H2020 FETHPC-1-2014

An Exascale Programming, Multi-objective Optimisation and Resilience Management Environment Based on Nested Recursive Parallelism

Project Number 671603

D3.3 – AllScale Compiler Prototype

WP3: High-level parallel API and API-aware optimising source-to-source compiler

Version: 1.0
Author(s): Peter Thoman (UIBK)
Date: 2016-12-28
| **Due date:** | PM15 |
| **Submission date:** | 28/12/2016 |
| **Project start date:** | 01/10/2015 |
| **Project duration:** | 36 months |
| **Deliverable lead organization** | UIBK |
| **Version:** | 1.0 |
| **Status** | Final |
| **Author(s):** | Peter Thoman (UIBK) |
| **Reviewer(s)** | Kiril Dichev (QUB)  
Khalid Hasanov (IBM) |

| **Dissemination level** |
| **PU** | *Public* |

**Disclaimer**

This deliverable has been prepared by the responsible Work Package of the Project in accordance with the Consortium Agreement and the Grant Agreement Nr 671603. It solely reflects the opinion of the parties to such agreements on a collective basis in the context of the Project and to the extent foreseen in such agreements.
Acknowledgements

The work presented in this document has been conducted in the context of the EU Horizon 2020. AllScale is a 36-month project that started on October 1st, 2015 and is funded by the European Commission.

The partners in the project are UNIVERSITÄT INNSBRUCK (UBIK), FRIEDRICH-ALEXANDER-UNIVERSITÄT ERLANGEN NÜRNBERG (FAU), THE QUEEN’S UNIVERSITY OF BELFAST (QUB), KUNGLIGA TEKNISKA HÖGSKOLAN (KTH), NUMERICAL MECHANICS APPLICATIONS INTERNATIONAL SA (NUMECA), IBM IRELAND LIMITED (IBM).

The content of this document is the result of extensive discussions within the AllScale Consortium as a whole.

More information

Public AllScale reports and other information pertaining to the project are available through the AllScale public Web site under http://www.allscale.eu.

Version History

<table>
<thead>
<tr>
<th>Version</th>
<th>Date</th>
<th>Comments, Changes, Status</th>
<th>Authors, contributors, reviewers</th>
</tr>
</thead>
<tbody>
<tr>
<td>0.1</td>
<td>16-11-30</td>
<td>First draft</td>
<td>Peter Thoman</td>
</tr>
<tr>
<td>0.2</td>
<td>16-11-30</td>
<td>Integrated comments from Roman</td>
<td>Peter Thoman, Roman lakymchuk</td>
</tr>
<tr>
<td>0.3</td>
<td>16-12-20</td>
<td>Source location moved to github, integrated Erwin's feedback</td>
<td>Peter Thoman, Erwin Laure</td>
</tr>
<tr>
<td>0.4</td>
<td>16-12-20</td>
<td>Integrated Kiril's feedback</td>
<td>Peter Thoman, Kiril Dichev</td>
</tr>
<tr>
<td>1.0</td>
<td>16-12-28</td>
<td>Final version</td>
<td>Peter Thoman</td>
</tr>
</tbody>
</table>
# Table of Contents

Executive Summary .................................................................................................................. 5  
1  Introduction .......................................................................................................................... 6  
2  Build instructions ................................................................................................................ 6  
3  Component Overview .......................................................................................................... 7  
   3.1  The AllScale Compiler Core .......................................................................................... 8  
   3.2  The AllScale Compiler Frontend .................................................................................. 8  
   3.3  The AllScale Compiler Backend ................................................................................ 9  
4  Summary and Future Work ................................................................................................. 10
Executive Summary

This document describes the status and structure of the AllScale Compiler Prototype (D3.3). This is a source code deliverable. Instructions on how to obtain, build and run the prototype will be provided as well as an overview of the current implementation and its components.
1 Introduction

Deliverable D3.3 is a source code deliverable providing the first early prototype of the AllScale Compiler. This prototype maps the basic core primitives of the AllScale API (D3.1) – the pfor and fun calls as well as treetures and operations on them – to the runtime interface specified by deliverables D4.1 and D4.2.

It fits into the entire AllScale software infrastructure as indicated in Figure 1, depending on the external Insieme compiler as an underlying implementation technology.

![Figure 1: AllScale Compiler Relationship to other Software Components](image)

The major components of this prototype are the intermediate representation of the AllScale Core API, its semantics-aware frontend translation, and the backend code generation conforming to the runtime interfaces.

The remainder of this document will provide instructions on how to build the prototype as well as an overview of the implementation.

2 Build instructions

The AllScale Compiler is based on the Insieme research compiler infrastructure, which is an open source project publicly available on github: https://github.com/insieme/insieme

This infrastructure in turn has a set of dependencies, as well as a simplified installation script for those dependencies. These topics are discussed in the Insieme project readme.

A second dependency of the AllScale Compiler Prototype for functional testing is the AllScale Runtime System prototype. Installation instructions for it are available in deliverable D4.2.
D3.3 – AllScale Compiler Prototype

The source code of the Allscale Compiler Prototype is hosted on github at this URL: https://github.com/allscale/allscale_compiler. Note that the repository uses submodules, so it should be cloned with `git clone --recursive`.

