Improving the Separation of Parallel Code in Skeletal Systems

Diogo Telmo Neves
Departamento de Informática
Universidade do Minho
Portugal
dneves@di.uminho.pt

João Luís Sobral
Departamento de Informática
Universidade do Minho
Portugal
jls@di.uminho.pt

Abstract

This paper describes how to improve separation between domain-specific code and parallel code in skeletal systems. Traditionally, the code used to exploit parallelism is tangled among domain-specific code, which leads to problems such as: poor maintainability, lower flexibility, and weak scalability.

In this paper we introduce the design of the YaSkel framework, which is a support tool to write parallel programs. We argue that the design of YaSkel framework allows more freedom to change the parallelization strategy when compared with traditional skeleton frameworks. To change the parallelization strategy we rely on DI – Dependency Injection – to inject a reference of a specific skeleton in latter development stages. We also show that AOP – Aspect Oriented Programming – could be used to minimize the impact of applying skeleton based approaches to legacy code.

1. Introduction

The multicore technology have increased, and will increase even more [2], the processing power of (desktop) computers. In order to take advantage of such processing power, the software must be able to exploit hardware parallelism. Thus, it is expected that a great number of software developers will have to become parallel programmers because "all" programs will be parallel.

Pattern based parallel programming (PPP) address the complexity to develop parallel applications by providing a set of well known parallelization patterns, which can be applied "off-the-shelf" to build parallel applications. Portability can be ensured by providing efficient pattern implementations for multiple target platforms.

PPP considerably raises the abstraction level improving programmer's productivity, although, there are three important limitations of this kind of approach: (i) it is difficult to extend system provided patterns when they do not fit into application specific needs; (ii) the parallelization pattern must be selected at design time, which means that after developing the code it is hard to use a different parallelization pattern; (iii) it requires a complete reengineering of legacy code (pattern code is explicitly managed in, i.e. mixed with, user code).

The YaSkel framework aims to overcome these limitations by layering on top of inheritance, dependency injection [1] and class refinement [3]. Inheritance supports extensibility of system provided patterns through subclassing. Dependency injection allows the injection of the pattern implementation in latter development stages. Class refinement supports the replacement of a given class by its enhanced version in a type-compatible manner, which reduces the reengineering required to apply parallel patterns to legacy code.

The remainder of this paper is organized as follows. Section 2 presents an overview of YaSkel framework. Section 3 explains how to improve decoupling of parallel code in skeletal systems. Section 4 presents performance results. Section 5 compares this work against similar work. Finally, section 6 concludes the paper giving directions for future work.

2. YaSkel overview

YaSkel – Yet another Skeleton framework – is an extensible lightweight pattern-based parallel programming framework. As a lightweight framework is has a small footprint and, thereby, a small learning curve. The framework hides from the final user details and complexities of *how to develop a parallel code*. Thus, a developer can focus on writing their domain-specific code and not on issues that aren't from its domain, such as multithreading and parallel patterns.

In YaSkel, there are two major entities: `Task` and `Skeleton`. `Tasks` expose application dependent computations that can be performed in parallel. `Skeletons` are parallelization patterns that specify how `Tasks` are processed in parallel. The potential parallelism, specified in the form of `Tasks`, can be exploited by one or several YaSkel built-in `Skeletons`.

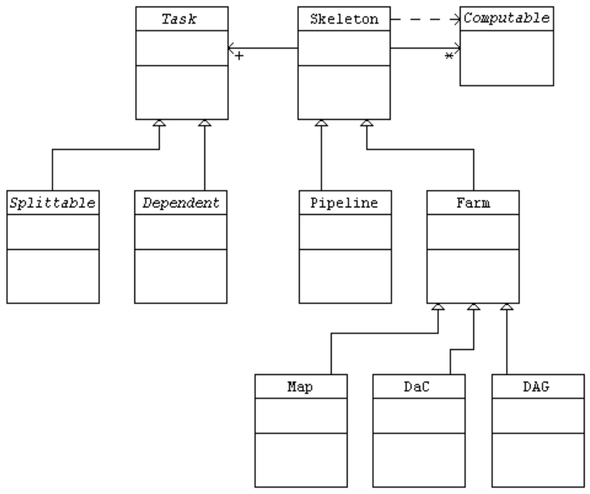

Figure 1. YaSkel simplified class diagram

Figure 1 shows a simplified class diagram of YaSkel, where it is possible to see that the framework is based on inheritance. Through inheritance we could extend either a `Skeleton` or a `Task` (for example, a `SplittableTask`, identified in Figure 1 as *Splittable*, is a specialization of `Task` that we could split into several ones). YaSkel is also based on software patterns. Software patterns are used to express high level parallel patterns, for example, problems that are embarrassingly parallel are solved with a `Skeleton` that exploits this kind of parallelism (e.g., `Farm`).

3. Improving decoupling

Traditional skeleton-based programming systems directly instantiate a skeleton in the domain specific code (see, for instance, the JaSkel framework [8]). Thus, after developing the code it is not possible to change the skeleton used (e.g., to change a Farm to a Pipeline parallelization).

