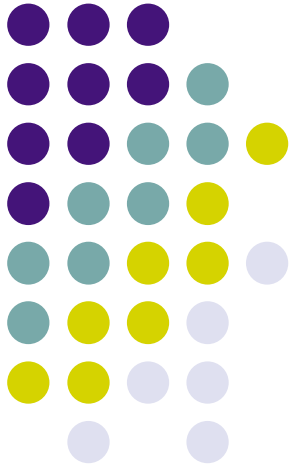
A Process for Separation of Crosscutting Grid Concerns

Paulo Henrique M. Maia[1], Nabor C. Mendonça[1],
Vasco Furtado[1], Walfredo Cirne[2], Katia Saikoski[3]

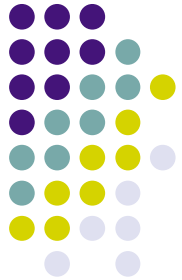
[1]University of Fortaleza (UNIFOR)
Fortaleza – CE – Brazil

[2]Federal University of Campina Grande (UFCG)
Campina Grande – PB – Brazil

[3]HP Brasil
Porto Alegre – RS – Brazil

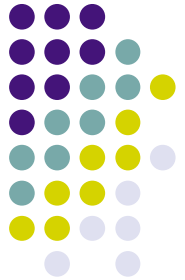


Talk Overview



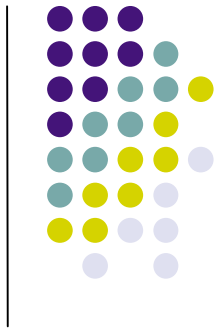
- Motivation
- Research goals
- The GridAspecting process
- An exampple
- Conclusions

Motivation



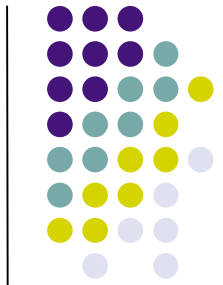
- Increasing demand for commercial and scientific applications that need to process a vast amount of information
- Many of those applications can have significant performance gains by means of parallelization
- However, high-performance parallel machines are too expensive for general use
- A more cost-effective solution is the on-demand integration of multiple independent networked computers as a platform (“grid”) for the execution of parallel applications

Motivation (II)



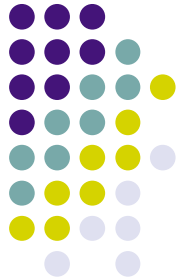
- Many grid systems available: Globus, OurGrid, …
- From a development perspective, grid-based parallel applications must be implemented in a way to avoid a direct dependency on a specific grid platform
  - Easies maintenance, evolution and reuse
- However, explicitly separating grid-specific concerns when implementing a grid application is non-trivial
  - Grid services typically used as the abstractions for application decomposition
  - Tangling of functional and grid-specific concerns in the application code

Motivation (III)



- The *aspect-oriented programming* (AOP) paradigm provides new types of abstractions and composition mechanisms that allow the modular implementation of crosscutting design concerns that otherwise would be tangled and scattered throughout the application code
  - AOP support now available for many languages, most notably Java (AspectJ, AspectWerks, JBoss-AOP, JAsCo, …)
- AOP could be a handy solution to help separate grid concerns in grid-based parallel applications!

Research Goals



- To investigate and define a process for separation of grid-specific concerns in the development of grid-based Java applications using AOP
- To demonstrate the feasibility of the process by applying it to separate grid concerns in existing (“real-world”) parallel applications

The GridAspecting Process

- Main characteristics:
  - Uses a restricted subset of the Java threads model as the mechanism for functional decomposition of the parallel application into separate (concurrent) tasks
  - Uses AspectJ to dynamically intercept the application's thread creation and thread initialization method calls, and to replace them with calls to the corresponding task creation and task intialization sevices provided by the underlying grid API
- Current version targeted for Bag-of-Tasks (BoT) parallel applications developed using the OurGrid open source grid middleware

The GridAspecting Process (II)

- Benefits of the process:
  - Simplifies the grid programming model
  - Improves the parallel application's modularity
  - Allows to functionally test the parallel application with or without the grid
  - Avoids the relatively long submission-execution-evaluation cycle that is typical to the development of grid applications

