D2.2 – Requirement Specifications and Reports on External Technological Developments (b)

WP2: Requirements and overall system architecture design

Version: 3.0
Author(s): Roman Iakymchuk (KTH), Herbert Jordan (UIBK), Peter Thoman (UIBK), Thomas Heller (FAU), Stéphane Monté (NUMECA), Benoît Leonard (NUMECA), Emanuele Ragnoli (IBM), Fearghal O. Donncha (IBM), Khalid Hasanov (IBM), Xavier Aguilar (KTH), Dana Akhmetova (KTH), Kiril Dichev (QUB), Thomas Fahringer (UIBK)
Date: 31/07/17
# D2.2 – Requirement Specifications and Reports on External Technological Developments (b)

<table>
<thead>
<tr>
<th><strong>Due date:</strong></th>
<th>PM22</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Submission date:</strong></td>
<td>31/07/2017</td>
</tr>
<tr>
<td><strong>Project start date:</strong></td>
<td>01/10/2015</td>
</tr>
<tr>
<td><strong>Project duration:</strong></td>
<td>36 months</td>
</tr>
<tr>
<td><strong>Deliverable lead organization</strong></td>
<td>Royal Institute of Technology (KTH)</td>
</tr>
<tr>
<td><strong>Version:</strong></td>
<td>3.0</td>
</tr>
<tr>
<td><strong>Status</strong></td>
<td>Final</td>
</tr>
<tr>
<td><strong>Author(s):</strong></td>
<td>Roman Iakymchuk (KTH), Herbert Jordan (UIBK), Peter Thoman (UIBK), Thomas Heller (FAU), Stéphane Monté (NUMECA), Benoît Leonard (NUMECA), Emanuele Ragnoli (IBM), Fearghal O. Donncha (IBM), Khalid Hasanov (IBM), Xavier Aguilar (KTH), Dana Akhmetova (KTH), Kiril Dichev (QUB), Thomas Fahringer (UIBK)</td>
</tr>
<tr>
<td><strong>Reviewer(s)</strong></td>
<td>Philipp Gschwandtner (UIBK), Stéphane Monté (NUMECA)</td>
</tr>
</tbody>
</table>

## Dissemination level

- **PU**  
  - **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.
# 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>01/12/15</td>
<td>First draft on the pilot applications</td>
<td>Benoît Leonard, Emanuele Ragnoli, Dana Akhmetova, Roman Iakymchuk</td>
</tr>
<tr>
<td>0.2</td>
<td>04/12/15</td>
<td>First draft on the external technological developments</td>
<td>Thomas Heller, Peter Thoman, Kiril Dichev, Roman Iakymchuk</td>
</tr>
<tr>
<td>0.3</td>
<td>09/12/15</td>
<td>First draft on the AllScale Environment</td>
<td>Roman Iakymchuk, Xavier Aguilar, Thomas Heller, Peter Thoman, Kiril Dichev</td>
</tr>
<tr>
<td>0.3.1</td>
<td>10/12/15</td>
<td>Fixed fonts and styling as well as the AMDADOS part</td>
<td>Roman Iakymchuk</td>
</tr>
<tr>
<td>0.4</td>
<td>18/12/15</td>
<td>Completed the AllScale APIs and Compiler parts; Moved some parts on Fine/Open to Annex; Improved the iPIC3D part</td>
<td>Benoît Leonard, Herbert Jordan, Peter Thoman, Roman Iakymchuk</td>
</tr>
<tr>
<td>0.4.1</td>
<td>21/12/15</td>
<td>Improved the pilot applications, especially benchmarks, with respect to the AllScale Environment requirements on them</td>
<td>Benoît Leonard, Roman Iakymchuk</td>
</tr>
<tr>
<td>0.5</td>
<td>21/12/15</td>
<td>Added Introduction</td>
<td>Roman Iakymchuk</td>
</tr>
<tr>
<td>0.5.1</td>
<td>13/01/15</td>
<td>Revised iPIC3D part and improved the iPIC3D benchmarks</td>
<td>Roman Iakymchuk</td>
</tr>
<tr>
<td>0.5.2</td>
<td>19/01/15</td>
<td>Revised</td>
<td>Kiril Dichev</td>
</tr>
<tr>
<td>0.5.3</td>
<td>26/01/15</td>
<td>Revised the Runtime System and other parts</td>
<td>Thomas Heller, Fearghal O. Donncha, Roman Iakymchuk</td>
</tr>
<tr>
<td>0.5.4</td>
<td>01/02/16</td>
<td>Reviewed FINE™/Open benchmark</td>
<td>Benoît Leonard</td>
</tr>
<tr>
<td>0.5.5</td>
<td>02/02/16</td>
<td>Revised</td>
<td>Roman Iakymchuk</td>
</tr>
<tr>
<td>0.5.6</td>
<td>05/02/16</td>
<td>Improved Sections 2.2-3 and Section 3.4</td>
<td>Thomas Fahringer and Xavier Aguilar</td>
</tr>
<tr>
<td>0.5.7</td>
<td>08/02/16</td>
<td>Added objectives</td>
<td>Roman Iakymchuk</td>
</tr>
<tr>
<td>0.5.8</td>
<td>22/02/16</td>
<td>Incorporated internal reviewers comments</td>
<td>Herbert Jordan, Thomas Heller, Peter Thoman, Kiril Dichev, Xavier Aguilar, Fearghal O. Donncha, Roman Iakymchuk</td>
</tr>
<tr>
<td>0.5.9</td>
<td>10/03/16</td>
<td>Incorporated internal reviewers comments</td>
<td>Benoît Leonard</td>
</tr>
</tbody>
</table>
D2.2 – Requirement Specifications and Reports on External Technological Developments (b)

<table>
<thead>
<tr>
<th>Version</th>
<th>Date</th>
<th>Description</th>
<th>Author</th>
</tr>
</thead>
<tbody>
<tr>
<td>0.6</td>
<td>31/03/16</td>
<td>Updated section 2.2 and 2.3 to reflect new structure decided at Erlangen</td>
<td>Peter Thoman</td>
</tr>
<tr>
<td>0.6.1</td>
<td>27/04/16</td>
<td>Incorporated Section 3.6 (AllScale Scheduler)</td>
<td>Khalid Hasanov</td>
</tr>
<tr>
<td>0.7</td>
<td>01/06/16</td>
<td>Improved section 2.2, 2.4 and 5</td>
<td>Thomas Heller</td>
</tr>
<tr>
<td>0.8</td>
<td>12/07/16</td>
<td>Enhanced objectives according to the version in the periodic report on M09</td>
<td>Roman Iakymchuk</td>
</tr>
<tr>
<td>0.9</td>
<td>17/10/16</td>
<td>Added executive summary and conclusions; improved introduction</td>
<td>Roman Iakymchuk</td>
</tr>
<tr>
<td>1.0</td>
<td>17/10/16</td>
<td>Reviewed and enhanced the document</td>
<td>Roman Iakymchuk</td>
</tr>
<tr>
<td>1.1</td>
<td>18/10/16</td>
<td>Scheduling sub-section (2.5) has been added into Section 2</td>
<td>Khalid Hasanov</td>
</tr>
<tr>
<td>1.2</td>
<td>19/10/16</td>
<td>Enhanced the sub-section 2.5</td>
<td>Khalid Hasanov</td>
</tr>
<tr>
<td>1.3</td>
<td>19/10/16</td>
<td>Enhanced the FINE™/Open part</td>
<td>Stéphane Monte</td>
</tr>
<tr>
<td>1.4</td>
<td>24/10/16</td>
<td>Improved the resilience section (3.6)</td>
<td>Kiril Dichev</td>
</tr>
<tr>
<td>1.5</td>
<td>22/11/16</td>
<td>Incorporated a major part of the reviewers’ comments</td>
<td>Roman Iakymchuk</td>
</tr>
<tr>
<td>1.6</td>
<td>23/11/16</td>
<td>Incorporated the reviewers’ comments on the sections 2.5 and 3.4</td>
<td>Khalid Hasanov</td>
</tr>
<tr>
<td>1.7</td>
<td>25/11/16</td>
<td>Incorporated the reviewers’ comments on the sections 4.2 and 7</td>
<td>Stéphane Monté</td>
</tr>
<tr>
<td>1.8</td>
<td>25/11/16</td>
<td>Incorporated the reviewers’ comments on the sections 2.2, 3.1, and 3.2</td>
<td>Peter Thoman</td>
</tr>
<tr>
<td>1.9</td>
<td>25/11/16</td>
<td>Incorporated the reviewers’ comments on the sections 2.1 and 3.3</td>
<td>Thomas Heller</td>
</tr>
<tr>
<td>2.0</td>
<td>25/11/16</td>
<td>Final version of D2.1</td>
<td>Roman Iakymchuk</td>
</tr>
<tr>
<td>2.1</td>
<td>12/06/16</td>
<td>Updated sections 2.6 and 3.5</td>
<td>Xavier Aguilar</td>
</tr>
<tr>
<td>2.2</td>
<td>13/06/17</td>
<td>Updated sections 3.1 and 3.2</td>
<td>Peter Thoman, Herbert Jordan</td>
</tr>
<tr>
<td>2.3</td>
<td>14/06/17</td>
<td>Updated Sections 2.7 and 3.6</td>
<td>Kiril Dichev</td>
</tr>
<tr>
<td>2.4</td>
<td>15/06/17</td>
<td>Updated Sections 2.5 and 3.4</td>
<td>Khalid Hasanov</td>
</tr>
<tr>
<td>2.5</td>
<td>16/06/17</td>
<td>Updated Section 2.2</td>
<td>Peter Thoman</td>
</tr>
<tr>
<td>2.6</td>
<td>22/06/17</td>
<td>Updated Sections 4.2 and 7</td>
<td>Stéphane Monte</td>
</tr>
<tr>
<td>2.7</td>
<td>23/06/17</td>
<td>Updated Sections 2.4 and 3.3</td>
<td>Thomas Heller</td>
</tr>
<tr>
<td>2.8</td>
<td>27/06/17</td>
<td>Updated Section 4.3</td>
<td>Fearghal O Donncha</td>
</tr>
<tr>
<td>2.9</td>
<td>30/06/17</td>
<td>Updated Section 4.1</td>
<td>Roman Iakymchuk</td>
</tr>
<tr>
<td>2.9.1</td>
<td>01/07/17</td>
<td>Draft for internal review</td>
<td>Roman Iakymchuk</td>
</tr>
<tr>
<td>2.9.2</td>
<td>12/07/17</td>
<td>Incorporated reviewers’ comments</td>
<td>Roman Iakymchuk</td>
</tr>
<tr>
<td>3.0</td>
<td>19/07/17</td>
<td>Final version of D2.2</td>
<td>Roman Iakymchuk</td>
</tr>
</tbody>
</table>
# Table of Contents

Executive Summary ........................................................................................................... 9

1 Introduction .................................................................................................................... 11

2 Exascale Technologies: External Technological Developments ......................... 12
   2.1 Hardware Development ....................................................................................... 12
   2.2 Application Programming Interface (API) ....................................................... 12
   2.3 Compilers .......................................................................................................... 15
   2.4 Runtime Systems ............................................................................................... 15
   2.5 Scheduling in Many-Task Runtime Systems .................................................... 16
   2.6 Monitoring Frameworks ..................................................................................... 18
   2.7 Fault Tolerance and Resilience Approaches ..................................................... 18

3 AllScale Environment .................................................................................................. 21
   3.1 AllScale API ....................................................................................................... 21
      3.1.1 Requirements on the AllScale APIs ............................................................ 22
      3.1.2 Requirements Towards the Other Components ......................................... 22
   3.2 AllScale Compiler ............................................................................................. 24
      3.2.1 Requirements on the AllScale Compiler .................................................. 24
      3.2.2 Requirements Towards the Other Components ......................................... 25
   3.3 AllScale Runtime System .................................................................................. 26
      3.3.1 Requirements on the AllScale Runtime System ....................................... 26
      3.3.2 Requirements Towards the Other Components ......................................... 26
   3.4 AllScale Scheduler ............................................................................................ 27
      3.4.1 Requirements on the Allscale Scheduler .................................................... 27
      3.4.2 Requirements Towards the Other Components ......................................... 28
   3.5 AllScale Monitoring Framework ....................................................................... 28
      3.5.1 Requirements on the AllScale Monitoring Framework .............................. 29
      3.5.2 Requirements Towards the Other Components ......................................... 29
   3.6 AllScale Resilience and Fault Tolerance Management ....................................... 29
      3.6.1 Requirements on the AllScale Resilience and Fault Tolerance Management 29
      3.6.2 Requirements Towards the Other Components ......................................... 30

4 Pilot Applications .......................................................................................................... 31
   4.1 iPIC3D: Implicit Particle-in-Cell Code for Space Weather Applications ....... 31
      4.1.1 Introduction ............................................................................................... 31
      4.1.2 Scientific / Commercial Impact / Stakeholders ....................................... 31
      4.1.3 Exascale Potential ..................................................................................... 32
      4.1.4 Current Parallelization Approach .............................................................. 32
      4.1.5 AllScale Potential ..................................................................................... 33
      4.1.6 Requirements Specification ....................................................................... 34
      4.1.7 Benchmarks ............................................................................................... 35
   4.2 FINE™/Open: Unstructured CFD solver for large industrial unsteady simulations .................................................................................................................. 36
      4.2.1 Introduction ............................................................................................... 36
      4.2.2 Scientific / Commercial Impact / Stakeholders ....................................... 36
      4.2.3 Exascale Potential ..................................................................................... 37
      4.2.4 Current Parallelization Approach .............................................................. 37
      4.2.5 AllScale Potential ..................................................................................... 39
      4.2.6 Requirements Specification ....................................................................... 39
      4.2.7 Benchmarks ............................................................................................... 43
D2.2 – Requirement Specifications and Reports on External Technological Developments (b)

4.3 AMDADOS: Adaptive Meshing and Data Assimilation for the Deepwater Horizon Oil Spill (IBM) .......................................................... 45
  4.3.1 Introduction ........................................................................... 45
  4.3.2 Scientific / Commercial Impact / Stakeholders ......................... 46
  4.3.3 Exascale Potential .................................................................. 46
  4.3.4 AllScale Potential .................................................................. 46
  4.3.5 Requirements Specification ...................................................... 47
  4.3.6 Benchmarks .......................................................................... 48