DI [1] is a specific form of Inversion of Control (IoC), which allows to inject a reference (i.e., an object) into a class rather than let the class take the responsibility of creating the object. DI could be achieved through different ways, the main ones are: *Constructor Injection*; *Setter Injection*; and *Interface Injection*. We use the DI pattern for wiring domain-specific code with the skeleton that exploits parallelization, thus it's possible to reduce the dependency to a single reference.

Let us suppose the following two scenarios for the JGF Series [4] example: (i) all computations (all *A* and all *B terms*) will be computed by a single `Farm` skeleton, (ii) all *A terms* will be computed by a `Farm` skeleton and all *B terms* will be computed by another `Farm` skeleton. In scenario (i) we have a “simple” skeleton and in scenario (ii) we have a “composed” skeleton, a `Farm` of `Farms` skeleton (despite the fact that we could use two “simple” skeletons to perform the computation of this scenario). Through the use of the DI pattern it is possible to achieve a solution that addresses both scenarios.

First we write the domain specific code using a generic interface/class, `Skeleton`, to compute tasks:

```
class SeriesTest {

  private Skeleton<SeriesTask> skel;

  public void setSkeleton(
     Skeleton<SeriesTask> skeleton) {
    this.skel = skeleton;
  }
  void YaSkelDo() {
    ...
    futures = skel.compute();
    ...
  }
  ...
}
```

To apply the DI principle to the Series example we used *Setter Injection*, the `setSkeleton` method of `SeriesTest` class. The concrete skeleton will be provided latter by calling this setter method. The code is given next:

```
public class JGFSeriesBenchSizeC {

  public static void main(...) {

    SeriesTest se = new SeriesTest();

    if (args[0].equals("composed")) {
      se.setSkeleton(
          new ComposedSkeleton(...) );
    } else {
      se.setSkeleton(
          new SimpleSkeleton(...) );
    }
    se.JGFrun(2);
  }
}
```

The `JGFSeriesBenchSizeC` class is a client code of the Series example and has been rewritten to inject a specific skeleton onto the `SeriesTest` class (either `SimpleSkeleton` or a `ComposedSkeleton`). An alternative would be to develop two different `main` methods.

AOP offers an alternative way to implement DI. In this case, we write an aspect that injects the desired skeleton implementation after the creation of `SeriesTest` instances:

```
after() : call(SeriesTest.new(..)) ...
{
  SeriesTest st = proceed();
  st.setSkeleton(...);
}
```

We now present an example of how to apply YaSkel skeletons to legacy code using AOP techniques. The example is the RayTracer from the JGF.

The base code creates a `RayTracer` object that renders a given image `Interval`:

```
RayTracer rt = new RayTracer();
Interval interval =
       new Interval(0, width, height);

rt.render(interval);
```

In this case we can refine the `RayTracer` to become a `Task`, e.g., the computation is the call to the method `render` for a given interval.

```
declare parents : RayTracer
        extends Task<Interval, int[]>;

Interval RayTracer.myInterval;

public Task<Interval, int[]>
                     RayTracer.call() {
  this.render(myInterval);
  return(this);
}
```

Now we can write an aspect that delegates the execution of the original `render` method to the YaSkel framework:

```
int[] around(RayTracer rt, Interval intr) :
  call(* RayTracer.render(Interval)) ...
    // build tasks
    List<Interval> tasks = split(intr);
    List<RayTracer> farmTasks = ...;
    // build skeleton
    Skeleton<RayTracer> skeleton = ...;
    List<Future<RayTracer>> futures;
    skeleton.setTasks(farmTasks);
    ...
        // call YaSkel to compute tasks
        futures = skeleton.compute();
        for (Future<RayTracer> fut : futures) {
          RayTracer rt = fut.get();

          ... // merge results
        }
      }
    }
```

When the `render` method is called in the original code, the aspect splits the given interval and each sub-interval is associated to a `RayTracer` task. The set of `RayTracer` tasks is then processed by the YaSkel `Farm` skeleton.

As shown DI makes it possible to inject/change the parallelization strategy to be applied in the *legacy code* without change a single line of code. The YaSkel design allows us to change the skeleton used to parallelize user code and, in along with other technologies (e.g., AOP, DI), it is possible to minimize the code invasion; this is an unique feature of YaSkel.

4. Benchmarks

To assess the performance of the current implementation of the YaSkel framework we compared the execution time of two benchmarks from the Java Grande Forum [4]: the RayTracer (size B) and the Series (size A). Both benchmarks were parallelized with the `Farm` skeleton. Figure 2 compares the execution time of the YaSkel implementation against the JGF implementation based on Java Threads. Benchmarks were performed on a dual Xeon E5420 (a total of eight 2.5GHz cores) with 8 GB of RAM. This machine is part of a larger cluster that runs Rocks Linux and Sun JDK 1.5.0_07 64-bit Server VM.

