ParaFormance™: An Advanced Refactoring Tool for Parallelising C++ Programs – Part 2

Chris Brown, Vladimir Janjic, Kevin Hammond
University of St Andrews, Scotland
@chrismarkbrown
@rephrase_eu
Outline

1. Introduction and overview to Insertion
2. Live Demonstration of Insertion
3. Follow along interactively
4. Image Convolution and Ant Colony examples
What is Insertion?

1. Introduces parallelism into an application semi-automatically
2. Refactors a sequential portion of code into a parallel version
3. Introduces all parallel ‘business logic’
Inserting a Farm

\[ S \equiv Farm(S) \]

\textit{farm intro/elim}
Inserting a Pipeline

\[ S_1 \circ S_2 \equiv Pipe(S_1, S_2) \]

**Figure 3.3: Some Standard Skeleton Equivalences**

The following describes each of the patterns in turn:

- **a MAP** is made up of three \( \text{OPERATION} \)s: a worker, a partitioner, and a combiner, followed by an \( \text{INPUT} \);
- **a SEQ** is made up of a single \( \text{OPERATION} \) denoting the sequential computation to be performed, followed by an \( \text{INPUT} \);
- **a FARM** is made up of a single \( \text{OPERATION} \) denoting the working, an \( \text{INPUT} \) and the number of workers (\( NW \)); and,
- **a PIPE** is made up of at least one \( \text{OPERATION} \) denoting each stage of the pipeline, followed by an \( \text{INPUT} \).

Each pattern has an \( \text{IDENT} \) attribute, allowing other patterns to reference it. This flattens the XML structure instead of nesting, providing a structure that is easier to read and reason about. Finally, an \( \text{OPERATION} \) is either a \( \text{COMPONENT} \) (describing the module, name and language for a unit of computation) or another pattern (described by a reference to the pattern’s identifier).

### 3.2 Language-Independent Rewrite Rules

In this section we introduce a number of language-independent skeleton rewrite rules that will be used to motivate the refactoring rules given in the remainder of the chapter. Figure 3.3 shows four well-known equivalences. Here \( \text{Pipe} \), \( \text{Map} \) and \( \text{Seq} \) refer to the skeletons for parallel pipeline, function composition and parallel map, respectively, \( S \) denotes any skeleton, and \( \text{Seq}(e) \) denotes that \( e \) is a sequential computation. All skeletons work over streams. \( \text{Decomp} \) and \( \text{Recomp} \) are primitive operations that work over non-streaming inputs. \( d \) and \( r \) are simple functions that specify a decomposition (\( d : a \rightarrow [b] \)), and a recomposition (\( r : [b] \rightarrow a \), as with the \( \text{Partition} \) and \( \text{Combine} \) functions above. Working from left to right in these equivalences introduces parallelism, while working from right to left reduces parallelism. Reading from left to right, we can therefore interpret the rules as follows:

- **Stage 1**
- **Stage 2**
- **Thread 1**
- **Thread 2**
Why?

• Huge saving in effort over manual
• Difficult to get right!
• We have seen 40 hours of manual programming effort reduced into 5 hours work (a 8x productivity improvement)
• The programmer doesn’t have to learn lots of boilerplate
• The programmer doesn’t have to keep track of library/standards changing
Image Convolution
Image Convolution, Refactored
THANK YOU!

http://rephrase-ict.eu

@rephrase_eu