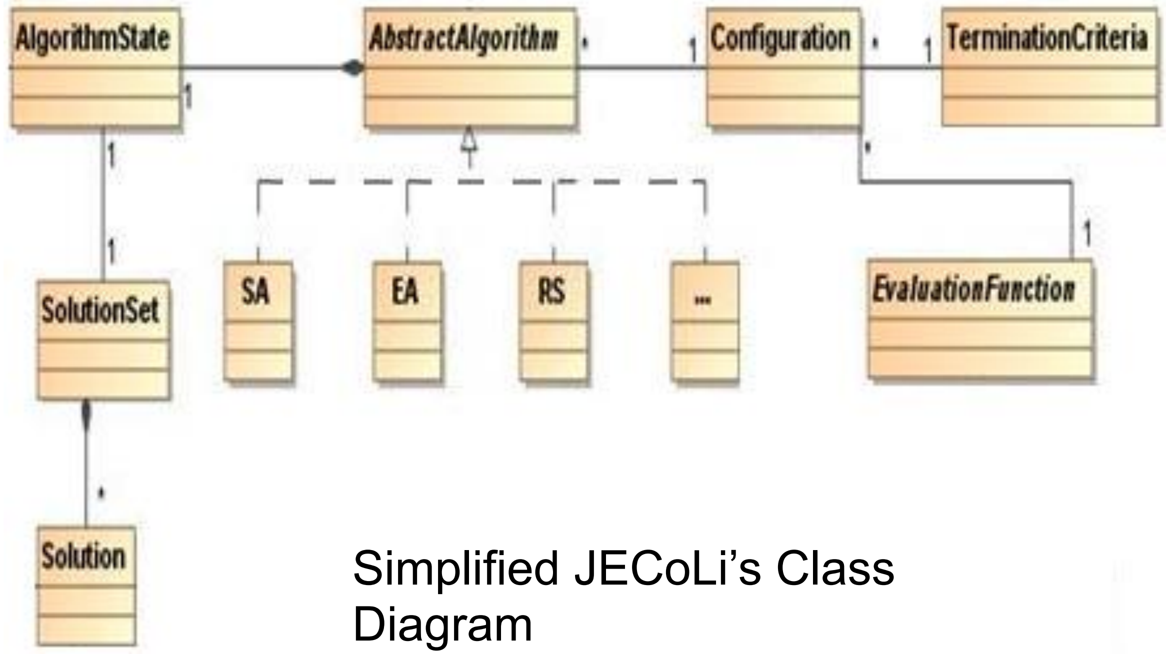
- Multiple code versions can coexist
  - Deploy modules (services) for each target platform / grid functionality
  - Composition of modules (services)
- Pre-built refinements implement common gridification concerns

Case study: JECoLi Parallelisation

- Java Evolutionary Computation Library (JECoLi)
  - Focus on implementing optimization approaches from the Genetic and Evolutionary Computation Field
- Main characteristics
  - Implements a large set of meta-heuristics algorithms
    - EAs, differential evolution, genetic programming, simulated annealing, multi-objective
  - Extensive set-up of each algorithm
    - Solution’s encoding, reproduction, selection, termination criteria
  - Modular & Extensible software platform
  - 100% Java & Open Source
  - Approx. 462 classes and 55k lines of code