4.4 Benchmarking Environment ......................................................... 48

5 AllScale Objectives ........................................................................ 50

6 Conclusions and Future Work ......................................................... 54

7 Annex: FINE™/Open .................................................................... 55
  7.1 Full-HEX mesh ........................................................................ 55
    7.1.1 Refinement for Full-HEX mesh ............................................ 55
    7.1.2 Hanging node ..................................................................... 55
    7.1.3 Combination of refinement .................................................. 57
  7.2 Flow solver: stencil – data structure – multigrid ....................... 58
    7.2.1 Stencils .............................................................................. 58
    7.2.2 V-cycle multigrid ................................................................. 60

Index of Figures

Figure 2: AllScale Architecture ............................................................... 21
Figure 3: Simulation with iPIC3D of plasma particles interacting with Earth magnetic field lines (black lines) ......................................................... 31
Figure 4: Time-stepping of one component in the recursive PIC algorithm .......................................................... 33
Figure 5: PIC data movement at individual cell level ................................ 33
Figure 6: Earth’s (Van Allen) radiation belts with satellites. Credit: NASA .......................................................... 34
Figure 7: LES of transition at two different inlet turbulence intensities .......................................................... 36
Figure 8: Scalability study ................................................................. 38
Figure 9: Full HEX mesh without viscous layers .................................. 40
Figure 10: Hybrid mesh without viscous layers ..................................... 40
Figure 11: ‘Viscous layer’ cells .......................................................... 41
Figure 12: Fine mesh – Coarse mesh for DPW4 Euler mesh .................. 42
Figure 13: Details of the region for DA-AM (black contour) and sources of observations (dots) ......................... 45
Figure 14: ξηζ-refinement ................................................................. 55
Figure 15: ξηζ-refinement ................................................................. 55
Figure 16: ξηζ-refinement ................................................................. 55
Figure 17: Hanging node .................................................................. 56
Figure 18: Hanging node .................................................................. 57
Figure 19: Sequence of refinements ..................................................... 57
Figure 20: Impossible combination of refinement ................................ 58
Figure 21: Stencil for Euler – Upwind 1st order – Finest level ............. 58
Figure 22: Stencil for Euler – Upwind 1st order – Coarse level ............ 58
Figure 23: Stencil for Euler – Upwind 2nd order – Finest level ............ 58
Figure 24: Stencil for Euler – Upwind 2nd order – Coarse level .......... 59
Figure 25: Stencil for Viscous – Upwind 1st order – Finest level .......... 59
Figure 26: Stencil for Viscous – Upwind 1st order – Coarse level .......... 59
Figure 27: Stencil for Viscous – Upwind 2nd order – Finest level .......... 59
Figure 28: Stencil for Viscous – Upwind 2nd order – Coarse level .......... 60
Figure 29: Sketch of V-cycle for Central scheme ................................ 61
Figure 30: Sketch of V-cycle for Central scheme ................................ 61

Copyright © AllScale Consortium Partners 2017
Index of Tables

Table 1: Realistic test cases for iPIC3D................................................................. 35
Table 2: Spectrum of approaches to turbulence (*assuming Moore’s Law holds) ............... 37
Table 3: Realistic test cases for FINE™/Open................................................................ 44
Table 4: Computational cost of AM and DA .................................................................... 45
Table 5: AMDADOS test cases ranging from tiny to real-world applications .................... 48
Executive Summary
This document is an update to Deliverable 2.1 (submitted on PM14) that
• identifies external technological developments potentially influencing the project objectives and requirements;
• describes requirements of the project with respect to the AllScale Environment components, the pilot applications, and the project objectives;
• specifies pilot applications’ benchmarks for performance evaluations.

As agreed, the text of Deliverable 2.2 is integrated within the text from Deliverable 2.1. To note, there are mainly minor changes compared to Deliverable 2.1 since the major work on the requirements, the project objectives, and the external technology watch function was completed by PM14. We highlight the corresponding changes as follows
• on external technological developments
  o an update to the hardware development section is mainly focused on reflecting upon the current trends;
  o the API section is complemented with our classification matrix of the task-parallel APIs, which was presented in more details in our recently accepted article (‘A Taxonomy of task-based technologies for High-Performance Computing’) to the PPAM conference;
  o minor updates for the compilers, runtimes, and scheduler sections. For instance, Legion is included into the scope;
  o the monitoring section is updated with the clarification on the Legion’s and OmpSs’s monitoring capabilities;
  o the fault tolerance and resilience section is refactored with an update on self-stabilization and self-healing strategies.
• the AllScale Environment
  o AllScale API: minor updates on treetures; modification to the requirements for the pilot applications’ benchmarks;
  o AllScale Compiler: minor clarification of the requirements towards the AllScale Runtime System;
  o AllScale Runtime System: minor changes;
  o AllScale Scheduler: minor changes;
  o AllScale Monitoring Framework: clarification regarding the notifications to be sent to the AllScale Resilience and Fault Tolerance component;
  o AllScale Resilience and Fault Tolerance Management: clarified the main focus of the Resilience Manager to be on the node failures, which will be made automatic without or with minimal involvement of the application developers; redefined requirements on the Resilience Manager as well as updated the requirements towards the other components, in particular the AllScale Runtime System, and the AllScale pilots.
• the pilot applications
  o iPIC3D: updated the benchmarks according to the modified requirements in Section 3.1;
D2.2 – Requirement Specifications and Reports on External Technological Developments (b)

- FINE/Open: mainly updated the benchmarks following the modified requirements in Section 3.1; minor changes to the Appendix;
- AMDADOS: updated the requirements on the pilot, clarifying the numerical scheme.
1 Introduction

This deliverable is composed of two parts: a report on external technological developments and requirements specifications. The first part provides an overview of the external state-of-the-art technological developments related to the AllScale project. The second part specifies the requirements for the AllScale Environment. For that purpose, the analysis of the AllScale Environment as well as the three pilot applications in relation with the AllScale Environment have been included. From that analysis a list of requirements has been generated for each component within the AllScale Environment and each of the pilot applications.

The document is structured as follows:

Section 2 manifests the state-of-the-art external Exascale technologies; Section 3 presents the AllScale Environment requirements for each of its components as well as the corresponding requirements on the pilot applications; Section 4 describes the three pilot applications and both their requirements for the AllScale Environment and their benchmarks; Finally, we discuss the AllScale Objectives in Section 5 and draw conclusions in Section 6.
### 2 Exascale Technologies: External Technological Developments

#### 2.1 Hardware Development

The goal is to reach Exascale Computing by the year 2020. In order to achieve that goal, the underlying computing hardware needs to undergo various transformations to accommodate the greatest challenge to fit into a power envelope of 20 MW per exaflop as stated by the DOE and DARPA. The trajectory of the current development leads us to believe that potential Exascale systems will be equipped with deep and complex memory hierarchies, billion way global parallelism, and concurrency in the order of a thousand cores on a single node (Shalf, Dosanjh and Morrison 2010). This will lead to hardware designs featuring heterogeneous cores with multiple memory levels and complex network topologies to account for I/O and scalable communication between compute nodes.

Today, three competing designs towards Exascale are emerging. The first is focusing on taking advantage of the development of low-power CPUs for embedded devices and retrofit them to be usable for scientific computing, this is backed by 64-bit ARM processors with the help of embedded GPUs in the Mont-Blanc project (Rajovic, et al. 2013). Secondly, IBM and NVIDIA are working on building a successor to the BlueGene/Q based supercomputers on the basis of highly power efficient POWER CPUs and NVIDIA based accelerators (O. R. Laboratory 2016). This effort is based on the experience from IBM building the very energy efficient BlueGene/Q and NVIDIA's currently leading accelerator technology based on their Tesla product line forming a powerful heterogeneous computing platform. Last but not least, Intel is focusing on their Many-Core Product, the Xeon Phi. The generation of Xeon Phi, Knights Landing, is resembling a technology that is not build on heterogeneous compute environments, but massive on node parallelism with less powerful CPUs (Trader 2016).

The biggest driver for hardware development in recent years has been the rise of neural networks. In contrast to the more traditional computational science applications, neural networks only require half precision floating point operations. As accelerators will be equipped with half-precision functional units, the overall scientific community can benefit due to the development of mixed-precision algorithms to reduce memory transfer, which is considered to be the main factor of energy consumption in modern systems.

#### 2.2 Application Programming Interface (API)

A promising approach currently crystallizing in the initial community of Exascale users is the use of C++ template libraries which abstract the details of the underlying hardware and even software infrastructure from application experts, who are required to expose as much fine-grained parallelism as possible in their kernel code. Implementations of this principle include Kokkos (Edwards and Trott 2013) from Sandia National Laboratories and the RAJA portability layer (Hornung and Keasler 2014), which is currently in development at Lawrence Livermore National Laboratory. A lower level of abstraction, which is based on
similar technology, is provided by the OCCA library (Medina and St-Cyr 2014). All of these systems are based on the C++ template abstraction of parallelism, and most are designed to allow domain scientists to focus on expressing the parallelism inherent in their domain code without regard for low-level technical details. As such, their increasing prevalence supports our design choices for AllScale, which features a high-level templated AllScale User API.

More conventional strategies for allowing programmers to specify parallelism are parallel programming languages and language extensions. Such languages and extensions include: Cilk (R.D. Blumofe 1995) and OpenMP (OpenMP Specifications 2016) for shared memory parallelism; PGAS (PGAS 2016) languages such as UPC and X10 for distributed memory; OpenACC (OpenACC - Directives for Accelerators 2016), CUDA (NVIDIA 2016), and OpenCL (Khronos Group 2016) for accelerator computing.

In addition to language extensions, an ongoing effort to develop new programming languages is emerging, aiming at better support for parallel programming. Some parallel programming languages like OpenCL or Intel Cilk Plus are focusing on node level parallelism. While OpenCL focuses on abstracting hardware differences for all kinds of parallelism, Intel Cilk Plus supports a fork-join style of parallelism. In addition, there are programming languages that explicitly support distributed computing, like UPC (UPC Consortium 2005) or Fortress (Oracle 2016), but lack support for intra-node level parallelism. Current research, however, is developing support for both, inter and intra-node level parallelism based on a partitioned global address space (PGAS (PGAS 2016)). The most prominent examples are Chapel and X10, which represent PGAS languages. HCMP (Chatterjee, et al. 2013) shows similarities with the AllScale programming model by offering interfaces for asynchronous distributed computing, either based on distributed data driven futures or explicit message passing in an MPI compatible manner.

The utilization of nested recursive parallelism for conducting concurrent operations was previously leveraged by NESL (Blelloch 1995). However, its sample algorithms and implementation were written in the 1990s with dependencies on contemporary technologies, making them difficult to apply or evaluate on modern platforms. Other languages, including X10 (Charles, et al. 2005), Habanero-C MPI (Chatterjee, et al. 2013), Charm++ (Kale 1993) and Chapel (Chamberlain, Callahan and Zima 2007) try to extend the task based, nested recursive programming model of Cilk to large scale, distributed systems. However, a major disadvantage associated with these approaches is the dependency on a new programming language and the resulting coding and porting effort for existing libraries and applications.

As the space of viable high-performance parallel APIs grows rapidly, it is becoming increasingly challenging to maintain a holistic view of the feature set each of them offers. To alleviate this issue, we have created a classification matrix for task-parallel APIs that extends previous work in the field (Kasim 2008) to include features relevant for modern HPC systems, such as heterogeneity and resilience management. We will now define each of these
D2.2 – Requirement Specifications and Reports on External Technological Developments (b)

characteristics and their available options for categorization. Note that explicit (e) support generally refers to features that are supported, but require extra effort or implementation by the application developer, while implicit (i) support means that the toolchain manages the feature automatically given a default representation of the program in the API.

- **Technology Readiness**: The readiness of the API and its implementations according to the European Commission definition.

- **Distributed Memory**: Whether targeting distributed memory systems is supported. Options are no support, explicit support, or implicit support. Explicit refers to, for example, message passing between address spaces, while automatic data migration would be an example of implicit support.

- **Heterogeneity**: Indicates whether tasks can be executed on accelerators (e.g. GPUs). Again, explicit and implicit as well as no support are possible, where the former means that the application developer has to actively provision tasks to run on accelerators, using a distinct API.

- **Worker Management**: Whether the worker threads and/or processes need to be started and maintained by the user (explicit) or are provided automatically by the environment (implicit).

- **Task Partitioning**: This feature indicates whether each task is atomic - can, thus, only be scheduled as a single unit - or can be subdivided/split.

- **Work Mapping**: Describes the way tasks are mapped to the existing hardware resources. Possibilities include explicit work mapping, implicit work mapping (e.g. stealing), or pattern-based work mapping.

- **Synchronization**: Whether tasks are synchronized in an implicit fashion, e.g. by regions or the function scope, or explicitly by the application developer.

- **Resilience Management**: Describes whether the API has support for task resilience management, e.g. fine-grained checkpointing and restart.

---

Figure 1: Feature matrix for task-parallel APIs.
2.2 – Requirement Specifications and Reports on External Technological Developments (b)

- **Communication Model**: Either shared memory (smem), message passing (msg), or Partitioned Global Address Space (pgas).
- **Result Handling**: How the tasking model supports handling the results of task computation: implicit via write-back to existing data, or explicitly provided task result types (which might, for example, be accessed as futures).
- **Graph Structure**: The type of task graph dependency structure supported by the given API: a tree structure, an acyclic graph (dag) or an arbitrary graph.
- **Task Cancellation**: Whether the tasking model supports cancellation of tasks: no cancellation support; cancellation is supported either cooperatively (only at task scheduling points) or preemptively.
- **Implementation Type**: How the API is implemented and addressed from a program. A tasking API can be provided either as a library, a language extension, e.g. pragmas, or an entire language with task integration.

2.3 Compilers

Effective use of the variety and complexity presented by the hardware platforms currently envisioned for Exascale systems will require greater tool support, both to improve productivity of program development as well as to enable performance portability across current and towards future architectures (Amarasinghe, et al. 2011).

As described in Section 2.2, many library-based approaches to HPC are emerging or already in common use. They have the advantage – compared to languages and language extensions – of working with established toolchains, enabling easier development and debugging. However, this advantage comes at the cost of tool support, as compilers are inherently incapable of analyzing and comprehending parallelism expressed in opaque library calls.