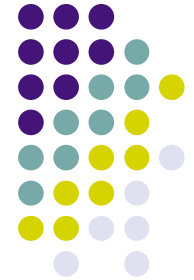
The GridAspecting Process (III)

- Process steps:
  - Step 1 – Task identification and separation of grid concerns
  - Step 2 – Task implementation
  - Step 3 – Task execution and coordination
  - Step 4 – Task parallelization using aspects
- Provide implementation guidelines to:
  - Develop a new grid application
  - Improve the modularization of an existing grid application
  - Parallelize an existing sequential or concurrent application

The GridAspecting Process (IV)

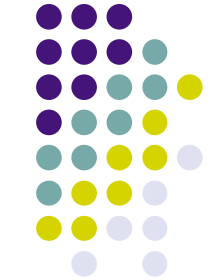
- Step 1 – Task identification and separation of grid concerns
  - Implement or identify parallelizable functional components (task candidates)
  - Separate them from the rest of the code
  - Can benefit from existing concern extraction techniques and tools

The GridAspecting Process (V)



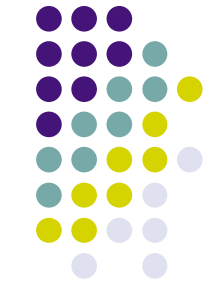
- Step 2 – Task implementation
  - (Re)Implement the tasks candidates identified in the previous step using threads
  - Each task should be able to execute independently and concurrently with the other tasks, without making any direct reference to the grid API
  - All communication from the main application to each of its concurrent tasks should be restricted to parameters passed to the tasks constructors at task creation time

The GridAspecting Process (VI)



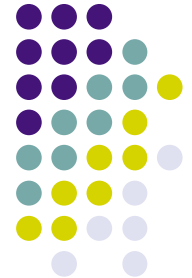
- Step 3 – Task execution and coordination
  - Application creates and executes its concurrent tasks by creating the corresponding task objects and then calling their `start` method
  - Application waits until all tasks have finished their execution by calling their `join` method, and then proceeds to collect their results
  - After this step, the parallel application is functionally operational, and can be tested concurrently in a single machine

The GridAspecting Process (VII)



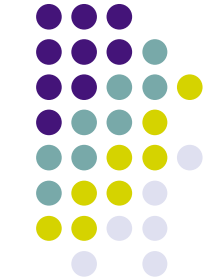
- Step 4 – Task parallelization using aspects
  - Allows parallel execution of application tasks using aspects
  - Aspects used to redirect thread creation and thread initialization calls to calls to the corresponding services provided by OurGrid
  - Task initialization aspect:
    - Makes task objects serializable (via introduction of the `Serializable` interface)
    - Makes task objects executable (via introduction of a `main` method)
    - The `main` method:
      - unpacks the serialized task object received as one of the task's input files at each grid machine
      - executes the task object's `run` method
      - serializes the task object to be returned back to the application by the grid as one of the task's output files

The GridAspecting Process (VIII)



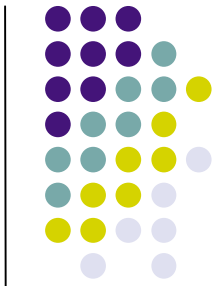
- Step 4 – Task parallelization using aspects (cont.)
  - Task execution aspect:
    - Intercepts the application flow at each task initialization time (calls to `start`) and when collecting their results (calls to `join`)
    - Defines join points and `around` type advices for methods `start` and `join`
    - Advice for method `start`:
      - Serializes the task object
      - Creates a new OurGrid task object and defines its appropriate input and output files
      - After all tasks have been created (i.e. last call to `start`) submits them as a single job to the OurGrid scheduler
    - Advice for method `join`:
      - Replaces the application's original task object with the serialized task object received from OurGrid

The GridAspecting Process (IX)



- Implementation restrictions
  - The task class must not define a main method
  - A task object must not access any static field of its own class or of any of its ancestors
  - A task object must store its results in its own instance attributes or in a single external file
  - A task object must not access any other external file unless its absolute path name is passed as an input parameter to the task constructor
- Restrictions necessary to guarantee that the base concurrent application satisfies the BoT parallel programming model