Case study: JECoLi Parallelisation

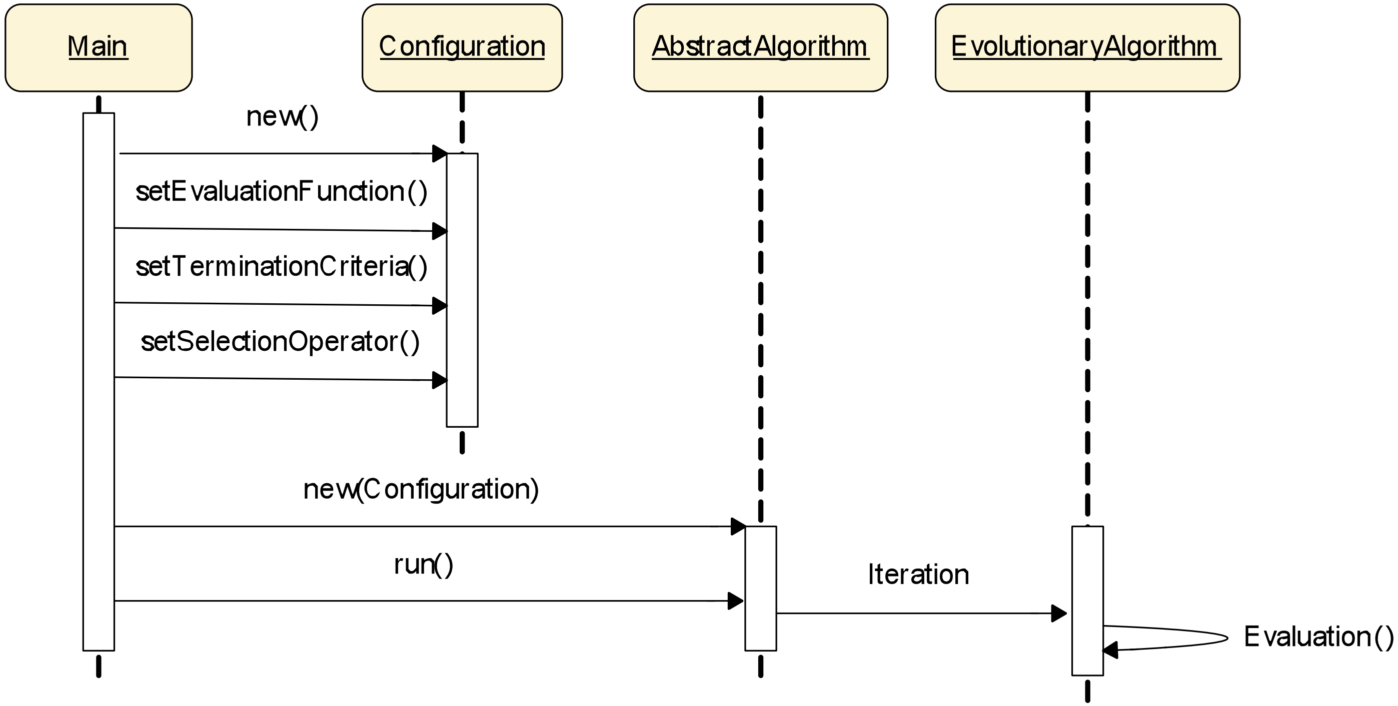
- Main abstractions
  - Solutions
    - Encoding
  - Evaluation Function
  - Termination Criteria
  - Algorithm
    - Optimization method
  - Configuration
    - Algorithm configuration



