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

Project Number 671603

D6.7 – AMDADOS Deepwater Horizon application preparation and evaluation (b)

WP6: Integration, testing and pilot applications

Version: 2.0
Author(s): Fearghal O’Donncha (IBM), Albert Akhriev (IBM)
Date: 26/04/18
Due date: PM31
Submission date: 30/04/2018
Project start date: 01/10/2015
Project duration: 36 months
Deliverable lead organization: IBM
Version: 1.0
Status: Final
Author(s): Fearghal O’Donncha (IBM), Albert Akhriev (IBM)
Reviewer(s): Charles Gillan (QUB), Philipp Gschwandtner (UIBK)

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.
D6.7 – AMDADOS application – preparation and evaluation (b)

Acknowledgements

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

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

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

More information

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

Version History

<table>
<thead>
<tr>
<th>Version</th>
<th>Date</th>
<th>Comments, Changes, Status</th>
<th>Authors, contributors, reviewers</th>
</tr>
</thead>
<tbody>
<tr>
<td>0.1</td>
<td>02/03/17</td>
<td>First draft of D6.6</td>
<td>Fearghal O'Donncha, Emanuele Ragnoli</td>
</tr>
<tr>
<td>0.2</td>
<td>08/03/17</td>
<td>Benchmarking performance</td>
<td>Fearghal O'Donncha</td>
</tr>
<tr>
<td>0.3</td>
<td>14/03/17</td>
<td>Herbert Jordan review</td>
<td>Herbert Jordan</td>
</tr>
<tr>
<td>0.4</td>
<td>15/03/17</td>
<td>Arne Hendricks review</td>
<td>Arne Hendricks</td>
</tr>
<tr>
<td>1.0</td>
<td>17/03/17</td>
<td>Final version of D6.6</td>
<td>Fearghal O'Donncha</td>
</tr>
<tr>
<td>1.1</td>
<td>18/04/18</td>
<td>First version of D6.7</td>
<td>Fearghal O'Donncha, Albert Akhriev</td>
</tr>
<tr>
<td>1.2</td>
<td>22/04/18</td>
<td>First draft of D6.7</td>
<td>Fearghal O'Donncha</td>
</tr>
<tr>
<td>1.3</td>
<td>24/04/18</td>
<td>Charles Gillan review</td>
<td>Charles Gillan</td>
</tr>
<tr>
<td>1.4</td>
<td>25/04/18</td>
<td>Philipp Gschwandtner review</td>
<td>Philipp Gschwandtner</td>
</tr>
<tr>
<td>2.0</td>
<td>26/04/18</td>
<td>Final version of D6.7</td>
<td>Fearghal O'Donncha</td>
</tr>
</tbody>
</table>
Table of Contents

More information........................................................................................................... 3
Executive Summary ......................................................................................................... 5
1 Introduction .................................................................................................................... 6
2 Functionality of the pilot applications ......................................................................... 6
   2.1 Governing Equations ............................................................................................... 7
   2.2 Data Assimilation .................................................................................................... 7
   2.3 Adaptive Grid Refinement ....................................................................................... 8
3 Developing pilot application within the AllScale API/SDK ......................................... 9
4 Validation and performance ......................................................................................... 12
   4.1 Developer experience and productivity .................................................................. 14
5 Benchmarks .................................................................................................................. 15
6 Conclusions .................................................................................................................. 17
Bibliography .................................................................................................................... 17

Index of Figures

Figure 1: Example of domain decomposition into (16 x 16) subdomains. Blue points depict location of sensors pseudo-randomly scattered across the domain. Coordinates are defined as indices of nodal points................................................................. 8

Figure 2: Simulation of advection-diffusion process. Top row: Evolution in time of the "correct" solution computed offline as a single global domain. Bottom row: data-assimilation solution that starts from zero density field and gradually catches up the ground-truth........................................................................................................ 13

Figure 3: Relative difference between the ground-truth density and data assimilation solution, as a function of "relative time": = 100t / T, where t is a physical time in seconds, and T is the full integration period........................................... 14

Figure 4: Scalability test results. Number of seconds to run a 3500 second simulation when the number of working threads is varied................................................. 16

Figure 5: Scalability test results. Number of seconds to run a 3500 timestep simulation when the size of the problem (number of subdomains) increases, each subdomain consists of 16x16 finite difference cells............................................ 16

Glossary

API – Application Programming Interface
DA – Data Assimilation
DD – Domain Decomposition
PDE – Partial Differential Equations
Executive Summary

This document is an update to D6.6, which was a deliverable submitted on month 18. This present deliverable describes the incremental preparation, deployment and validation of the AMDADOS pilot application. The document focuses on extensions and developments since deliverable D6.6 to provide a feature complete and scalable AMDADOS pilot application developed using the AllScale API. The application is presented emphasising numerical implementations and challenges to (and opportunities for) parallelism; we present details on porting to the AllScale API and how our experiences can inform future AllScale users. The document concludes by presenting an evaluation of performance and productivity of current implementation using the AllScale API. Major differences between this document and the previous deliverable D6.6 focuses on:

- Porting all core computational structures to the stencil API from the previous implementation (which was primarily based on parallel pfor over spatial domain only). This unlocks the full potential of nested recursive parallelism by providing a space-time decomposition of workload.
- Integrating the application with the adaptive grid capabilities of the AllScale API to 1) maximise the accuracy of data assimilation routines and 2) provide more efficient usage of compute resources.
- Implementation of a ghost zone based subdomain synchronisation scheme to reduce the frequency of neighbour-to-neighbour communication (replacing the alternative Schwarz iterative method).
- Full integration with the AllScale I/O API.
- Development of a dedicated observation generator scheme that integrates directly with the AMDADOS configuration file. This allows for ready generation of user defined observations to allow for quick experimentation across multiple experimental configurations.
- Provides more detail on the algorithmic structure and in particular the localised data assimilation scheme as the authors feel these will be of significant interest to the scientific community and will further the development of scalable data assimilation implementations.
- Provide an application developer’s perspective on code development using the API to better quantify how AllScale may facilitate and promote user productivity.
- The primary focus (and extension relative to D6.6) of this deliverable D6.7 is:
  1. The scalability of the AMDADOS application using the AllScale API
  2. Analysis of the impact of the API on developer productivity.
1 Introduction

The AMDADOS pilot application was described in the month 18 deliverable D6.6. The acronym AMDADOS stands for Adaptive Meshing and Data Assimilation for Dispersion of Oil Spills. Details on the functionality and motivation of the pilot were described in that deliverables and are integrated here with algorithmic modifications made in the intervening period. We have gained, since D6.6 was completed, considerable experience developing, experimenting and benchmarking within the AllScale toolchain. This deliverable aims to communicate the results of the AMDADOS pilot application developed using the AllScale environment. Results emphasise both HPC scaling and benchmarking aspects, together with impact on developer productivity, an aspect which we believe is one of the key AllScale contributions. In the remainder of the deliverable, the AMDADOS pilot application is described; this focuses on algorithmic structures and data assimilation routines together with practical aspects to enable a functional simulation pipeline. Section 3 describes the development of the application using the AllScale User API. Key parallel constructs are compared with an MPI implementation to demonstrate the novel paradigms offered by AllScale. Section 4 presents the simulation results from the application and compares to a serial solution to demonstrate correctness of solution. It also provides an evaluation of developer productivity using the AllScale API and summarises the experience of the authors in developing a complex data assimilation framework within the AllScale boundaries. Section 5 presents experimental benchmarks of the application focusing on parallel scalability. Finally we conclude with a brief summary and outline the future work to be completed in this project (for the AMDADOS pilot application).

2 Functionality of the pilot applications

Feasible and scalable systems for the accurate estimation of advection diffusion processes are required in several applications. Examples include forecasting oil spill evolution for remediation efforts (Guo et al. 2009), quantifying the transport of nutrients around aquaculture installations (O’Donncha et al. 2013) and monitoring releases from industrial operations (Koziy et al. 1998). Typically, these are provided from the solution of a set of Partial Differential Equations (PDEs) on a discretised grid. To improve the accuracy of the prediction, methods exist to update the prediction using measurements of the actual state via data assimilation (DA). DA improves the accuracy of forecasts provided by physical models and evaluates their reliability by optimally combining a priori knowledge encoded in equations of mathematical physics with a posteriori information in the form of sensor data.

Domain Decomposition (DD) is a standard tool in many scientific domains to reduce the complexity or computational cost of solution. Some of the factors which have motivated DD approaches include: 1) the solution of the subproblems is qualitatively or quantitatively easier than the original, 2) the original problem does not fit into the available memory space and 3) the subproblems can be solved with some concurrency (i.e. in parallel).
AMDADOS combines DD approaches to implement scalable DA algorithms by localising to individual subdomains. The global domain is split into subdomains and the equations are discretised on each subdomain. Interface boundary conditions are enforced using a ghost cell approach that overlaps neighbouring solutions. The data assimilation algorithm aligns with the data decomposition strategy by adopting a set of localised data filters unique to each subdomain.