An Example



- Using GridAspecting to re-modularize GridSorter, a sample parallel sort application that is part of the OurGrid distribution
  - Two main methods: `sort` and `createSorterTaskSpec`

```
public Vector sort(Vector list) {
   MyGridSetter gridSetter = new MyGridSetter();
   gridSetter.setGrid(gridDescriptionFile);
   Configuration.reset();
   Configuration.getInstance(Configuration.MYGRID);
   JobSpec aJobSpec = new JobSpec("Sorter");
   …
   for (int i = 0; i < ntasks; i++) {
      File input = TempFileManager.createTempFile();
       // set input file
      taskList.add(createSorterTaskSpec (input.getPath()));
   }
   aJobSpec.setTaskSpecs(taskList);
   UIServices services = ConcreteUIServices.getInstance();
   int jobId = services.addJob(aJobSpec);
   services.waitForJob(jobId);
   …
}
```

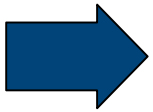
```
private TaskSpec createSorterTaskSpec(String inputFile) {
   IOBlock initBlock = new IOBlock();
   initBlock.putEntry(new IOEntry("STORE", inputFile,
                "sorter-$TASK.input"));
   String remoteScript = "nice sort $STORAGE/sorter-$TASK.input >
                sorter- $TASK.output";
   IOBlock finalBlock = new IOBlock();
   finalBlock.putEntry(new IOEntry("GET", "sorter-$TASK.output",
                "sorter-$TASK.output"));
   TaskSpec taskSpec = null;
   taskSpec = new TaskSpec(initBlock, remoteScript, finalBlock);
   return taskSpec;
}
```

Grid-related code shown in red

An Example (II)

- Re-implementing GridSorter using threads

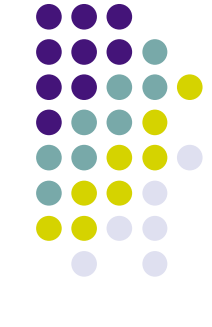
```
private TaskSpec createSorterTaskSpec(String inputFile) {
   IOBlock initBlock = new IOBlock();
   initBlock.putEntry(new IOEntry("STORE", inputFile,
                "sorter-$TASK.input"));
   String remoteScript = "nice sort $STORAGE/sorter-$TASK.input >
                sorter- $TASK.output";
   IOBlock finalBlock = new IOBlock();
   finalBlock.putEntry(new IOEntry("GET", "sorter-$TASK.output",
                "sorter-$TASK.output"));
   TaskSpec taskSpec = null;
   taskSpec = new TaskSpec(initBlock, remoteScript, finalBlock);
   return taskSpec;
}
```



```
public SorterThread createSorterTaskSpec(
                                String inputFile,
                                String outputFile){
   SorterThread gst = new SorterThread(
                                        inputFile,
                                        outputFile);
   return gst;
}
```

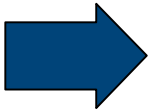
```
public class SorterThread extends Thread {
   private String inputFile = "";
   private String outputFile = "";
   public SorterThread (String inputFile,
                                String outputFile) {
      this.inputFile = inputFile;
      this.outputFile = outputFile;
   }
   public void run(){
      String cmd = "sort " + inputFile + " -o "
                        + outputFile;
      try {
         Runtime.getRuntime().exec(cmd);
      } catch (Exception e) {
          e.printStackTrace();
      }
   }
}
```

An Example (III)