Simplified JECoLi's Class Diagram

Case study: JECoLi Parallelisation

- Typical workflow in the **JECoLi**

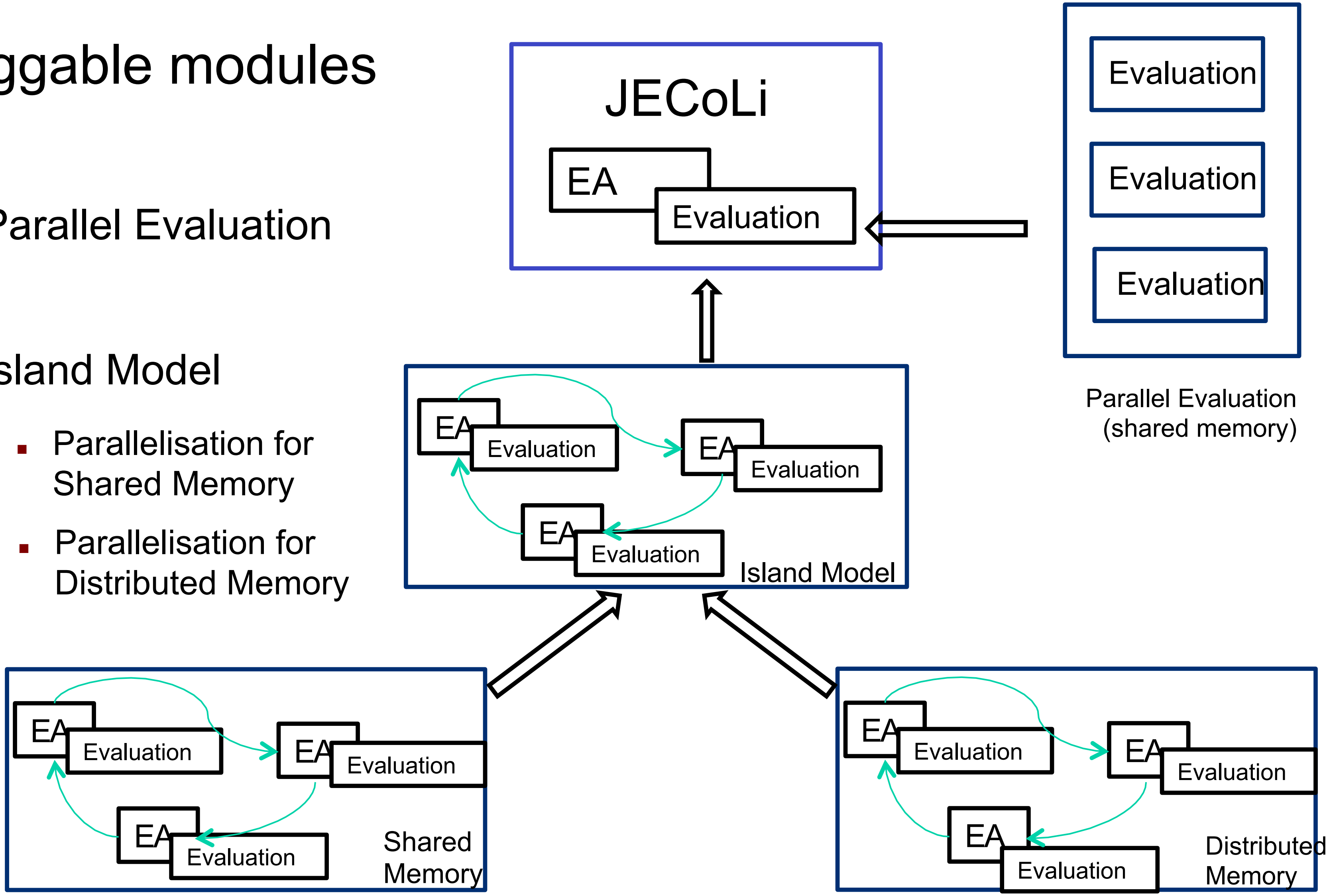


Case study: JECoLi Parallelisation

- Original version was sequential
  - Could not benefit from multi-core & cluster & Grid
- Well known techniques to develop parallel EA
  - Parallel evaluation
  - Island model (with migration)
  - Hybrid (multi-island, island with parallel evaluation, …)
- Our goal:
  - Non-invasive and pluggable parallelisation
  - Support multiple parallelisation models / target platforms
  - Attain complex parallelisation by composing models