Once the sources have been obtained either via the Git SCM or a downloadable snapshot, the build process is configured using CMake, which interfaces with the native build tools of the given platform. A set of unit tests is included, and it can be built and executed by calling `make test`. The main compiler driver is `allscalecc`, which can be executed with the same flags as those supported by most C/C++ compilers.

The code is organized as follows:
- `<project root>`
  - `api` – a submodule housing the AllScale API
  - `code` – containing the source code for the AllScale Compiler
    - `include` – headers and interfaces
    - `src` – compiler sources
    - `test` – unit tests
  - `insieme` – a submodule for the Insieme compiler infrastructure
  - `runtime` – containing the AllScale Runtime and HPX for testing code generated by the compiler backend
  - `test` – input files for integration testing
  - `CMakeLists.txt` – cmake build script
  - `README.md` – short summary and usage hints

3 Component Overview

The AllScale Compiler Prototype is split into three distinct parts:
- the Allscale Compiler **Frontend**, tasked with semantic translation of the AllScale core API to the compiler intermediate representation (IR);
- the Allscale Compiler **Core**, which provides an intermediate representation of the semantic information relevant for the compilation of AllScale programs;
- and the AllScale Compiler **Backend**, which performs all tasks required to generate the program representation expected by the AllScale Runtime System from the intermediate representation of a program.
3.1 The AllScale Compiler Core

As the intermediate language provided by this component is both the target output for the frontend as well as the input for the backend, we will start by providing an overview of this component. We will use INSPIRE syntax in the descriptions in this section, for reference consider the 2014 thesis by Jordan¹.

The AllScale IR module defines two types central to AllScale, as well as the operations to build and manipulate these types. They are:

- **recfun<'a,'b>**, which describes a recursive task with parameter ‘a and return type ‘b.
- **treeture<'t,'r>**, describing a tree-synchronized future (“treeture”) of type ‘t which can be either released (‘r = t) or unreleased (‘r = f). The latter indicates a treeture which can receive additional dependencies, while the former is already launched (note that this is an implementation refinement which occurred after the current API deliverable was completed).

The central **prec** operator is then typed as follows:

```
( (recfun<'a,'b>, 'c...) ) -&gt; ( 'a ) =&gt; treeture<'b,f>
```

In addition to these language definitions and operations (in the lang namespace) as well as some auxiliary helper functions, the core module also contains a set of checks designed to ensure the semantic integrity of a given IR fragment that makes use of the operators provided by the AllScale IR module. An example of such a check would be ensuring that the function types of all the individual closures passed to the prec operator call fit the overall type of the recursive function being constructed.

3.2 The AllScale Compiler Frontend

The Frontend of the AllScale compiler is implemented as a FrontendExtension within the Insieme infrastructure. This mechanism allows fine-grained control over the translation of C++ types, expressions and declarations to the INSPIRE intermediate representation used in Insieme.

This component is tasked with **semantics-aware translation** of the AllScale API primitives. In practice, this means that the translation process is aware of the meaning of each core API primitive independent of its specific C++ implementation. Individual components present in the C++ program are translated to a semantically (but often not syntactically) equivalent representation based on INSPIRE and the AllScale IR module.

Since it needs to extract all relevant information from an often complex representation of C++ template instantiations provided by Clang, the AllScale

---

¹ Insieme - A Compiler Infrastructure for Parallel Programs. UIBK 2014.
Compiler Frontend features some rather intricate interdependencies. These are illustrated in Figure 3.

![Figure 3: AllScale Compiler Frontend Processing](image)

### 3.3 The AllScale Compiler Backend

Similarly to the AllScale Compiler Frontend, the AllScale Compiler Backend makes use of the various mechanisms for extending the base compilation process provided in the Insieme infrastructure.

First, a series of preprocessing steps is performed, each of which encodes a more low-level representation of some AllScale structure in the original program in INSPIRE. This representation is subsequently used to generate the final C++ output code. The preprocessor steps currently include:

- The **PrecConverter**, which generates work item descriptions from calls to the prec operator.
- The **EntryPointWrapper**, encoding the program entry point in a format executable by the AllScale runtime system.
- The **CppLambdaToBindConverter**, which converts lambda-to-bind calls to a format which can be processed in the remainder of the backend.

The intermediate representation generated by these preprocessing passes is subsequently processed by the general backend to generate a C++ abstract syntax tree (AST).
In this process, additional type handlers and operator converters are installed. For types, the IR \textit{treeture} type is translated to the allscale runtime \textit{treeture}, and INSPIRE tuples are translated to the HPX \textit{tuple} type. During operator conversion, work item spawning as well as operations on \textit{treetures} are translated to the corresponding runtime system invocations.

4 Summary and Future Work

The initial compiler prototype as of the submission of this deliverable is capable of translating programs using a subset of the AllScale API (without full data item support) to the interfaces provided by the AllScale Runtime System. It represents all relevant semantics in the INSPIRE intermediate representation, which is crucial for achieving the full goals of WP3, particularly the data dependency analysis required for the full prototype. Besides the ongoing task of adapting compiler frontend and backend to support expanded API and runtime functionality, data dependency analysis for the work item functions created for each \texttt{pfor} invocation will be the major focus of the compiler work package during the upcoming phase of the project. This analysis is essential in order to enable the migration of data-dependent work items in a distributed memory system.