To overcome this limitation, the AllScale Compiler will be extended to comprehend the core set of Allscale API primitives, enabling it to perform semantics-aware compilation and analysis on programs using that API to express parallelism. A similar approach, but on a much smaller scale and focused entirely on shared memory parallelism, has been demonstrated (Thoman, Moosbrugger and Fahringer 2015) to be highly beneficial for the optimization of fine-grained task parallelism.

Liao et al. (Liao, et al. 2010) also investigated semantics-aware compilation in parallel computing. Their goal was to improve the applicability of compiler auto-parallelization by taking into account STL container semantics in the ROSE compiler framework. The ROSE source-to-source compiler infrastructure (Quinlan and Liao 2011) was also leveraged for research in multiple Exascale-related topics, such as fault tolerance (Lidman, et al. 2012) and hardware/software co-design (Dakshinamurthy and Dechev 2012).

2.4 Runtime Systems

The latest research in the field of parallel runtime systems for Exascale systems points to the efficient utilization of intra-node parallelism with the ability to
exploit the underlying communication infrastructure using fine grain task based approaches to deal with concurrency.

In recent years, runtime systems targeting lightweight tasks are more and more commonplace. HPX, the runtime system used within the AllScale Environment, is no exception to this trend. Most discussed solutions in section 2.2 represent task based parallel programming models which implement their own runtime systems using lightweight task based approaches.

While those APIs expose parallelism to the user in different ways, they all seem to converge on a continuation style programming model. Some of them use a futures-based programming model, while others use data-flow processing of dependent tasks with an implicit or explicit Data Acyclic Graph (DAG) representation of the control flow. While the majority of the task based programming models focus on node level parallelism, HPX presents a solution for homogeneous execution of remote and local operations. When looking at library solutions for task-based programming, we are mainly looking at C/C++ for the scope of AllScale. Examples of pure C solutions are StarPU (Augonnet 2011) and Qthreads (S. N. Laboratory 2016). They both provide interfaces for starting and waiting on lightweight tasks as well as creating task dependencies. While Qthreads provides suspendable user level threads, StarPU is built upon Codelets which, by definition, run to completion without suspension. Each of these strategies, Codelets and User-Level-Threads, has its advantages. HPX is a C++ library and tries to stay fully inside the C++ memory model and its execution scheme. For this reason, it follows the same route as Qthreads to allow for more flexibility and for easier support for synchronization mechanisms. What these libraries have in common is that they provide a high performance solution to task-based parallelism with all requirements one could think about for dealing with intra-node level parallelism. What they clearly lack is a uniform API and a solution for dealing with distributed memory parallel computing. This is one of the key advantages HPX has over these solutions: A programming API that conforms to the C++11 and the upcoming C++14 standards (The C++ Standards Committee 2011) (The C++ Standards Committee 2014) but is augmented and extended to support remote operations.

**2.5 Scheduling in Many-Task Runtime Systems**

Despite task-based parallelism is a conceptually simple paradigm, the increasing complexity of HPC systems makes the development of portable applications that efficiently map their tasks to the available resources more challenging than it was before.

Scheduling on shared-memory machines has been intensively studied. Cilk (R.D. Blumofe 1995) was one of the first to implement the task-based paradigm within its runtime for shared-memory platforms. Its runtime employs a randomized work-stealing scheduling algorithm which is provably efficient in terms of time, space, and communication. Work-stealing algorithms have been shown to efficiently load balance fine-grain task-parallel programs on multicore machines.
D2.2 – Requirement Specifications and Reports on External Technological Developments (b)

OpenMP (OpenMP Specifications 2016) is the de-facto standard for shared-memory parallel programming and has been the base for implementing different scheduling algorithms. OpenMP 3.0 standard added task parallelism to its API and supports a naive breadth-first scheduling as well as a work-stealing scheduling algorithm (Alejandro Duran 2008). A research work in (Marie Durand 2013) discusses an adaptive scheduling algorithm for OpenMP loops that dynamically changes the granularity of the work while taking the underlying system state on NUMA architectures.

OmpSs (Duran A 2011) is an extension of StarSs (Labarta 2010) programming model and provides breadth-first and work-first scheduling algorithms through its Nanos++ runtime. The runtime also provides a socket-aware scheduler, where top level tasks with depth one are assigned to a set of NUMA nodes by the user before task creation, while nested tasks created by them will run on the same node as their parent. The other type of scheduling policy provided by Nanos++ is called a versioning scheduling for multi-GPU heterogeneous systems (J. Planas 2013), where the scheduler can handle multiple implementations of the same task. By means of automatic profiling the scheduler selects the most suitable implementation and executes it on a thread that minimizes execution time of the task, taking the task’s data size and each worker’s task list into account. ClusterSs (Enric Tejedor 2011) extends the StarSs programming model to distributed-memory platforms. It is based on asynchronous PGAS model and works in a master-slave mode. The master node manages task generation and the slave nodes execute the generated tasks. Initially, the data created by the sequential part of the application on the master node. In addition, the runtime supports data-allocating tasks that allocate data on the same node as the worker. The main node keeps track of the data locations. The runtime relies on this information to realize locality-aware scheduling. The master-slave design of the runtime may result in scalability issues due to bottleneck on the master node.

StarPU (Augonnet 2011) provides a modular scheduling engine for heterogeneous systems. Its scheduling engine comes with a rich set of scheduling algorithms such as eager, heft (heterogeneous earliest finish time) (Haluk Topcuoglu 2012), work-stealing, data-aware, locality-aware, and energy-aware scheduling policies. Most of these policies are based on performance models of the processing units and tasks. The models can either be provided by the application developer or built during runtime from historical profiling data. In addition, StarPU employs regression-based performance models for algorithms that have different type and size of inputs. StarPU-MPI (Emmanuel Agullo 2016) is an extension to the StarPU runtime to distributed memory platforms. The main idea is that the application is responsible for initial data distribution over the MPI nodes, and the sequence of tasks is identically submitted by the application on all MPI nodes. Tasks run on the node that owns data they write to. Within each node the StarPU runtime schedules the local tasks. Any inter-node data dependencies are handled by transparently posting non-blocking MPI send and receive calls.

Legion (Bauer, M. E., 2014) runtime provides a mapper interface to application developers to implement scheduling policies for distributed-memory systems.
Also, it comes with a default scheduling policy that checks the type of processor for which a given task has implementation and then if the fastest implementation is the local processor the task runs locally, otherwise it is sent to the closest fastest processor.

HPX runtime implements several scheduling policies, including priority-based FIFO scheduling where each OS thread has its own queue of tasks and NUMA-aware work-stealing policies. AllScale scheduler will be based on the HPX scheduler and extend it by taking multiple objectives, such as time, energy, and resource utilization into account. It will dynamically steer the application towards fulfilling the trade-offs among the objectives by utilizing multiversioning feature of its runtime, dynamically load balancing, and incorporating objectives imposed by the resiliency management component.

### 2.6 Monitoring Frameworks

Exascale computing requires a change in the process of performance data collection and analysis. The high concurrency and dynamic behavior of upcoming HPC systems demands a transition from current static post-mortem analysis approaches to real-time performance data collection, filtering and introspection.

Some performance analysis frameworks have started to provide online introspection capabilities. For instance, Score-P (Score-P. 2014) permits their clients to use its monitoring infrastructure remotely via a TCP/IP connection. Tools such as the PIA (Aguilar 2013) module in IPM (Integrated Monitoring Tool (IPM) 2016) and TAUg (K. A. Huck 2006) provide performance introspection feedback in real time. Other tools, however, implement techniques for online analysis of performance data while it is generated. Autopilot (Ribler 1998) uses performance sensors together with decision procedures to automatically tune applications. In a similar manner, Periscope (Shajulin Benedict 2009) utilizes different monitoring agents in conjunction with a set of rules and properties to detect performance problems automatically while the application is running.

Various runtime systems use performance data in order to improve their scheduling mechanisms. For instance, Harmony (Yalamanchili 2008) and StarPU (Augonnet 2011) (Cédric Augonnet 2009) build performance models based on historical data to help the scheduler with its decision making process. OmpSs has an extension that uses introspection data to choose the most appropriate code version among several implementations of the same task (J. Planas 2013). Task mappers in Legion can request performance information to the runtime in order to evaluate the performance implications of their task mapping decisions (M. E. Bauer 2014). In a similar manner to AllScale, the XPRESS project develops a highly scalable and reactive monitoring component (APEX) (K. S. Huck 2013) to provide introspection information to the HPX system while applications run.

### 2.7 Fault Tolerance and Resilience Approaches

The topics of fault tolerance and resilience have been approached in a number of ways by the scientific community in the last decades.
In the HPC community, fault tolerance is an important topic, particularly so in the context of Exascale computing (Schroeder and Gibson 2010). Mean time between failures reduces linearly with the number of cores. Models of Exascale applications demonstrate the need for fault tolerant mechanisms when utilizing tens of thousands of cores (Ferreira, et al. 2011).

A classification of faults in itself is a complex topic. In agreement with (Hoemmen and Heroux 2011), we generally speak of hard vs soft faults. Soft faults can be further subdivided into transient, sticky, or persistent.

The traditionally well studied approach to fault tolerance are rollback recovery mechanisms. A good overview of rollback recovery mechanisms in message-passing systems is (Elnozahy, et al. 2002). The authors classify them into log-based and checkpoint-based (C/R) recovery. The checkpoint-restart based recovery is possibly the best known fault tolerance approach. There is a huge variety among C/R techniques. In recent years, one of the most efficient and scalable C/R techniques uses in-memory checkpointing at peer nodes (Zheng 2012). This technique has been used extensively in recent years, even for supercomputer runs (Gamell, et al. 2014). Also, multilevel checkpointing is an increasingly popular approach to checkpointing (Moody, et al. 2010).

Instead of general-purpose rollback recovery mechanisms, more efficient mechanisms through application-specific optimizations have been studied in recent years. A dominant approach to fault-tolerance in the last few years is algorithm-based fault tolerance (ABFT). This approach has its beginnings in work on detecting and correcting errors in matrix-matrix multiplication (Huang 1984), where both detection and correction of errors allow the algorithm to loop forward without rolling back. Subsequently, this approach has been applied to parallel implementations of matrix-matrix multiplication (Bosilca, et al. 2009), or implementations of iterative solvers (Fasi, et al. 2015). Naturally, approaches combining algorithm-based error detection with the possibility to restart from the most recent checkpoint can also be used (Z. Chen 2013).

Another direction of fault tolerance altogether deals with faults by replication of resources (Ferreira, et al. 2011). While the hardware requirements are a doubling of hardware resources (full replication), this approach does have a number of advantages, e.g. significantly reduced probability of a fault for a node and its replica. Earlier work on MPI for volatile nodes (Bouteiller, et al. 2003) includes an implementation that enables restart processes on volatile nodes without stopping the parallel application.

Process migration has not been a popular approach in the MPI community (with an early paper (Stellner 1996) a rare exception). Instead, the majority of fault tolerance in case of node failure focuses on a checkpoint/restart mechanism on the physical node that failed. Still, process migration could be used both in a proactive and a reactive fashion in general. An example of proactive usage is the migration due to load imbalance, or due to an indication that a hard fault will
soon occur. An example of reactive usage is the recovery from hard faults (e.g. node failure) via process migration.

In the distributed systems community, examples of fault tolerance include self-stabilization and self-healing. Self-stabilization has been proposed in a seminal paper by Dijkstra (Dijkstra 1974). Self-stabilizing algorithms are defined by the two properties of convergence and closure. As an example for self-stabilization, a system may revert back to some good state from any arbitrary state in finite time. This may be accomplished, for instance, by recovering state by looking at other states in the neighborhood or by forcing a part of the network to re-compute from a previously good state. Recent work (Sao and Vuduc 2013) proposes an "imperfect" self-stabilizing step for the Conjugate Gradients method. The appeal of the technique is in the low overhead, but CG then loses any convergence guarantees.

In self-healing algorithms, examples can include severe failures like node failures. When an individual node fails, the nodes in its neighborhood take local corrective action so that the topology or computation is restored to some good state in quick time. Also, so called self-healing algorithms based on iterative solvers have been proposed relying on replication. (Chen and Dongarra 2009) present a framework for building self-healing high-performance numerical computing applications so that they can recover from a hard fault using a checkpoint. Their framework is based on efficient strategies of diskless checkpointing.
3 AllScale Environment

Figure 2 outlines the overall architecture of the AllScale Environment, comprising three major components:

- The AllScale API at the top of the software stack offering a unified parallel programming interface
- The AllScale Compiler utilizing static analysis to generate and optimize codes for various target architectures
- The AllScale Runtime System managing and auto-tuning the execution of parallel programs for multiple objectives

Apart from the above-mentioned core components, the AllScale Environment also includes the AllScale Scheduler, the AllScale Monitoring Framework as well as the AllScale Resilience and Fault Tolerance Management. We provide below a detailed specification of each of these components.

3.1 AllScale API

User applications will be implemented based on a set of generic parallel APIs offered by the project -- the AllScale API. This component is subdivided into two layers: the AllScale User-Level API and the AllScale Core API. The AllScale Core API layer covers a small, concise set of essential parallel primitives: constructs for expressing nested recursive parallelism, synchronization, and distributable data structures. While concise and expressive, they are not supposed to be directly utilized by application developers due to their complex nature. Their main purpose is to provide a compact interface for actual implementations, in particular the one realized by the AllScale Compiler. To support application
developers, an AllScale User API layer is introduced and maintained by expert developers. It maps common parallel patterns to the Core-API primitives. The AllScale User API supports stencil operations, reduction and parallel loop operations -- thus, support for the vast majority of performance demanding, parallel codes. Also, it supports a comprehensive set of data structures (e.g. arrays, trees, maps, etc.) as required for the development of a large variety of applications based on the underlying recursive primitives.

3.1.1 Requirements on the AllScale APIs
The User-Level API has to provide primitives supporting the pilot applications as well as future variations of those such as

• Parallel Control Flow Primitives:
  o Basic operations covering parallel loops, individual tasks and futures, reductions;
  o More sophisticated patterns: regular grid stencils API, adaptive grid stencil API.
• Data Structure Constructors:
  o Multi-dimensional grids, static and dynamically sized;
  o A data structure to model unstructured multi-level grids supporting multiple node and edge types as well as recursive graph decomposition.
• Synchronization primitives:
  o Decomposable hierarchical futures (aka treetures).
• IO operations:
  o To read/write structured input/output data to the file system.
• Instrumentation:
  o Primitives to instrument the user code for obtaining profiling data are required.