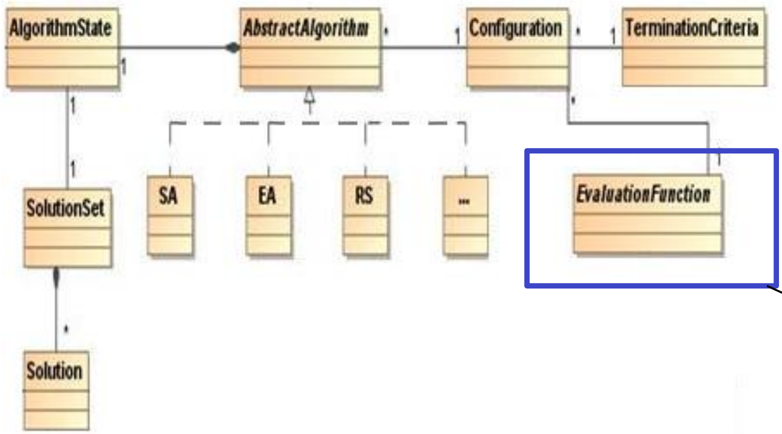
Case study: JECoLi Parallelisation

- Pluggable modules
  - Parallel Evaluation
  - Island Model
    - Parallelisation for Shared Memory
    - Parallelisation for Distributed Memory



Case study: JECoLi Parallelisation Refinement for Parallel Evaluation

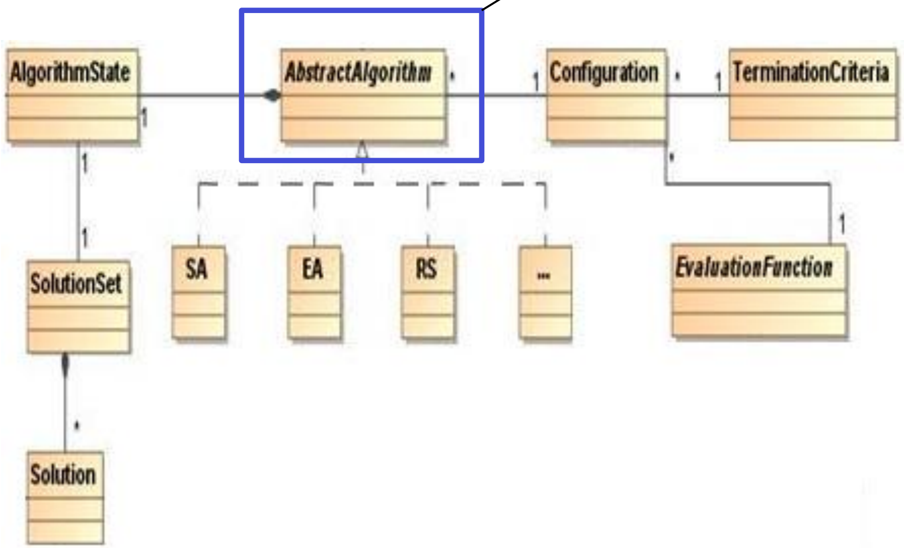
- **Logically divides the solution set into subsets**
- **Spawns a thread to *evaluate* each solution subset**




```
public class ParallelEvaluation refines EvaluationFunction {

  public void evaluate(ISolutionSet solutionSet) {
     ... // split solutionSet into subsets
     for (int i = 0; i < numberOfSolutionSubsets; i++) {
       Thread t = new Thread() {
          void run() {
             original(...);    // invoke original method on each subset
          }
       }
       t.start();
       ... /* ... */
  }
}
```