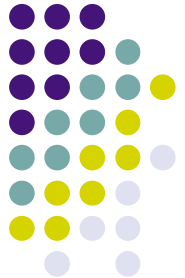
- Re-implementing GridSorter using threads (cont.)

```
public Vector sort(Vector list) {
   MyGridSetter gridSetter = new MyGridSetter();
   gridSetter.setGrid(gridDescriptionFile);
   Configuration.reset();
   Configuration.getInstance(Configuration.MYGRID);
   JobSpec aJobSpec = new JobSpec("Sorter");
   …
   for (int i = 0; i < ntasks; i++) {
      File input = TempFileManager.createTempFile();
       // set input file
      taskList.add(createSorterTaskSpec (input.getPath()));
   }
   aJobSpec.setTaskSpecs(taskList);
   UIServices services = ConcreteUIServices.getInstance();
   int jobId = services.addJob(aJobSpec);
   services.waitForJob(jobId);
   …
}
```



```
public Vector sort(Vector list) {
    …
    SorterThread[] taskList = new SorterThread[ntasks];
    for (int i = 0; i < ntasks; i++) {
       File input = TempFileManager.createTempFile();
       File output = TempFileManager.createTempFile();
         ...
       taskList[i] =  createSorterTaskSpec(
                        input.getPath(),output.getPath());
       taskList[i].start();
    }
    for (int i = 0; i < ntasks; i++) {
         taskList[i].join();
    }
    …
}
```

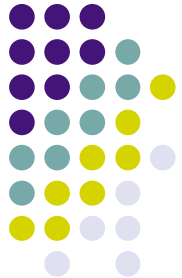
An Example (IV)



- GridSorter initialization aspect

```
public aspect InitializationAspect {
    declare parents: SorterThread implements Serializable;

    public static void SorterThread.main(String args[]) {
        SorterThread sorterThread;
        Deserializer deserializer = new Deserializer();
        Object obj = deserializer.execute(args[0]);
        String fileInput = args[1]; String fileOutput = args[2];
        sorterThread = (SorterThread) obj;
        sorterThread.setInputFile(fileInput); sorterThread.setOutputFile(fileOutput);
        sorterThread.run();
        String file = args[3];
        try{
            FileOutputStream fos = new FileOutputStream( file, true );
            BufferedReader reader = new BufferedReader(new
        FileReader(fileOutput));
            String line;
            while ((line = reader.readLine()) != null) {
                fos.write( ( line +"\n").getBytes() );
            }
            reader.close();
            fos.close();
        }
            catch( IOException fe ) {System.err.print( "Error can't write on file" ); }
    }
}
```

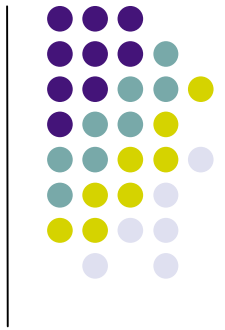
An Example (V)



- GridSorter execution aspect

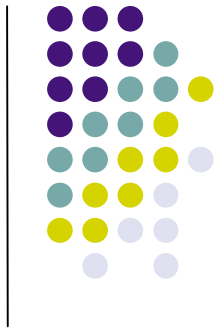
```
public aspect ExecutionAspect {
   pointcut start(SorterThread sorterThread):
            call (public void start()) && target(sorterThread);
   pointcut join(SorterThread sorterThread):
            call (public void join()) && target (sorterThread);

   void around (SorterThread sorterThread):  start(sorterThread){
       inputFileName = serializedFileName + idInput + ".file";
       Serializer serializer = new Serializer();
       serializer.execute(sorterThread,inputFileName);
       tasks.add(this.deployTask(inputFileName,
                   sorterThread.getInputFileName(),
                   sorterThread.getOutputFileName()));
       idInput++;
       if (idInput == ntasks) { // last call to start
          sendJob(tasks);
          idInput = 0;
       }
   }
   void around (SorterThread sorterThread): join(sorterThread){
       File file = new File(sorterThread.getOutputFile());
       while (!(file.exists()));
           idOutput++;
   }
}
```

Conclusions



- Proposal of an aspect-oriented implementation process for separation of grid concerns in the development of grid-based parallel Java applications
  - Improves the applications' modularity
  - Simplifies the grid programming model
  - Allows to functionally test the applications with or without the grid
- Process illustrated using a sample parallel sort application
- A more in-depth evaluation was performed using a real-world parallel application to compute optimal patrol routes for police forces (see paper for details)

Conclusions (II)



- Future work:
  - Investigate aspect-oriented metrics to evaluate modularity gains quantitatively
  - Generalization of the initialization and execution aspects to improve reuse
  - Use aspects for automatic verification of implementation restrictions
  - Extend the process to support inter-task communication and other less restricted parallel programming models

Thanks!

Nabor C. Mendonça
(nabor@unifor.br)