To provide an interface to be manageable by the AllScale Compiler, all AllScale User-Level API interfaces have to be exclusively mapped to the AllScale Core API components as follows

• A parallel recursive operator providing the means to express a parallel nested recursive computation offering means to express base-case conditions, multiple base-case and step-case implementations. Synchronization is to be supported utilizing treetures;
• Treetures for synchronization between tasks;
• Basic grids and meshes for data distribution.

In addition, external library integration interfaces have to be provided by the AllScale Core API.

3.1.2 Requirements Towards the Other Components

3.1.2.1 The AllScale Compiler
To support the AllScale Core API, the AllScale Compiler has to be capable of intercepting the AllScale Core API and implant AllScale functionality. To that end, the AllScale Compiler has to support the C++14 features utilized by the implementation of the AllScale API.
3.1.2.2 The AllScale Runtime System

To facilitate the effective development and tuning of the AllScale API towards high-performance on the intra- and inter-node level, the Runtime System has to provide competitive performance within

- **Intra-node** level (comparable with e.g. OpenMP, Cilk);
- **Inter-node** level (extrapolation of intra-node scalability).

3.1.2.3 The Pilot Applications

The AllScale API will be offered as a C++ template library making extensive use of C++11/14 features. Thus the following requirements are imposed:

- For its utilization, a C++14 compatible compiler is required (e.g. GCC 5.2.1);
- The pilot applications that use the API have to be written in C++ as well.

To enable the API, and thus the AllScale infrastructure, to effectively manage the parallel execution of the resulting programs, full control over parallelization decisions is required. Consequently,

- pilot applications are required to port parallel operations to the AllScale User API;
- no other parallel APIs, including e.g. MPI, OpenMP, OpenCL, CUDA, Cilk, pthreads, HPX primitives, C++11 threading facilities, POSIX APIs for process control or others may be used within the application code;
- no external library may be integrated, which uses such third-party parallel constructs.

Furthermore, to enable the AllScale infrastructure to manage IO operations a corresponding API will be part of the AllScale User API. Thus,

- the pilot applications are required to utilize AllScale IO operations exclusively for several reasons:
  - The evaluation order of operations can differ from the evaluation order in the conventional implementation:
    - Due to the recursive decomposition of the space-time some output results of later time steps may be already present while data from other locations is not yet present. The IO must therefore not impose any particular output order and still be efficient;
    - The runtime scheduler influences the execution order of tasks for conducting load balancing;
    - For overlapping the computation and IO continuously, a full-stop of the simulation to conduct IO operations is not intended and shall not be supported.
  - The dynamic scheduling may move tasks between different processes and those resources like file-handlers cannot be migrated unless supported by the system; we aim to avoid this.
  - For providing rollbacks in the system in case of an error, data that has been outputted by some partially processed task has to be discarded; to be able to do so, IO has to be under the control of the corresponding management system.

Finally, to facilitate efficient tuning of the code within an intra- and inter-node level, the pilot application groups are required to provide realistic benchmarks for
D2.2 – Requirement Specifications and Reports on External Technological Developments (b)

- test scale problems to be processed by a single CPU core (e.g. shared memory SMP, 1 core, ~1sec sequential execution time);
- tiny scale problems to be processed by a single desktop node (e.g. shared memory SMP, 8 cores, ~10sec parallel execution time);
- small scale problems to be processed by a single server node (e.g. shared memory NUMA, 4x8 cores, ~1min parallel execution time);
- medium size problems for small distribute memory systems (e.g. 4 server nodes, ~10min parallel execution time);
- large size problems for large distributed memory systems (e.g. 100-1000 nodes, ~1h parallel execution time);
- real world problem for benchmarking on Exascale-like system.

All those benchmarks have to consist of a main-program provided in C++, an input set, including potential additional execution parameters like number of time steps or physical parameters, and means to verify the correctness of the obtained results.

3.2 AllScale Compiler

Codes implemented utilizing the AllScale API can be compiled by standard C++ tools and executed on parallel, shared memory machines. This mode is intended for use during the development and debugging phases of an application development project. However, for obtaining an extreme scale high-performance version that benefits from all the novel services offered by the AllScale Environment, the same code base will have to be compiled by the API-aware AllScale Compiler. Unlike standard C++ compilers, the AllScale Compiler will be aware of the interpretation of the parallel primitives offered by the AllScale Core API layer and will restructure the application by introducing additional versions of encountered parallel code fragments. Each version will target a different architecture and/or represent a different trade-off between multiple optimization objectives, e.g. more parallelism versus less overhead. Together with information describing the specific traits of the versions -- such as hints on degree of parallelism -- as well as details regarding their (data) dependencies, the resulting set of implementations will be forwarded to the unified AllScale Runtime System.

The implementation of the AllScale Compiler will be based on the existing Insieme source-to-source compiler infrastructure and its high-level intermediate representation (Jordan, et al. 2013).

3.2.1 Requirements on the AllScale Compiler

The AllScale Compiler is required to compile applications that rely upon the AllScale API to a C++ program utilizing the AllScale Runtime System API, enabling the Runtime System to effectively control and steer the parallel execution of the resulting program. As such, the compiler has to support:

- the C++14 features utilized by the AllScale API for providing parallel primitives.
- the capability of intercepting and interpreting the AllScale Core API primitives.
- analysis to deduce static information regarding data requirements from the processed input codes.
D2.2 – Requirement Specifications and Reports on External Technological Developments (b)

- transformation of the analysis results into a format to be utilizable by the AllScale Runtime System to distribute:
  - parallel operations;
  - data.
- generation of code variants enabling the AllScale Runtime System to move computations:
  - between distributed memory nodes;
  - to accelerators.
- the generated code should include code-line annotations, enabling the generated code to be traced back to the original input code;
- providing code versions of the processed tasks enabling the creation and restoration of local checkpoints;
- providing feedback to the user regarding the success/failure of analysis and code generation steps.
Furthermore, the AllScale Compiler has to provide interfaces for the resilience and instrumentation groups to integrate required constructs into the generated target code.

3.2.2 Requirements Towards the Other Components

3.2.2.1 The AllScale Runtime System
The interface to the AllScale Runtime System is the AllScale Runtime System API. This is required to provide primitives that can be utilized by the AllScale Compiler to express
- Parallel control flows;
- Synchronization operations;
- Data requirements and dependencies;
- IO operations;
- Operations and data structures to be processed / placed on accelerator hardware;
- Meta information to be provided to the AllScale optimizer to aid the decision making processes.

3.2.2.2 The AllScale Resilience and Instrumentation
Depending on modifications to be conducted to the code to be generated by the AllScale Compiler, corresponding primitives have to be offered as a part of the AllScale Runtime System API.

3.2.2.3 The Pilot Applications
Since it is the compiler's responsibility to extract data dependency information from the input codes provided by the application groups, some restrictions have to be imposed on those codes. Among those are strict constraints including:
- All sources need to be available;
- No external third party library unless explicitly integrated (no black boxes -- semantic information needs to be provided);
- Clang 3.6 compatible code;
- No labels and gotos (not supported by the AllScale Compiler);
- No inline assembler;
- No intrinsics.
D2.2 – Requirement Specifications and Reports on External Technological Developments (b)

- No C++ exceptions

Furthermore, several constraints that would significantly help in increasing the accuracy of the conducted program analysis and, therefore, in increasing the quality and flexibility of the resulting code:

- No use of C++ exception like try/throw/catch in user codes;
- Focus on POD (plain old data) types, meaning user codes should not introduce types that require side effects or complex memory operations on copy/move and assignment;
- Utilization of standard containers and the AllScale data structures.

3.3 AllScale Runtime System

The AllScale Runtime System will be at the core of the AllScale Environment. Due to the use of the unified AllScale Core API, an entire application can be executed in parallel without having to deal with different levels of parallelism within the program, while still maintaining high performance with the help of the AllScale Runtime System. The runtime system will employ multi-objective optimization to achieve a desired trade-off among the considered tuning objectives. The Runtime System can, for this purpose, rely on dynamically updated knowledge provided by online monitoring services capturing the state of the target system as well as the managed program execution.

The AllScale Runtime System will be based on HPX (Kaiser, et al. 2014). Profiling support will be based on a solution that builds on top of the HPX performance counters interface.

3.3.1 Requirements on the AllScale Runtime System

The Runtime System is in charge of executing the parallel program as delivered by the AllScale Compiler. The Runtime System has to be portable in order to run on various different architectures as well as to support accelerators in the form of GPUs. Thus, the Runtime System has to be hardened against hardware failures. The Runtime System has to provide an interface for the AllScale Scheduler to conduct its operations, abstracting the underlying implementation details. In addition, the Runtime System has to provide services that allow efficient and scalable load balancing, performance monitoring, and support for fault tolerance.

3.3.2 Requirements Towards the Other Components

3.3.2.1 The AllScale Compiler

In order for the Runtime System to be able to execute the parallel program efficiently and correctly, the Compiler needs to provide additional information:

- Dependencies between different tasks: Data and Control Flow;
- Annotate the source code with serialization capabilities to allow the runtime to migrate and send code and data to remote processes;
- Generate code thus the Runtime System is able to create distributed data structures;
- Meta information that will help to assess performance that will be passed on to the optimizer.
3.3.2.2 The AllScale Monitoring Framework
The AllScale Monitoring Framework should ideally be built around the infrastructure that is already provided within the HPX runtime system. The usage of other parallel programming APIs like MPI have to be avoided since they are not interoperable with HPX. The monitoring Framework should enable online and post-mortem performance analysis and facilitate debugging.

3.3.2.3 The AllScale Resilience and Fault Tolerance Management
No further requirements are necessary for this component.

3.3.2.4 The Pilot Applications
Since all pilot applications will be compiled with the AllScale Compiler, there are no special requirements for them.

3.4 AllScale Scheduler
The AllScale Scheduler will be a central component in the AllScale Runtime System. The existing HPX intra-node scheduler will be adapted to support a common interface for the local scheduling within a node and inter-node task scheduling. The Scheduler is responsible to decide where and how to process the tasks.

The AllScale Scheduling component will provide abstract and extensible interface to realize different scheduling policies, including data-aware, energy-efficient, and multi-objective scheduling policies.

3.4.1 Requirements on the Allscale Scheduler
The execution of the compiled application, along with the execution of tasks, can trigger the spawn of new tasks. A specific command will be designed in order to provide the necessary information to the scheduler regarding a task to queue. Using this information, the Scheduler will take decision regarding granularity and locality of the task to execute.

The Scheduler component will dynamically auto-tune the program for multiple objectives to achieve a desired trade-off among the considered tuning objectives. A user selected multi-objective criteria should be provided to the Scheduler by the AllScale Runtime System. The set of objective trade-offs can be defined as follows:

- Energy_only
- Resource_only
- Time_only
- Energy_resource
- Energy_time
- Resource_time
- Energy_resource_time

The AllScale Resilience and Fault Tolerance Management may require to flush some task queues due to failure detection. The Scheduler component will provide the appropriate API in this manner.
3.4.2 Requirements Towards the Other Components

3.4.2.1 The AllScale Compiler
The AllScale Compiler should provide the required information related to each task to execute such as which data are required by a task, what granularity can be achieved and code variants and their targeted architecture exists for this task.

3.4.2.2 The AllScale Runtime System
The AllScale Runtime System should provide the necessary means for data and task movement/migration across different localities of the system, including accessing the different code variants, and the architecture of the different localities, such as processor type and existing accelerators.

The Runtime System Data Manager should provide the necessary APIs to move data to a different locality for load balancing purposes and track data.

3.4.2.3 The AllScale Monitoring Framework
The Scheduler needs enough information regarding the available hardware infrastructure in order to effectively steer the application. This includes static information such as node structure, network topology, memory hierarchy, GPGPU information.

In addition, the scheduler needs to get dynamic information, including current load of hardware components and their power consumption. The expected execution time and expected energy consumption of the tasks should be provided by the monitoring component as well.

3.4.2.4 The AllScale Resilience and Fault Tolerance Management
No further requirements for this component.

3.4.2.5 The Pilot Applications
No special requirement toward the pilot applications.

3.5 AllScale Monitoring Framework
For an effective scheduling of applications running on the AllScale Runtime System, information regarding the available hardware infrastructure and its utilization is required. In addition to static data covering details on the system architecture, including the node structure, network topology and memory hierarchy, dynamic information in real-time like the current load of hardware components, their power consumption or parameters characterizing their efficient utilization (e.g. cache misses) is of equal importance. Additionally, application-level runtime information, e.g. execution time of certain tasks defined by the compiler/user or length of task queues, constitutes vital input to the sophisticated scheduling policies targeted by AllScale. Furthermore, the monitoring component will also inform the resilience manager of node failures. The objective of the monitoring component within the AllScale Runtime System is to provide a framework for the collection, aggregation and exchange of the various types of information prescribed above.
3.5.1 Requirements on the AllScale Monitoring Framework
The AllScale Monitoring Framework will provide an abstract, extensible interface of performance sensors, functions conducting on-the-fly data aggregation operations to reduce the amount of maintained data, and a scalable communication infrastructure enabling system wide access to required data on-demand.

3.5.2 Requirements Towards the Other Components

3.5.2.1 The AllScale Compiler
In order to relate the collected measurement to specific program parts, the compiler needs to provide hooks where the monitoring system can take control and collect the performance data required, e.g. entry and exit points of functions or tasks.

3.5.2.2 The AllScale Runtime System
The monitoring component will be build on top of the HPX performance counters layer, and thus, it requires the HPX system to provide the proper performance raw measurements, i.e. information about the state of the runtime as well as the state of the hardware (hardware performance counters). In addition, the runtime system needs to provide static information about the underlying system architecture.

3.5.2.3 The AllScale Resilience and Fault Tolerance Management
No further requirements for this component.

3.5.2.4 The Pilot Applications
No special requirement toward the pilot applications.

3.6 AllScale Resilience and Fault Tolerance Management
The main focus of the Resilience Manager is on recovery from node failures. The recovery will follow via checkpoint-restart. However, due to the task-based nature of the AllScale architecture, task-based checkpoint-restart mechanisms will be developed, with particular challenges and opportunities.

Checkpointing of tasks and application data will take place. The nested recursion within the AllScale architecture will support performing such checkpointing on various granularity levels.