Case study: JECoLi Parallelisation Refinement for Island Model

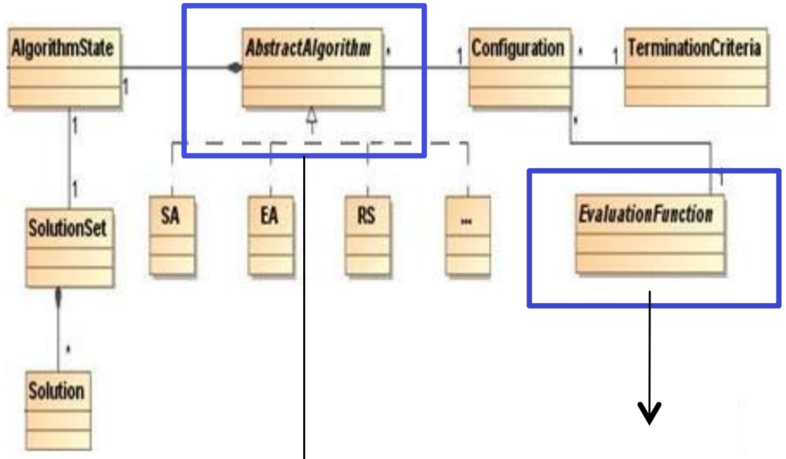
- Acts upon calls to method *run*
  - Creates multiple EAs
  - Builds interconnection topology among islands
  - Calls *run* on each island




```
public class IslandModel refines AbstractAlgorithm {

  public new AbstractAlgorithm(IConfiguration conf) {
    ... // clone configuration and build island topologies
     original(conf);
  }

  public IAlgorithmResult run() {
     ... // calls the run method on each IConfiguration clone
     ... // joint multiple IAlgorithm instances into a single value
  }

  protected ISolutionSet iteration(ISolutionSet solutionSet) {
     do_migrations();
     orginal(solutionSet);
  }

  void do_migrations() { ... }   // migrate solutions among islands
}
```

Case study: JECoLi Parallelisation
JECoLi deployable versions

1. Base Framework
2. Parallel Evaluation
3. Island Model
4. Hybrid Model



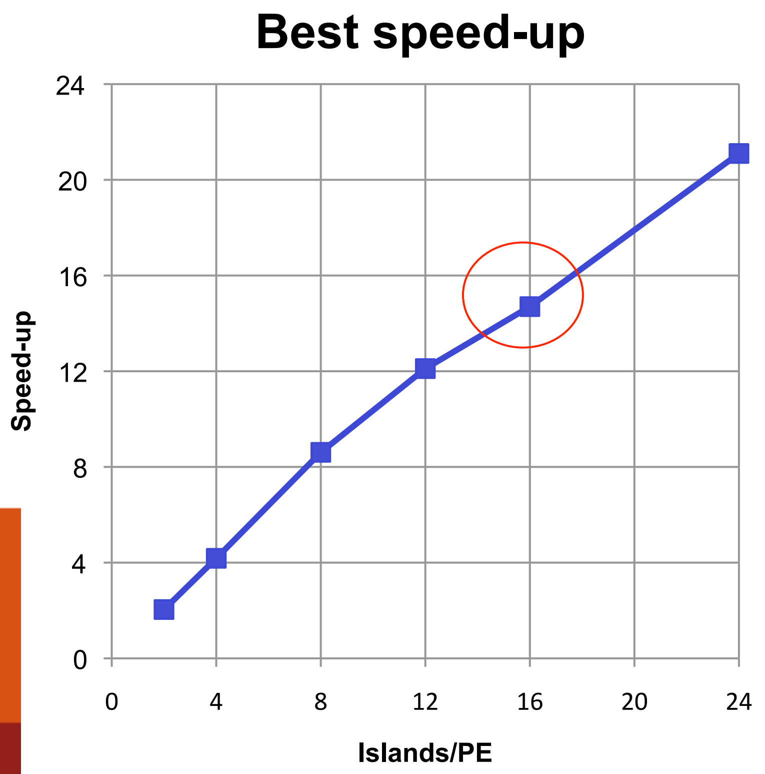

```
public class ParallelEvaluation refines EvaluationFunction {

t solutionSet) {
ets
tionSubsets; i++) {

iginal method on each subset
```