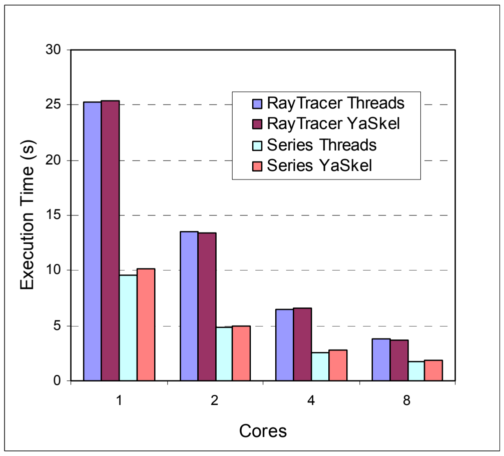

Figure 2. Performance assessment of YaSkel against Java Threads

Execution times for the RayTracer are roughly equivalent to the JGF Java threads implementation. The YaSkel implementation of the Series presents a slightly higher overhead.

YaSkel overheads are due to: task creation (e.g., to create an object to represent a `Task`); task submission /execution (currently the execution is delegated to a Java Executor); the synchronization to get the futures result. Finer grained tasks impose higher overheads, which is the case of the Series benchmark.

Note that AOP techniques can be used, in certain cases, to avoid the overhead of creating a `Task` instance. In that case we use class refinement to make a class in the base code to extend the `Task` class.

5. Related work

We focus the discussion of related work on research efforts on skeletons/patterns.

Skeletons [5] address the complexity of parallel programming by providing high level programming abstractions, modeling common, reusable, parallelism exploitation patterns. Skeletons provide a clear separation between skeleton user and skeleton implementation/exploitation. On the down side, they enforce a specific programming style based on the provided skeleton API (i.e., IoC), which makes the parallelization of legacy code invasive and preempts stronger changes to the parallelization strategy (only the pattern implementation can be changed). Other limitations include the limited pattern/skeleton extensibility to address new patterns.

Recent skeletal systems have used class hierarchy and inheritance to provide skeletons in the form of object oriented frameworks. Skeleton extensibility/adaptability has been supported by giving access to lower implementation layers [6]. Other systems focus on providing support for ad-hoc parallelism [7]. The JaSkel system [8] addresses extensibility of skeletons through OO inheritance and improves reuse of skeleton implementation code across platforms by relying on code generators to synthesize skeletons for Clusters and Grids from the shared memory implementation of skeletons [9] (which can be considered a kind of class refinement).

YaSkel differs from these systems by relying on DI and class refinements, which provides a better decoupling between the parallelization pattern. YaSkel is a pioneer framework to use DI and AOP techniques to compose skeleton/patterns into domain specific code.

6. Conclusions and future work

The primary contribution of the YaSkel framework is to provide a better decoupling between user code and parallel pattern implementation. For instance, we demonstrate that it is possible to change the parallelization strategy in latter development stages and, also, during execution. To achieve this goal we used the DI pattern as support to inject the desired skeleton in the code. The second contribution is showing that with YaSkel it is possible, in a seamlessly way, to parallelize *legacy code.*

The YaSkel framework already has support for parallel patterns such as Farm, Divide and Conquer, Map, Direct Acyclic Graph, and Pipeline. We plan to add more patterns in the future. We also intend to give support not only for regular applications but also for irregular applications, turning YaSkel a framework with support for optimistic parallelism.

7. Acknowledgments

This work was supported by the project AspectGrid (GRID/GRI/81880/2006) and Parallel Refinements for Irregular Applications (UTAustin/CA/0056/2008) funded by FCT and European funds (FEDER).

8. References

[1] http://martinfowler.com/articles/injection.html
[2] S. Borkar, "Thousand core chips: a technology perspective". 44th Annual Conference on Design Automation (DAC '07), San Diego, California, June 2007).
[3] D. Batory, J. Sarvela, A. Rauschmanyer, "Scaling Sep-Wise Refinement", IEEE Transactions on Software Engineering, 30(6), June 2004.
[4] A: Smith. J. Bull, J. Obdrzálek, "A Parallel Java Grande Benchmark Suite", Supercomputing 2001. Denver. November 2001.
[5] D. Cole, "Algorithmic Skeletons: structured management of parallel computation", Pitman/MIT press, 1989.
[6] K Tan, D Szafron, J Schaeffer, J Anvik, S , "Using generative design patterns to generate parallel code for a distributed memory environment", PPoPP 2003.
[7] A Benoit, M Cole, S Gilmore, J Hillston, "Flexible Skeletal Programming with eSkel", EuroPar 2005.
[8] L. Fernando, J. Sobral, A. Proenca. "JaSkel: A Java Skeleton-Based Framework for Structured Cluster and Grid Computing", CCGrid'2006, Singapore, May 2006.
[9] J. Sobral, A. Proença, "Enabling JaSkel Skeletons for Clusters and Computational Grids", Cluster'07, Austin, Texas, September 2007.