The focus on node failure recovery, and the nature of task-based checkpoint-restart strategies has important implications about how the Resilience Manager is embedded into the AllScale architecture. Most importantly, the Resilience Manager is aligned very closely with the AllScale Runtime System. The AllScale pilot applications may be involved in the objective of application-specific checkpoints, in case this process cannot be fully automated.

3.6.1 Requirements on the AllScale Resilience and Fault Tolerance Management
The requirements towards the Resilience Manager are:

- The ability to store at runtime closures and data checkpoints per AllScale task. This progress needs to be stored in a scalable fashion, preferably without global synchronization.
The ability to trigger recovery in the presence of hardware faults, resulting in continued application execution. This includes:

- The recovery of the “context” of failed tasks, including their dependencies and arguments.
- The recovery of the application data of failed tasks, i.e., the application checkpoints.

### 3.6.2 Requirements Towards the Other Components

#### 3.6.2.1 The AllScale Compiler

In a fully developed version, the AllScale Compiler will automate the checkpointing of task closures, and will automate the data checkpointing of tasks for simple data structures. By automate, we mean the AllScale Compiler will generate checkpointing routines that otherwise would be hand-written into the AllScale Runtime System. We postpone the final specification of these requirements to a later phase in the project, since we focus on the direct implementation of task-based checkpoint-restart within the AllScale Runtime System for the moment.

#### 3.6.2.2 The AllScale Runtime System and Scheduler

The most important requirements the Resilience Manager poses are to the AllScale Runtime System. The runtime needs to provide:

- "get" methods for the task closure and the application data associated to a task
- (de)serialize methods of the above closure/data
- send/receive calls for this stream over the network (for in-memory checkpointing)
- cancel or reschedule calls for tasks as part of the recovery strategy
- calls querying the dependency of 2 tasks (returning True or False) for advanced rollback strategies

#### 3.6.2.3 The AllScale Monitoring Framework

The AllScale Monitoring Framework should provide online monitoring of the online/offline status of hardware components to support the resilience module in triggering recovery strategies. To this end, a detection protocol for online/offline status of AllScale processes is essential. While the main subscriber of this protocol is the Resilience Manager, this protocol is part of the Monitoring Framework.

#### 3.6.2.4 The Pilot Applications

Requirements on the applications may be made in case the AllScale Runtime System is unable to automate local or global checkpointing. While most local checkpoints will be automated, we believe deducing globally consistent checkpoints may require some application developer input.
4 Pilot Applications

AllScale is validated by applications from fluid dynamics as well as environmental hazard and space weather simulations provided by SME (Small and Medium-sized Enterprises), industry and scientific partners. These applications are described in the following subsections.

4.1 iPIC3D: Implicit Particle-in-Cell Code for Space Weather Applications

4.1.1 Introduction

Space weather is the study of the processes originating in the sun and propagating through the solar system, with effects on people and technology in space and on the earth, ranging from auroras in the polar regions to electromagnetic disturbances causing disruptive currents in infrastructure such as power and communication lines.

KTH, together with KU Leuven, has implemented the massively parallel Particle-in-Cell code, iPIC3D as a C++ program using MPI and OpenMP (S. Markidis 2010) to study space weather events. The iPIC3D code simulates the interaction between solar wind and the Earth magnetic field. The magnetosphere is a large system with many complex physical processes, requiring realistic domain sizes and billions of computational particles. In the PIC model, plasma particles from the solar wind are mimicked by computational particles. At each computational cycle, the velocity and location of each particle are updated, the current and charge density are interpolated to the mesh grid and Maxwell’s equations are solved.

4.1.2 Scientific / Commercial Impact / Stakeholders

The use case for the AllScale project, plasma particles interacting with the Earth’s magnetic field, is depicted in Figure 3. Some highly energetic particles are trapped while others escape the confinement of the magnetic field. This leads to large load imbalances since particles concentrate close to the Earth (violet cloud in the figure) while few particles are located in other regions of space (almost

Figure 3: Simulation with iPIC3D of plasma particles interacting with Earth magnetic field lines (black lines)
invisible particle outside the violet cloud in the figure). Within AllScale, we will carry out the PIC simulation of magnetosphere formation enabling new science and new discoveries in the field of interaction between solar wind and planetary magnetic field.

The iPIC3D code is used in production by several groups in European Universities, such as KU Leuven and the University of Pisa, all of which will benefit from the AllScale results.

4.1.3 Exascale Potential

The computational cost of iPIC3D scales linearly with the number of particles for resolving the dynamics of the particles and as Nglog(Ng), where Ng is the number of cells in the domain, for solving Maxwell’s equations. An analysis of the need of Exascale for iPIC3D (G. Lapenta 2013) concludes that a first simulation of the magnetosphere with realistic parameters requires 6.26x10^{10} processes. The proposed realistic AllScale simulation of Earth’s radiation belts with iPIC3D would require a grid with 5,000x5,000x5,000 cells. Particle-wave interaction can be modeled accurately only if at least 10,000 particles per cell are used. For this reason, 10^{15} particles are needed in total. 100 FLOPs per time step are typically needed to calculate the new position and velocity for one particle in the iterative iPIC3D mover and, therefore, 10^{17} FLOPs are used to calculate particle trajectories per time step. Since typical iPIC3D simulations consists of 100,000-1,000,000 time steps, the total number of FLOPs required by the simulation of radiation belts is 10^{22} -10^{23}. This simulation on an Exascale supercomputer will require 10,000-100,000 seconds (27 hours) in the absence of any communication overhead. This amount of parallelism will be available only at Exascale.

4.1.4 Current Parallelization Approach

The iPIC3D code was initially written entirely in C++ with MPI C bindings and consists of approximately 10,000 lines of code, but now the code makes use of hybrid MPI + OpenMP. iPIC3D is used with HDF5, a parallel I/O library, and file format for storing and managing data. HDF5 supports a variety of data types, and is designed for flexible and efficient I/O and for high volume and complex data. The iPIC3D MPI communication is dominated by point-to-point communication, arising by communicating particles host cells among neighbor processes, and by global reductions arising from the solution of two linear systems every simulation time step.

The workflow of simulations by iPIC3D consists of approximately thousands of time steps, each of which carries out four major steps sequentially: interpolation of particles to grid, integration of Maxwell’s equations on the grid, interpolation of fields to particles, and integration of particle equation of motions.

There are mainly three scenarios where communication between adjacent tasks is required: field interpolation to calculate charge density (rho) and current density (J) at each grid node, Maxwell field solver to calculate electric field E and magnetic field B at each grid node, and particle mover to move exiting particles from source processor to destination processors.
4.1.5 AllScale Potential

The current implementation of iPIC3D suffers from considerable load imbalance problems, as depicted in Figure 3. As part of the AllScale project, we restructured the iPIC3D code to exploit recursive parallelism that can then benefit from the dynamic load balancing and resilience functionality supported by the AllScale Environment. We already changed the numerical scheme of iPIC3D, switching from the implicit formulation of the PIC numerical scheme to the explicit formulation, in order to allow for the recursive parallelism of AllScale to be introduced at global level. Consequently, we started developing a prototype version of iPIC3D allowing us to avoid global synchronization points and to use the AllScale Environment potential.

Figure 4: Time-stepping of one component in the recursive PIC algorithm

Figure 4 illustrates three time steps of one recursive component – the set of progressively smaller cubes. In the iPIC3D formulation, we apply the following principle: the 4D space is spanned by 3D data and simulation time steps are tiled. We also consider the scenario when particles move between cells.

Figure 5: PIC data movement at individual cell level

As Figure 5 illustrates on the individual cell level the communication required is quite similar. Instead of a left and right shadow elements from the previous time step, six shadow cells from the surrounding space are needed, which will be reflected in the structure of the 4D decomposition.
4.1.6 Requirements Specification.

The workload of iPIC3D mainly depends on the size of the computational grid and the number of computational particles and it consists of four major steps carried sequentially. Among these steps is the calculation of the field of the grid cells by solving a linear system, which represents the Maxwell’s equations. In the AllScale project, we proposed the explicit scheme to avoid the solution of a linear solver and the corresponding global synchronization on its each iteration.

iPIC3D uses the following data structure: each particle is represented by a struct data type and all the particles by an array of these structures; both a field and a cell are also represented by a struct. The latter contains also a list or an array of all particles associated to it. All the cells form a grid. The AllScale API provides primitives for multi-dimensional grids of static size.

As we calculate the grid cell on the flight and take the field values from a neighborhood cells, we restructured our data types, especially particles, in such a manner that they suite the nested task-based parallelism as well as will be manageable by the dynamic load balancer (the AllScale Scheduler). To enable proper usage of the AllScale Toolchain, we switched from the particles centric to the cell centric model. In this model, the particles movement among cells will be carried with a help of channels. We believe that this new approach suites well the load imbalanced problems such as the Van Allen radiation belts simulation, see Figure 6.

Figure 6: Earth’s (Van Allen) radiation belts with satellites. Credit: NASA.

Since we record data regarding to particle positions and velocities \((x,y,z,u,v,w)\) to reconstruct the trajectories, we require to have a suitable and scalable approach for recording this data into files for the further visualization. However, as
particles can change cells, meaning tasks, we require a support for parallel IO to handle these cases.

Concerning the resilience and fault tolerance, we require it to be provided through restart points, meaning dumping the information into a file every 5,000-10,000 steps. It would be beneficial for us if such service will be automatic checkpointing. Apart from the resilience and fault tolerance functionality provided by the AllScale Toolchain, we may consider to integrate some of those small and simple resilience and fault tolerance mechanisms inside the iPIC3D code. For instance, by implementing these mechanisms within the implicit field solver that relies upon the GMRES method or providing a polynomial variant of the GMRES method, which may reduce global synchronization to its minimum.

4.1.7 Benchmarks
One typical use case of iPIC3D is to simulate and detect high-energy particles trapped within the Earth’s radiation belts, see Figure 6. This problem is highly imbalanced by its nature. In the AllScale iPIC3D application, we expect to have initially 1,000,000 particles to be tracked. We also intent to perform wave particle interaction simulation, which is a load balanced problem, as a benchmark. These tests result can be compared with the reference simulations performed with the parent iPIC3D code.

In order to facilitate efficient tuning of the iPIC3D code within an intra- and inter-node level, we provide in Table 1 some realistic test cases.

<table>
<thead>
<tr>
<th>Problem size</th>
<th>Number of cells</th>
<th>Particles per cell</th>
<th>Time steps</th>
</tr>
</thead>
<tbody>
<tr>
<td>Test</td>
<td>$4^3$</td>
<td>$5^3$</td>
<td>10</td>
</tr>
<tr>
<td>Tiny</td>
<td>$8^3$</td>
<td>$8^3$</td>
<td>20</td>
</tr>
<tr>
<td>Small</td>
<td>$16^3$</td>
<td>$8^3$</td>
<td>40</td>
</tr>
<tr>
<td>Medium</td>
<td>$16^3$</td>
<td>$10^3$</td>
<td>512</td>
</tr>
<tr>
<td>Large</td>
<td>$160^3$</td>
<td>$12^3$</td>
<td>2,400</td>
</tr>
<tr>
<td>Real world</td>
<td>$320^3$</td>
<td>$16^3$</td>
<td>9,600</td>
</tr>
<tr>
<td>Real world</td>
<td>$5,000^3$</td>
<td>&lt;22$^3$</td>
<td>&lt;100,000</td>
</tr>
</tbody>
</table>

Table 1: Realistic test cases for iPIC3D.
4.2 **FINE™/Open: Unstructured CFD solver for large industrial unsteady simulations**

### 4.2.1 Introduction

Computational Fluid Dynamics (CFD), solving Navier-Stokes equations, is widely used in industry as a means to significantly reduce design turn-around times and improve product quality. To that end, the Fine™/Open CFD solver developed by NUMECA is used to run simulations on a few hundred millions of mesh points, on a few thousand cores, using second order accurate schemes, in steady Reynolds-Averaged Navier-Stokes (RANS) configurations. The request from industry towards higher fidelity of the simulations and increasing simulation reliability is much more demanding. Increasingly complex geometries must be handled, taking into account additional geometrical details, leading to very large mesh sizes in the order of the several billions of mesh points (Pirozzoli and Bernardini 2013). Furthermore, improved physical modeling, in particular turbulence, leads to the use of unsteady LES (Large Eddy Simulation) or DNS (Direct Numerical Simulation).

Moreover, in order to reduce the need for long lasting and costly experiments, industry demands turn-around times that are less than 24 hours. To fulfill those requirements, only reliable and accurate high-order CFD methods in combination with Exascale HPC will ensure closing the "confidence gap" and will enable rapid design processes significantly by reducing design costs – the latter being the driving motivational force of the industry.

### 4.2.2 Scientific / Commercial Impact / Stakeholders

The industrial applications of LES/DNS simulations for industrially relevant configurations, such as full aircrafts or engine components, will be a major step forward for the European industry. With AllScale, NUMECA expects to respond to those needs, as advanced LES/DNS simulations at the Exascale level will lead to improvement of designs, higher reliability and significant gains in productivity. High-level LES/DNS with turn-around times of one day, will reduce significantly the needs for experiments, and will boost NUMECA's position on the worldwide CFD market. Using of the AllScale Environment in Fine™/Open application will
allow NUMECA to run large unsteady CFD simulations using LES/DNS approaches and with real complex geometries.

### 4.2.3 Exascale Potential

DNS and LES require highly accurate and lengthy computations as well as extremely fine meshes when second order methods are used, which are hard to obtain on complex industrial geometries and high Reynolds numbers flows. In 2013 a direct numerical simulation (DNS), see (Pirozzoli and Bernardini 2013) has been conducted on an academic test case on 3-billion mesh points using a scheme of the order 5-6 and 32,000 processors, in just one day. Applied to industrial geometries, such simulations imply the use of much larger meshes sizes, ranging from 1 to 10 billion mesh points and even more depending on the order of the numerical scheme. In 2012, Spalart (Boeing), published (Spalart 2012) evaluating the number of mesh points required to run different turbulence models as well as the expected time when such computation could be run on commodity clusters (Spalart 2012). Those results clearly demonstrate the need of extreme scale architectures including Exascale systems and beyond to conduct the desired simulations with an adequate level of accuracy.