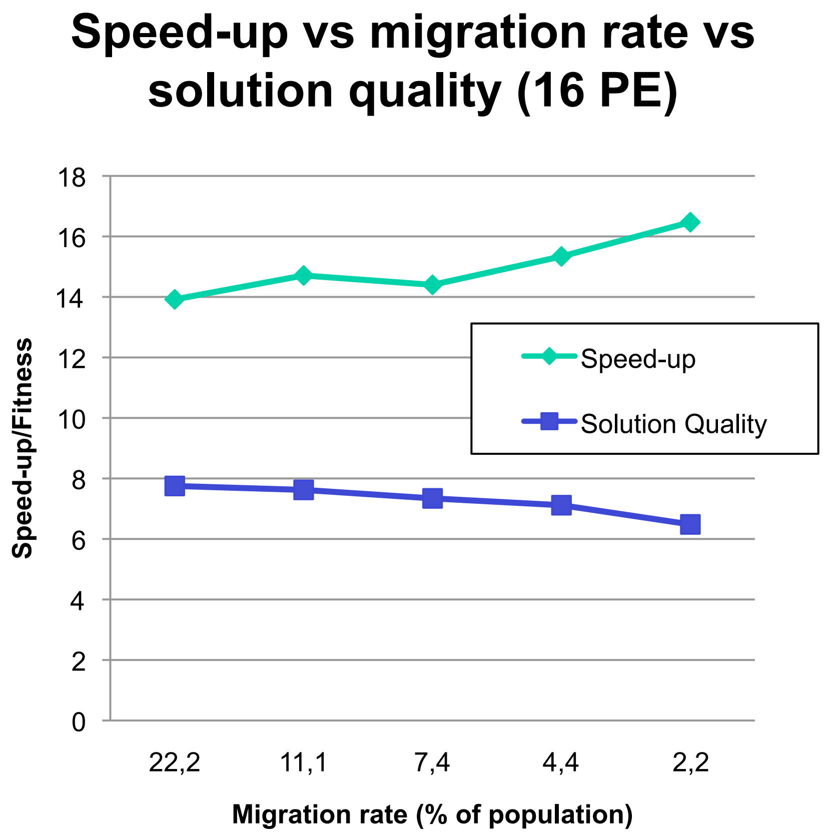
```
public class IslandModel refines AbstractAlgorithm {

  public new AbstractAlgorithm(IConfiguration conf) {
   ... // clone configuration and build island topologies
    original(conf);
  }

  public IAlgorithmResult run() {
    ... // calls the run method on each IConfiguration clone
    ... // joint multiple IAlgorithm instances into a single value
  }

  protected ISolutionSet iteration(ISolutionSet solutionSet) {
    do_migrations();
    orginal(solutionSet);
  }

  void do_migrations() { ... }   // migrate solutions among islands
}
```

Case study: JECoLi Parallelisation

- Benchmark
  - Optimization of fed-batch **fermentations** to achieve optimal feed-forward control
    - find the optimal input feeding trajectories, in order to improve the process performance
  - **Cluster of 4 machines**
    - Dual - Xeon E5520 (i7-Quad@2.26 GHz w/HT), 8 GB RAM
    - Myrinet 10Gb/s
    - mpiJava over OpenMPI

Benchmarks – Fermentation Cluster – Parallel Island





- Speed-up is linear up to 12 PE
  - Dropoff above due to migration

- Migration rate imposes a trade-off:
  - Decreasing migration rate increases speed-up but also decreases the solution quality

Conclusion

- AspectGrid framework
  - Fine-grained, non-invasive gridification
  - Based on Pluggable Services
  - Parallelisation service by Class Refinement
- Successful non-invasive gridification of the JECoLi framework
  - Base code is oblivious of grid execution concerns
  - Coexistence of multiple versions of the application
    - Best mapping can be used for each target platform
      - Load-time decision

Current and Future work

- Experiments running on a Grid environment comprising a 2 gLite-based sites (UM & UC)
  - Parallel execution spawning across both sites
    - Optimisation methods that require lower migration rates
- Another case study: Gridification of Molecular dynamics simulations
- Monitoring and fault-tolerance services