<table>
<thead>
<tr>
<th>Name</th>
<th>DNS</th>
<th>LES</th>
<th>DES</th>
<th>RANS</th>
</tr>
</thead>
<tbody>
<tr>
<td>Empiricism</td>
<td>No</td>
<td>Low</td>
<td>Medium</td>
<td>High</td>
</tr>
<tr>
<td>Unsteady</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>No (can be)</td>
</tr>
<tr>
<td># of points (Boing wing)</td>
<td>$10^{20}$</td>
<td>$10^{11}$</td>
<td>$10^7$ to $10^8$</td>
<td>$10^7$</td>
</tr>
<tr>
<td>In Service (Boing)</td>
<td>2080*</td>
<td>2045*</td>
<td>2010 (sub-regions)</td>
<td>1995</td>
</tr>
<tr>
<td>Vibration, Noise</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>No (buffet maybe)</td>
</tr>
</tbody>
</table>

Table 2: Spectrum of approaches to turbulence (*assuming Moore's Law holds).

### 4.2.4 Current Parallelization Approach

The current implementation of the Fine™/Open solver is based on pure MPI. The last scalability study has been performed on an industrial case, single block, of 60 millions of points on the super calculator AFRL Lightning CRAY XC30. Figure 8 shows the scalability curves obtained with and without the output of solution as well as the theoretical scalability curve, which presents a linear profile.
From this study, the following conclusions were drawn:

1. It is currently not possible to go above 2,000 processors due to a limitation in the definition of the tag required for MPI exchanges and synchronization. The limitation can be easily removed.
2. The output of solution can be considered as a sequential process and thus the performances of the flow solver are drastically reduced for large scale project. In the considered case, 1,000 iterations were performed and the solution was saved 10 times. Input/output have to be accounted for from the beginning of the project. For steady analysis, the solver saves the solution only once at the end of the simulation. However, for unsteady application, the user may want to save the solution after each physical time step.
3. Even if we already have a quite satisfactory efficiency, even for 30,000 cells/processor, it is necessary to profile our solver to be sure that there are no other implementation which could degrade the performances and the parallel efficiency.

The shared memory ‘property’ is not used in our current implementation and we do use mixed distributed/shared memory parallel approach.

We impose to our implementation that the iterative process to converge the solution is independent (if we consider only one multigrid level or several levels but with the sequential partitioner) of the number of processors. This means that the solver make several synchronization per iteration.

As explained earlier, FINE™/Open is currently using Open MPI (OMPI), which is an implementation of MPI. Within the AllScale project, we plan to benchmark only a small part of the solver. The maintenance of OMPI within AllScale
prototype will have to be managed. At this stage of the project, it is not yet clear how to handle that.

4.2.5 AllScale Potential
The implementation of the Fine™/Open solver used is based on MPI, which achieved reasonable scalability on several hundreds of cores. However, as the number of targeted cores grew to several tenths of thousands, severe scalability problems are expected, which would force the introduction of nested parallelism through the use of OpenMP. This implementation strategy would dramatically increase maintenance and development efforts. Furthermore, resilience to hardware failure still remains an open issue. By using the AllScale Environment these software development problems will be eliminated. At its core, Fine™/Open is based on isolated update operations on mesh points simulating individual time steps. It is possible to use a similar recursive decomposition for this mesh-time structure. This restructuring will reduce global communications, increase scalability, and introduce support for platform portability. Also, it will provide the Fine™/Open solver the capabilities of the AllScale’s sophisticated load balancing and resilience management services. In particular the auto-tuning of applications for customer target architectures as well as the configurable multi-objective optimization. The AllScale Environment will enable customers to utilize their computational resources more effectively in line with their own trade-offs among optimization objectives.

4.2.6 Requirements Specification
Fine™/Open is mainly used for complex configurations with several hundreds of millions of cells. The RAM memory required per million points is around 2 GB. In order to deal with unsteady complex configurations like full landing aircraft, it is necessary to limit the number of cells to around 50,000 cells (or even less) per processor to get acceptable industrial computing time, leading thus to the use of several thousands of processors.

Fine™/Open uses different techniques to accelerate the convergence of the iterative process such as the geometrical multigrid technique. A set of grids is created by agglomeration leading to polyhedral cells on coarse levels. The multigrid approach has to be considered for the final application.

The scalability and the parallel efficiency of the flow solver is also very important as well as the speed (CPU time/iteration) which has to be of same order as the current one.

For unsteady simulation, the number of I/O operations is proportional to the number of physical time steps or to a fraction of them. Since the number of cells is very high, an efficient parallel I/O needs to be implemented and cannot affect the efficiency.

4.2.6.1 Type of grids for FINE™/Open solver
In the FINE™/Open solver two type of grids can be currently used:

- Full-HEX mesh with hanging nodes, see Figure 9. The cells are only hexahedral and we can have hanging nodes (for a definition of hanging nodes see Section 7.1). In our HEXPRESS™ mesh generator we may also have prisms for some specific configurations. Within the AllScale project,
we will anyway consider that we have only hexahedral cells: meshes with prisms will not be considered.

Using our structured mesh generators IGC™ and AG5™, it is also possible to create Full-HEX structured mesh without hanging nodes. Note that these grids will be treated as unstructured grids even if they are structured by block; several structured blocks are assembled to create a Full-HEX block without hanging nodes. Anyway, these grids without hanging nodes are less complex to handle but is not a major part of our needs in the AllScale project.

• Hybrid mesh, see Figure 10. The cells can be tetrahedral, prism, pyramid or hexahedra. The mesh is conformal and there are thus no hanging nodes.

Usually, a mesh is composed of several blocks or domains. The domains are connected through Rotor-Stator interfaces in order to connect rotating from non-rotating blocks. Full Non-Matching Connection may also be used to connect blocks/domains with different physics (Conjugate Heat Transfer problem) or grids generated with different mesh generators. For simplicity, we will not consider multi block projects in the AllScale project. In terms of increasing complexity, we would have:

• Full-HEX mesh without hanging nodes
• Hybrid conformal mesh
• Full-HEX mesh with hanging nodes

For real-life applications, viscous effects cannot be neglected and the meshes need to be refined in the direction normal to the wall to capture viscous layers. The refinement is obtained by insertion of ‘viscous layer’ cells which can be identified in the mesh and in the flow solver, as depicted in Figure 11. Those
‘viscous layer’ cells are characterized by a high aspect ratio as illustrated in next figure.

![Figure 11: 'Viscous layer' cells](image)

From the definition of the hanging nodes and the explanations about refinement and the possible combinations of them, see Section 7.1, it is clear that Full-HEX mesh with hanging nodes is quite complex to handle. However, the ‘Full-HEX’ property of the meshes is very important for NUMECA since it is quite unique on the market and allows refinement/coarsening of the grid, which is a strong saling point.

At this stage of the project, we can consider that only Hybrid meshes can be used for the project but the ‘Full-HEX’ property should be kept in mind for the future extension of the developments that will be done within the AllScale project.

### 4.2.6.2 Multigrid agglomeration

When receiving geometry, a mesh is created around and/or inside the geometry. A computational domain is created and the domain will contain topological surfaces, on which boundary condition will be applied, topological edges and topological vertices.

The mesh stored in the mesh file is only the finest level of the mesh. The mesh generation process does not create the cell hierarchy. As explained before, it is possible that faces have children, the same remark apply to edges. Of course for Full-HEX mesh without hanging nodes or for hybrid mesh, the cells, faces and edges have no children.

In order to accelerate the convergence of our iterative process, a multigrid approach is used. From the mesh, coarser meshes are generated by agglomeration. Cells are agglomerated to form polyhedral cells that we call ‘agglo’ cell. The ‘agglo’ cells have only ‘agglo’ faces and do not contain information on the edges or vertices. On coarse levels, there are no edges and no vertices. Only ‘agglo’ cells and ‘agglo’ faces are present on coarse multigrid levels. A connection parent-child and child-parent are used to connect cells and faces between two successive levels.
Remark: in the parallel solver, we can impose that each processor has the full hierarchy of cells. This means that all the cells on the coarsest level have their children, grand children, grand-grand children available on the processor. This brings a lot of simplifications and reduces the communications without affecting too much the parallel performances of the solver.

4.2.6.3 Flow solver

4.2.6.3.1 Principles
The flow solver is based on explicit iterative Runge-Kutta methods. Some implicitness can be introduced to accelerate the convergence process but since this part of the code cannot be communicated, we will only consider explicit methods in the AllScale project. As mention earlier, multigrid cycles are used to accelerate the convergence. Different types of cycles can be used. We will first focus on V-cycles. Different schemes are available for the space discretization: 1st order upwind, 2nd order upwind, 2nd order central scheme with scalar dissipation, and 2nd order scalar scheme with matrix dissipation. In order to simplify the stencil, we can also neglect viscous effects and turbulence.

Additional information about stencils, data structure and multigrid is given in Section 7.2.

The equations are solved in the cell centers and the boundary conditions are applied on the boundary faces, boundary edges (finest level) and boundary vertices (finest level). The values in vertices (only on finest level) are obtained by linear interpolation of the solution in cell centers and boundary faces. On finest level we also use boundary edges, which are connected to topological edge and corners connected to topological vertices to increase the accuracy, but we may consider skipping them for the prototype. In the stencils that we will present, if the cell is connected to a topological face through a boundary face, the neighboring cell has simply to be replaced by the boundary face information.

The final goal of the project is to be able to run unsteady turbulent flow with one-equation turbulence model. Our preference would go for the central scheme but 2nd order upwind could also be acceptable.

4.2.6.4 Expectations concerning the overall system performance
- Using the AllScale Environment, we would be able to run FINE™/Open on all computing systems of the AllScale computing infrastructure.
• Using the AllScale Environment, we should be able to reduce the amount of global synchronization and thus improve the performances of the FINE™/Open solver. A first conservative objective is to reach the same performances of the current FINE™/Open solver. A more challenging objective is to reach an efficiency of 85%-90% on 10,000 cores for a mesh of ~1,000,000,000 cells.

• The AllScale Environment should be such that any extension to other type of grid, physics, discretization schemes has to be easy and user-friendly in order to reduce significantly the ‘parallel’ development for such applications.

4.2.6.5 Resiliency
Resiliency is still an open question. Assuming a mean time between failures (MTBF) of 1 hour, a global checkpoint/restart mode with solution staging on a parallel file system might still be feasible. However, it is not clear how fast of a restart could be. The AllScale proposal mentions a task-level checkpoint/restart, which is of our interest, however we wonder how well this actually fits with FINE/Open framework. We will focus on hardware or network failures, not bit errors (we suppose ECC memory should guard against this).

4.2.7 Benchmarks
The final goal of the project is to run an unsteady turbulent flow simulation on a hybrid mesh using four multigrid levels using the 2nd order upwind scheme or preferably the 2nd order central scheme. Scalability and efficiency of the developed algorithms will be measured and compared to current ones.

In the definition of the complexity of the final benchmark, four items must be accounted for: the type of the mesh (structured – unstructured full HEX w/o hanging nodes and hybrid meshes), the multigrid, the physics we want to solve and the spatial and temporal schemes we use.

For each item, the following steps can be considered to ease the testing of the prototype and the developments of required functionality:

• Mesh: From 1D Full-HEX (without hanging node) to 2D Full-HEX (without hanging node) then 3D Full-HEX (without hanging node) and finally 3D Hybrid mesh;

• Multigrid level: From 0 (single grid) to 4 grid levels;

• Physics: From Euler (Non viscous flow) to Laminar flow (Viscous flow) and then Turbulent flow (Spalart Allmaras turbulent model);

• Scheme: First upwind 1st order then upwind 2nd order and at last central matrix scheme.

The combination of these four items will create a list of test cases from the simpler to the more complex in terms of coding effort. Each test case will be used to:

• Validate quantitatively the developed algorithms by comparing the solution with the one given by existing solver running on the same grid with the same scheme and solving the same physics;
D2.2 – Requirement Specifications and Reports on External Technological Developments (b)

- Compare the speed, scalability and parallel efficiency obtained with existing pure MPI based solver.

Table 3 summarizes the list of benchmarks that have to be considered.

<table>
<thead>
<tr>
<th>Problem Size</th>
<th>Case</th>
<th>#cells</th>
<th>#procs</th>
</tr>
</thead>
<tbody>
<tr>
<td>Tiny</td>
<td>1D Heat Conduction</td>
<td>$10^3 - 10^5$</td>
<td>$10^0 - 10^2$</td>
</tr>
<tr>
<td>Small</td>
<td>2D Structured Heat Conduction [Steady]</td>
<td>$10^3 - 10^6$</td>
<td>$10^0 - 10^2$</td>
</tr>
<tr>
<td>Small</td>
<td>2D Structured Blast [Unsteady – All schemes]</td>
<td>$10^3 - 10^6$</td>
<td>$10^0 - 10^2$</td>
</tr>
<tr>
<td>Small</td>
<td>2D Unstructured Heat Conduction [Steady]</td>
<td>$10^3 - 10^6$</td>
<td>$10^0 - 10^2$</td>
</tr>
<tr>
<td>Small</td>
<td>2D Unstructured Blast [Unsteady – All schemes]</td>
<td>$10^3 - 10^6$</td>
<td>$10^0 - 10^2$</td>
</tr>
<tr>
<td>Medium</td>
<td>3D Structured Heat Conduction [Steady]</td>
<td>$10^6 - 10^8$</td>
<td>$10^0 - 10^4$</td>
</tr>
<tr>
<td>Medium</td>
<td>3D Structured Blast [Unsteady – All schemes]</td>
<td>$10^6 - 10^8$</td>
<td>$10^0 - 10^4$</td>
</tr>
<tr>
<td>Large</td>
<td>3D Unstructured Heat Conduction [Steady]</td>
<td>$10^6 - 10^8$</td>
<td>$10^0 - 10^4$</td>
</tr>
<tr>
<td>Large</td>
<td>3D Unstructured Blast [Unsteady – All schemes]</td>
<td>$10^6 - 10^8$</td>
<td>$10^0 - 10^4$</td>
</tr>
<tr>
<td>Real-world</td>
<td>3D Unstructured full aircraft [Steady]</td>
<td>$10^8 - 10^9$</td>
<td>$10^3 - 10^5$</td>
</tr>
<tr>
<td>Real-world</td>
<td>3D Unstructured full aircraft [Unsteady – All schemes]</td>
<td>$10^8 - 10^9$</td>
<td>$10^4 - 10^6$</td>
</tr>
</tbody>
</table>

The additional objectives for the final benchmark are:

- Efficient parallel I/O: Writing the solution should have no impact on the scalability and efficiency. Since the I/O could be a bottleneck, for each case, the scalability and efficiency will be measured with and without I/O. For steady applications, the output can be done only at the end of the simulation. For unsteady applications, the possibility to introduce an I/O frequency will be foreseen in order to limit the number of outputs.

- Portability: Ability to run on any architecture targeted by the Allscale architecture.

Two additional general objectives should be accounted for from the beginning of the project:

- The developed algorithms should allow a friendliness extension to Full-HEX mesh.
- The user-friendliness of the API and the possibility to extend it, once the project is finished, and to apply it to more complex physics (combustion, radiation, CHT) and high-order schemes.
4.3 AMDADOS: Adaptive Meshing and Data Assimilation for the Deepwater Horizon Oil Spill (IBM)

4.3.1 Introduction
The Deepwater Horizon oil spill is the largest accidental spill in the history of the petroleum industry. The BP blow-out lasted for 87 days, releasing approximately 4.9 million barrels (780,000 m$^3$) into the surrounding environment. Authorities collected huge volumes of data concerning the extent and evolution of the oil spill. While previous research has made use of some of the data, a system that harnesses the full potential of the dataset by integrating it with a set of highly accurate, adaptive models and meta-models is yet to be put in place. This is of paramount importance to understand related phenomena and take preventive actions or otherwise minimize damage to the environment and to local societies at risk. Data assimilation (DA) is a mathematical technique that enables the incorporation of physical observations within complex models. In each simulation step, observation data is combined with output from the model, yielding results that are considered as ‘the best’ estimate of the current state of the system. Adaptive Meshing (AM), on the other hand, is a method of dynamically changing the precision of a model by targeted refinement of the numerical grid resolution.

![Figure 13: Details of the region for DA-AM (black contour) and sources of observations (dots)](image)

Table 4: Computational cost of AM and DA

<table>
<thead>
<tr>
<th>Nesting Region</th>
<th>Resolution (meters)</th>
<th># cells ($10^9$)</th>
<th>AM FLOPS</th>
<th>DA FLOPS</th>
<th>Data per day (TB)</th>
</tr>
</thead>
<tbody>
<tr>
<td>1) Global Model</td>
<td>100</td>
<td>0.03</td>
<td>$6*10^{13}$</td>
<td>$3*10^{18}$</td>
<td>0.2</td>
</tr>
<tr>
<td>2.1) AM oil</td>
<td>20</td>
<td>0.01</td>
<td>$1*10^{14}$</td>
<td>$1*10^{18}$</td>
<td>0.07</td>
</tr>
<tr>
<td>2.2) AM oil</td>
<td>4</td>
<td>62.5</td>
<td>$3*10^{18}$</td>
<td>$6*10^{22}$</td>
<td>432</td>
</tr>
<tr>
<td>3.1) AM coast</td>
<td>20</td>
<td>0.03</td>
<td>$3*10^{14}$</td>
<td>$3*10^{18}$</td>
<td>0.2</td>
</tr>
<tr>
<td>3.2) AM coast</td>
<td>4</td>
<td>0.34</td>
<td>$2*10^{16}$</td>
<td>$3*10^{20}$</td>
<td>2.35</td>
</tr>
<tr>
<td>4.1) AM</td>
<td>20</td>
<td>25.0</td>
<td>$1*10^{17}$</td>
<td>$2*10^{19}$</td>
<td>172</td>
</tr>
</tbody>
</table>
In this pilot application, DA and AM are used jointly and embedded in a modeling implementation of the advection diffusion equations for simulating the Deepwater Horizon accident, see Figure 13. The system autonomously increases with AM the resolution at targeted locations, while DA incorporates observations into the model forecast. While advection diffusion codes for transport phenomena (such as oil spills) exist and are well developed, a novel one, that embeds AM-DA, that is scalable to harness all the available data, is unprecedented and significantly improves resolutions and accuracy in order to better understand the impact of the Deepwater Horizon accident. The volume of data simulated, see Table 4, makes AMDADOS extremely data intensive, challenging main memory design points for future Exascale machines, in fact the total volume of modeling outputs generated over the two months time windows of the Deepwater Horizon will be in the order of 100 Pbytes.

4.3.2 Scientific / Commercial Impact / Stakeholders
AMDADOS aligns with IBM Smarter Planet (IBM 2016), an IBM program that seeks to highlight how forward-thinking leaders in business, government and civil society around the world are capturing the potential of smarter systems to achieve economic growth, near-term efficiency, sustainable development and societal progress. AMDADOS’s code and modeling data will be incorporated as an outcome of Smarter Planet.

4.3.3 Exascale Potential
The computational cost of the AMDADOS system scales with grid resolution and the volume of data assimilated. The spatial domain of the AM-DA modeling system is discretized with a non-uniform adaptive mesh. The computational costs of the AM scheme and the DA scheme are detailed in Table 4, listing approximate floating-point operations (FLOPS) required to simulate one time step of the simulation of the oil spill. Depending on the current state of the system and the availability of observational data, the AM-DA scheme covers a range of implementations from simulating/assimilating the global model only to including all AM regions shown in Table 4. Across the various options, the required amount of computational resources to process any of these combinations demands Exascale systems.

4.3.4 AllScale Potential
The computational nature of the AM-DA system has three components, which fit well with the AllScale programming model. The formulation of the advection diffusion numerical scheme produces a 3D stencil application enabling a recursive spacetime decomposition. Furthermore, DA employs a set of demanding matrix/vector operations, which can be recursively formulated, thus providing additional nested parallelism. The same holds for the AM components that are inherently recursive and can directly benefit from the AllScale
D2.2 – Requirement Specifications and Reports on External Technological Developments (b)

Environment. Also, the evolving nature of AM leads to large load imbalances, requiring a continuous, dynamic redistribution of load across processors to effectively utilize available resources. Therefore, AMDADOS will benefit from AllScale’s sophisticated load management services. Finally, AllScale’s integrated resilience management system will enable AMDADOS to handle isolated hardware failures resulting in reducing the cost for checkpointing operations as well as the expenses for failed computations.

**Objective for AMDADOS system as part of AllScale.** As part of AllScale we plan to achieve the following objectives: (A) a modeling resolution of 4 meters in the key affected areas of Deepwater Horizon; (B) the assimilation of the entire dataset (NOAA 2016) into the AM-DA system. Both A and B are unprecedented. In fact, the maximum resolution previously achieved is 20m (Computerworld 2010) and no research institution has succeeded in assimilating the entire NOAA dataset. Achievement of A and B will constitute a benchmark for years to come in terms of both operational response planning and volume of modeling datasets for detailed environmental analysis. Beyond the Deepwater Horizon incident itself, the planned work through the use of a coupled DA and AM modeling approach will create a novel paradigm for operational oil spill response systems, particularly in areas with complex ecosystems such as beaches, wetland and estuaries.

### 4.3.5 Requirements Specification

The main area of requirements for AMDADOS is extreme parallelization and achievement of real time with a prescribed spatial resolution. The AllScale AMDADOS application consists primarily of grid-based and cell-based data structures to manage the solution of the advection-diffusion equation and integrate with data assimilation and adaptive meshing concepts. Grid structures contain information on each local subdomain and their relative position within the global domain. Information on variables such as model outputs, spatially variant model inputs and observation data for assimilation are contained within grid structures. Boundary update and synchronisation between neighbouring subdomains via an iterative algorithm are also managed. Cell data structures contain information on solution within subdomains. Operations such as projecting the solution forward in time, filtering solution with observations and transitioning from three different levels of grid refinement (100m, 20m and 4m) are implemented on these data structures.

To make extreme parallelism feasible, it is expected that AllScale will enable local stencil operations, global reductions, and parallel loop operations. Those will be applied to the numerical operations that characterize AMDADOS and that are suitable. In the specifics, the AllScale enablement will target the use of arrays and linear algebra structures like matrix and tensor operations. The application contains a number of extremely computationally demanding matrix/vector operations that exploit linear algebra libraries.

To provide computations in real-time, the expectation is that AllScale will enable the ability to assimilate the entire NOAA dataset, see Table 4, and to run AMDADOS on all computing systems of the AllScale computing infrastructure.
These requirements being met provide the platform to simulate the Deepwater Horizon spill in real-time, at unprecedented model grid resolution and with the assimilation of all available data measurements.

Resiliency of the system is of critical importance as simulating the evolution of the oil over the duration of the event requires a two months simulation. The application will contain restart checkpoints at regular intervals to ensure that if simulation halts, it can be started from the last checkpoint. Further, acceptable resilience at the MTBF expected at Exascale level will require an automated, integrated resilience management system, as that provided by the AllScale system.

### 4.3.6 Benchmarks

We are not aware of any system that tries to achieve both DA and AM for the Deep Water Horizon accident. The actual benchmark is measured in terms of resolution, as the 20 m one claimed in (Thibodeau 2010).

The final goal of the pilot application is simulation of the entire time period of the spill event at targeted resolution up to 4m while assimilating the entire observational dataset (NOAA, US National Oceanic And Atmospheric Administration 2016). In order to perform intermediate analysis of performance before the full system deployment AMDADOS will be run on selected spatial areas of the Gulf and selected temporal windows of the Deep Water Horizon accident. Table 5 presents a set of test cases for the AMDADOS application that will allow testing and tuning across a wide range of scales from laptop to the Exascale level.

<table>
<thead>
<tr>
<th>Problem size</th>
<th>Description</th>
<th>Number of cells</th>
<th>DA</th>
<th>AM</th>
<th>Data per day (TB)</th>
<th>Notes</th>
</tr>
</thead>
<tbody>
<tr>
<td>Tiny</td>
<td>Centre on BP platform</td>
<td>1x10⁴</td>
<td>1 dataset</td>
<td>100m</td>
<td>7x10⁻⁴</td>
<td>100 x 100 m</td>
</tr>
<tr>
<td>Small</td>
<td>Oil 1</td>
<td>2.5x10⁵</td>
<td>1 dataset</td>
<td>20m</td>
<td>2x10⁻²</td>
<td>20 x 20 m</td>
</tr>
<tr>
<td>Medium</td>
<td>Oil 2</td>
<td>6.25x10⁶</td>
<td>1 dataset</td>
<td>4m</td>
<td>4x10⁻¹</td>
<td>4 x 4 m</td>
</tr>
<tr>
<td>Large</td>
<td>Entire oil extents</td>
<td>6.29x10⁹</td>
<td>3x10¹⁸</td>
<td>FLOPS</td>
<td>4 – 20 m</td>
<td>430</td>
</tr>
<tr>
<td>Real-world</td>
<td>Entire domain</td>
<td>2.44x10¹⁰</td>
<td>~10⁻²¹</td>
<td>FLOPS</td>
<td>4 – 20 m</td>
<td>1680 AM oil + coast</td>
</tr>
</tbody>
</table>

Table 5: AMDADOS test cases ranging from tiny to real-world applications

### 4.4 Benchmarking Environment

The AllScale benchmark suite will contain the three AllScale pilot applications as well as the AllScale Environment components test cases. They will be packed together with the instructions on how to run them. As we will run the AllScale pilot applications and, therefore, the AllScale Environment numerous times on various platforms, we propose to use a benchmarking framework as a wrapper.
D2.2 – Requirement Specifications and Reports on External Technological Developments (b)

for these test cases. Needless to mention that configuring, compiling, and running benchmarks on several machines, followed by result verification and analysis takes a lot of time and energy. Without a benchmarking environment all these procedures have to be carried out manually. In addition, we suggest to use such framework for automatic testing. For instance, on git push or as overnight testing in order to identify any possible errors and fix them quickly.

One of such frameworks is Jenkins (Jenkins 2016) -- an open source, cross-platform, continuous integration and continuous delivery framework that provides functionality to support building, testing, deploying, and automation for virtually any project. Jenkins is in use by different European projects and benchmarks such OmpSs, StarPU, etc.
5 AllScale Objectives

We address the following fundamental problems encountered when extending state-of-the-art parallelization approaches to extreme scale systems:

- Conventional parallel paradigms hamper performance portability and global optimization;
- Flat parallelization approaches are unfit for large-scale parallel computing;
- Optimization is limited to single objectives lacking explicit energy and power budgeting;
- Compositions of runtime systems requires manual coordination to exploit all levels of parallelism;
- Increased probability of errors in Exascale computing systems.

The objective of the AllScale Environment is to eliminate – or at least mitigate – each of these problems.

AllScale has set six major scientific objectives focusing on a programming environment for the Exascale computing. In order to meet these objectives AllScale will catalyze the development of scalable, performance-portable and resilient applications in science and engineering. These applications will be optimized to achieve shorter execution-time, lower the energy, power and resource usage and support resilience, and with all these optimizations considered together, will adapt effectively to the parallel computing systems of all scales.

As AllScale is concerned with a programming environment, algorithmic development and efficient implementations, it is for most measures of success not useful to define an absolute target, but rather to report the development relative to the baseline established at the start of the project.

Objective 1: A single-source-to-any scale development environment. AllScale provides the programmer with a unified API to express parallelism at a higher level of abstraction, by using the standard C++ templates. The AllScale Compiler and Runtime System map a program to a given target architecture, through appropriate implementation supporting load balancing, locality management, and resilience management strategies. Due to this unified perspective, we expect to achieve significant advances in the automated applications tuning and, thus correspondingly providing substantially improved programmer productivity, through the automated performance portability of codes.

Measure of success: We will provide a single, concise, unified implementation of the AllScale pilot applications on multiple shared memory, distributed memory, and accelerator-equipped HPC systems, as well as for standard desktop hardware. The developers of the pilot applications will compare the (estimated) effort, which is required to replicate the AllScale functionality – by including the

Copyright © AllScale Consortium Partners 2017
fine-grained load balancing, performance portability, and resilience – in conventional programming models, against the effort required to produce the equivalent with AllScale.

**Objective 2: Maximize and exploit the potential of nested recursive parallelism in HPC.** Recursive algorithms overcome inherent limitations of conventional iterative algorithms that hamper performance, scalability and adaptivity of HPC applications. In particular, global communication and synchronization steps can be substantially reduced with the use of recursive parallelism. Furthermore, the fractal structure of nested recursive parallelism provides a foundation for the transparent integration of localized, sophisticated and flexible dynamic load management and failure handling routines.

*Measure of success:* We will provide nested recursive implementations of all pilot applications using AllScale and demonstrate the suitability of the recursive programming approach on a range of parallel architectures, from small to extreme-scale ones. We aim to demonstrate a reduction of global communication and synchronization operations of the pilot applications by 80%, compared to their baseline implementations based on conventional iterative parallelism.

**Objective 3: Multi-objective optimization for execution time, energy and resource usage.** The AllScale Environment will provide a multi-objective scheduling and optimization component capable of steering application execution targeting to satisfy dynamic, user-definable optimization trade-offs, which are specified as goal functions. This will be accomplished by (i) selectively employing the distinct code versions generated by the compiler, (ii) adjusting the mapping of work and data to the target architecture, and (iii) the management of hardware resources and customization of their parameters, e.g. frequency scaling.

*Measure of success:* We will demonstrate a trade-off between the objectives of execution time ($t$), energy consumption ($e$), and resource usage ($r$) for the three pilot applications. In particular, when running the pilot applications and varying the goal functions, in order to minimize the term $t^ne^mr^k$, for all $0 \leq n, m, k \leq 2$, the resulting execution times, as well as the energy and resource consumption, will feature a difference of more than 30% between the minimum and maximum for each specified objective. Furthermore, our approach will reduce either energy or resource consumption, by at least 25%, while not causing an execution time penalty bigger than 10%, when compared to a manually tuned version.

**Objective 4: A unified runtime system orchestrating parallel program executions on target architectures.** We will develop a unified runtime system, which will be able to tune the parallel execution of the entire application. Thus, the current drawback in coordinating multiple runtime systems, while targeting different levels of parallelism within a program, will be eliminated by the AllScale design. The AllScale Runtime System will dynamically manage the distribution of workload and data throughout for a given target architecture, and based on the options introduced by the AllScale Compiler. This includes (i) the allocation of necessary resources, (ii) adaptation of the underlying hardware, e.g.
clock speed or prefetching options, and (iii) resiliency management support, leading to a customizable system allowing to tune, at the same time, multiple objectives with different trade-offs (for example, minimization of execution time, energy consumption and resource usage).

**Measure of success:** The AllScale Runtime System will run the three pilot applications on all the computing systems, which are part of the AllScale computing infrastructure. We will demonstrate parallelism by executing “billions” of tasks, and achieve the enhanced resource utilization and/or energy efficiency, validating it by the three pilot applications, when compared to their baseline implementations. What we mean by “billions of tasks” is the ability that they are concurrently executed on billions of envisaged Exascale cores, as these tasks can be any type of process, for example, to copy data.

**Objective 5: Mitigating increased risk of soft/hard hardware faults.** To react to the anticipated error rates in future Exascale systems, AllScale will follow an innovative holistic approach. We will integrate the AllScale Resilience Manager, Monitoring Framework, Compiler, and Runtime System, for resilience strategies. With the help of the AllScale Compiler and Runtime System, an automated, task-level checkpoint-restart facility will be developed. To recover from soft faults, application-specific checks will be provided to the AllScale Environment. In order to recover from hard faults, the AllScale Resilience and Fault Tolerance Management, Monitoring Framework, and Runtime System, will be key recovery components.

**Measure of success:** The use of checkpointing at different granularity levels will allow us to measure the achieved success as follows. Given the possibility of different granularity of checkpointing, our measure of success is:

i) If we checkpoint at such a granularity level which allows to run faster than the version not using checkpoint/restart mechanism in the presence of faults.

ii) Given a number of possible granularity levels which are efficient in the sense of (i), the most efficient granularity level is chosen, in respect to the application runtime in the presence of faults.

**Objective 6: A monitoring framework for online analysis of non-functional parameters.** To enable the multi-objective optimization in the AllScale Environment (see Objective 3), a lightweight monitoring service with the dynamic introspection capabilities is required. We will build it on top of the HPX performance counters, as a monitoring component, which will link the system performance data with the application source code. This monitoring component will provide the collected data back to the Runtime System, in order to facilitate its task scheduling. Furthermore, we will investigate the efficient storage structures for non-functional data, such as flow graphs as well as the utilization of previous collected data to detect inconsistencies in an application's behavior.

**Measure of success:** AllScale will provide the Runtime System with online access to non-functional data (time, power, energy, etc.) of tasks and their respective
D2.2 – Requirement Specifications and Reports on External Technological Developments (b)

code regions, while the application runs at any scale level. This will be achieved while not exceeding 5% overhead for the total program execution time, and validated for the three pilot applications on all the architectures available in the AllScale computing infrastructure. Furthermore, the monitoring framework will store the collected non-functional information in an innovative efficient format, with reducing the storage size by a factor of at least 1.5, when compared to the storage size, which is required by the state-of-the-art event tracing techniques.
6 Conclusions and Future Work

The initial requirements on the AllScale Environment were establish within the first months of the project to drive and facilitate the project development. At first, a detailed investigation of the pilot applications has been carried out by KTH, NUM, and IBM, resulting in a preliminary description of the application-specific requirements for the AllScale Environment. This work was followed by the definition of the interface and functionality requirements component-wise within the AllScale Environment by UIBK, FAU, KTH, QUB, and IBM. In addition, the AllScale Environment also imposed requirements towards the pilot applications. The requirements and constrains definition has been regularly discussed and revised during WP2 bi-weekly conference calls as well as during the cross-reviewing phase from January to February. Moreover, some of the AllScale objectives and their measure of success have been revised to precisely represent the project goals. Furthermore, a corresponding technology watch function was set up and the external technological developments have been collected by KTH, UIBK, FAU, and QUB to ensure that the AllScale Environment adapts and responds to both hardware and software evolutions. This effort led to an accurate definition of requirements and external project-related developments described in this deliverable (D2.2).

The requirement specifications will be continuously updated based on the knowledge and experience obtained from the planned project development and research activities. The external technological developments will also be continuously updated through the technology watch function.
7 Annex: FINE™/Open

7.1 Full-HEX mesh

7.1.1 Refinement for Full-HEX mesh

For full-HEX mesh, there are several ways to refine a cell; the combination of the different refinements possibilities may lead to some complex configurations. Suppose that we have two cells: left and right and we attach to each of them three main directions $\xi, \eta, \zeta$. We have seven ways to refine a cell:

- **Isotropic refinement** [$\xi, \eta, \zeta$]: The cell is refined in the three directions so that 8 child cells are created.

  ![Figure 14: $\xi, \eta, \zeta$ refinement](image)

- **Anisotropic refinement** [$\xi, \eta - \xi \zeta - \eta \zeta$]: Each cell is refined in only two directions so that each cell has four children.

  ![Figure 15: $\xi, \eta - \xi \zeta - \eta \zeta$ refinement](image)

- **Anisotropic refinement** [$\xi - \eta \zeta$]: Each cell is refined in only one direction so that each cell has two children.

  ![Figure 16: $\xi - \eta \zeta$ refinement](image)

7.1.2 Hanging node

The definition of a hanging node depends if we are in the context of the mesh generator or the solver. For the mesh generator, a hanging node is a node lying on a face and not connected to that face through the face-node connectivity. For the flow solver, a hanging node is a node that connects two edges that have the same parent edge. In Figure 17, some configurations with hanging nodes are illustrated.
Suppose that we have a face f1 between a left cell and a right cell. The right cell is refined and subdivided in 8 cells, then:

- Face f1 which still exists since the left cell has not been refined has 4 children: f2, f3, f4 and f5
- The edges e1, e2, e3 and e4 still exist and each edge has two children e5, e6 – e7, e8 – e9, e10 – e11, e12.
- In the mesh generator:
  - The vertices v5, v6, v7 and v8 are hanging nodes since they lie on face f1 but are not part of the face-node connection.
  - Vertex v9 is also a hanging node. It is indeed on face f1 but it is not connected to the face f1 by the face-node connectivity. Indeed, only v1, v2, v3 and v4 are connected to face f1, i.e. only v1, v2, v3 and v4 are used to define the face f1.
- In the flow solver:
  - The vertices v5, v6, v7 and v8 are hanging nodes since the two edges at which they are connected have the same parent. Vertex v5 is connected to e5 and e6 and both edges have as parent e1. Note that v5, v6, v7 and v8 are not hanging node for the represented faces but for the other face of left cell.
  - Vertex v9 is a special vertex since for the mesh generator is considered as a hanging node. For the flow solver, v9 is not a hanging node since as we will see further f1 does not exist in the flow solver. V9 being part of the face-node connection for f2, f3, f4 and f5, it is not a hanging node.
Suppose for that case, see Figure 18, that there is two cells, the left one is not refined, while the right one is refine first in the $\eta$ direction. The face $f_1$ is split into two faces $f_2$ and $f_3$. If we refine only one child cell of the right original cell, we end up with the configuration shown in figure 17. The face $f_3$ is not refined and the face $f_2$ is split into two faces $f_4$ and $f_5$. Vertex $v_7$ is a hanging node for both the mesh generator and the flow solver. Indeed, the vertex $v_7$ is on the face $f_3$ and is not accessible through the face–vertex connection of the face $f_3$. It connects two edges $e_{10}$ and $e_{11}$, which have the same parent edge $e_9$.

### 7.1.3 Combination of refinement

Starting from two cells, there are a lot of combinations of refinement possible. However, not all of them are possible in reality. There is one main limitation is that we must have at most one hanging node per edge. There are also forbidden configurations that violate the rule defined for the mesh generator data structure. Let us consider the following refinement combination: the left cell is refined along the $\eta$ direction while the right cell would be refined along the $\zeta$ direction. This combination is not possible since it would create two hanging nodes on an edge.

First the left cell, $c_1$, is refined along the $\eta$ direction. The face $f_1$ which was connecting the original left cell $c_1$. The right cell $c_2$ is divided into two faces $f_2$ and $f_3$. The face $f_1$ still exists since the right cell $c_2$ has not been refined.

When the cell $c_2$ is refined in two cells $c_5$ and $c_6$, the face $f_1$ is split into two faces $f_4$ and $f_5$. At the end of the refinement, we have a configuration which is not possible since it is impossible to connect through parent-child connectivity the four new faces $f_2$, $f_3$, $f_4$ and $f_5$, see Figure 20.
7.2 Flow solver: stencil – data structure – multigrid

7.2.1 Stencils

7.2.1.1 *Euler* (no viscous effect) – first order upwind

For Euler flow using the first order upwind scheme, only the face neighbors are requested to compute the residual and thus the solution in the cell center of a given cell. This is true on all grid levels. Note that for the Upwind 1st order, stencils are the same for both the coarse and the finest level.

7.2.1.2 *Euler* (no viscous effect) – second order upwind

![Figure 23: Stencil for Euler - Upwind 2nd order – Finest level](image)
For Euler flow using the second order upwind scheme, the face neighbors as well as the neighbors by faces of the neighbors by face of the given cell are requested to compute the residual and thus the solution in the cell center of a given cell, see Figure 23. This is true only on the finest level. On the coarse level, the upwind first order scheme is used and thus the related stencil see figure 23.

7.2.1.3 Viscous – first order upwind

For a viscous flow using the first order upwind scheme, the neighbors by face as well as the neighbors by nodes of the given cell are requested to compute the residual and thus the solution in the cell center of a given cell on the finest level. On the coarse level, the neighbors by face are requested as well as the neighbors by faces of the neighbors by face of the given cell, see Figure 26.

7.2.1.4 Viscous – second order upwind
For a viscous flow using the first order upwind scheme, the neighbors by face as well as the neighbors by faces of the neighbors by face of the given cell are requested to compute the residual and thus the solution in the cell center of a given cell. This is true only on the finest level see Figure 27. On coarse level the upwind first order scheme is used, see Figure 28.

7.2.2 V-cycle multigrid
As mentioned earlier, a explicit 4 stages Runge-Kutta scheme is used. We usually split the fluxes computation in three steps:

- **The convective fluxes (or non viscous fluxes):** They are computed at each Runge-Kutta stage and just before the restriction (because they are needed to do the restriction). Note that the restriction is the operation to go from the fine to the coarse level.

- **The viscous fluxes:** They are only computed at the first Runge-Kutta stage and just before the restriction (because they are needed to do the restriction).

- **Artificial dissipation:**
  - For central scheme, the dissipation is only computed at first and second stage and just before the restriction.
  - For upwind scheme, the dissipation is computed at all Runge-Kutta stages and just before the restriction.

On each multigrid level, we may apply different number of sweeps. By default, 1 sweep is applied on finest level, 2 sweeps on the next coarser level, and three on next coarser level and so on.
Let’s give some additional information about the terminology we use. First we have to make a distinction between steady simulation and unsteady simulation:

- Steady simulation: there is no unsteadiness in the flow field or the flow field reach a steady state; there is thus one unique solution, which is obtained at the end of an iterative process. The iterative process needs several iterations to converge. When we say one iteration, it is in fact one V-cycle. In the solver, a V-cycle starts on the finest grid as indicated in the Figure 30 and finishes just after the last prolongation (operation done to transfer the solution from finest level-1 to finest level). The last prorogated solution is the starting solution for the next iteration or next V-cycle.

- Unsteady simulation: there is unsteadiness in the flow field and the solution evolves in function of physical time. We have thus to introduce a physical time step and have to compute a solution for each physical time step. Of course, the times steps, three of them, are linked and the solution at time $t^n$ and $t^{n-1}$ is used to obtain the solution at time $t^{n+1}$. At each physical time step, we compute a pseudo-steady state problem; this means that the solver described for steady problem is called with one additional source term to express the link with $t^n$ and $t^{n-1}$ solutions. So, at each physical time step, we perform some iterations, where each iteration is a V-cycle.
D2.2 – Requirement Specifications and Reports on External Technological Developments (b)
Bibliography


D2.2 – Requirement Specifications and Reports on External Technological Developments (b)


Computerworld. 2010.


Fasi, Massimiliano, Julien Langou, Yves Rovert, and Bora Ucar. *hal.inria.fr*. 2015. https://hal.inria.fr/hal-01242327.


Copyright © AllScale Consortium Partners 2017
D2.2 – Requirement Specifications and Reports on External Technological Developments (b)


D2.2 – Requirement Specifications and Reports on External Technological Developments (b)

Laboratory, Sandia National. The QThread Library. 2016.
NOAA. 02 20, 2016.
NVIDIA. CUDA. January 1, 2016.
http://www.pgas.org/.

Copyright © AllScale Consortium Partners 2017
D2.2 – Requirement Specifications and Reports on External Technological Developments (b)


D2.2 – Requirement Specifications and Reports on External Technological Developments (b)