### 2.1 Governing Equations

The situation being modelled is the widely-studied problem, of a domain, Ω, with some initial concentration \(u_{gt}(x, y, 0)\) at location \(p_c\) that is propagated forward in time. For the full treatment of this class of PDE, the reader is referred to Hundsdorfer and Verwer, (2013). The physical model of contaminant being dispersed and transported over a spatial domain is described by the following equation:

\[
\frac{\partial u}{\partial t} = D \left( \frac{\partial^2 u}{\partial x^2} + \frac{\partial^2 u}{\partial y^2} \right) - v_x \frac{\partial u}{\partial x} - v_y \frac{\partial u}{\partial y}, \quad \text{s.t.} \quad u|_{\partial \Omega} = 0.
\]

where \(D\) is diffusion coefficient, \(v_x = v_x(x,y,t)\), \(v_y = v_y(x,y,t)\) are the flow velocity components, the initial condition is defined as point source at some location \((x_c,y_c)\) and the boundary condition is of homogeneous Dirichlet type. Information external to the computational domain is prescribed by these boundary conditions. To discretise on a numerical grid, implicit Euler method is used for its simplicity, unconditional stability and ability to handle stiff problems. Discretisation of the governing equation is straightforward:

\[
\frac{u_{i,j}^{t+1} - u_{i,j}^t}{\Delta t} = D \left( \frac{u_{i+1,j}^{t+1} - 2u_{i,j}^{t+1} + u_{i-1,j}^{t+1}}{\Delta x^2} + \frac{u_{i,j+1}^{t+1} - 2u_{i,j}^{t+1} + u_{i,j-1}^{t+1}}{\Delta y^2} \right) - \frac{v_x}{2\Delta x} u_{i+1,j}^{t+1} - \frac{v_x}{2\Delta x} u_{i-1,j}^{t+1} - \frac{v_y}{2\Delta y} u_{i,j+1}^{t+1} + \frac{v_y}{2\Delta y} u_{i,j-1}^{t+1},
\]

where the discrete indices \((i,j)\) run over a subdomain, \(\Delta t\) and \(\Delta x\), \(\Delta y\) are the time and space discretization steps respectively.

### 2.2 Data Assimilation

To update solution, some sparse information, or ground-truth data is available on the evolution of the constituent concentration over time from sensors distributed within the domain (typically with some associated sensor uncertainty level). Figure 1 presents an example configuration with relatively few sensors randomly distributed across the domain. The sensors are assumed sufficiently accurate but scarce in number.
Since our main objective is to evaluate the capabilities of the Allscale API in real-world applications, the observation data is artificially generated for simplicity. The advantage of having the freedom to generate "ground-truth" is that sparsity can be controlled while it can also be used to assess the accuracy of the data assimilation solution. This data generator is written in Python for simplicity. The data assimilation routine (based on the widely used Kalman filter) produces an estimate of the state of the system as an average of the system's predicted state and of the new measurement using a weighted average. One of the key challenges of the Kalman filter is that it is computationally extremely demanding (Verhaegen and Van Dooren 1986), therefore making it disadvantageous for large scale systems like the one investigated here. Various methods of distributed Kalman filtering have been proposed, but many still suffer from scalability issues or depend on the structure of the problem. The common feature of those methods is that the distribution of filters is done for a discrete model by decomposition of the corresponding matrix, while here, the distribution of filters is done by means of spatial domain decomposition on a continuous level. Moreover, this work does not depend on the filter formalism, but applies a decomposition of the problem and observation allowing an independent choice of the local estimators.

2.3 Adaptive Grid Refinement
To improve the precision of the data assimilation framework and allow for more accurate specification of sensor location, subdomains are processed at different resolution using the multi-scaling capability of Allscale API. Subdomains can be processed at different resolutions, and we simultaneously use subdomains with either 16x16 or 8x8 nodal points. Namely, the subdomains with observations are processed at fine resolution (16x16) because this yields better Kalman filtering.
estimation. On the other hand, domains without observations (where we just integrate the governing equation) are processed at coarser resolution (8x8 cells) with less computational cost. This leverages the adaptive grid refinement structures provided by the AllScale User API to denote computations at different resolution levels and to synchronize interface boundaries between subdomains of different resolutions.

3 Developing pilot application within the AllScale API/SDK

Porting to the AllScale API exploits the domain decomposition paradigm of the application to leverage recursive parallelism. Namely, parallelism is implemented by distributing individual subdomains across cores with synchronization and latency hidden to the user. Contrary to an MPI parallel application, where synchronization must be handled by the user via repeated MPI calls, the AllScale prototype implementation has a much closer feel to a serial application. To demonstrate, this section presents the main parallel constructs of the AMDADOS application implemented using the library of available AllScale API operators. These are compared to an equivalent MPI implementation to elucidate on the increased software productivity enabled by the API. In particular, we address the complexity of parallel constructs, explicit synchronization and other concepts considered within the remit of the computer scientist rather than the natural scientist.

Code listing 1 demonstrates an implementation of a parallel iterator over multiple subdomains within the model. The code segment initialises all subdomain elements to zero. Grid structures are created containing information on the computational structures such as cells, boundary information, position (within global domain), resolution level etc. We initialize this information on an AllScale grid structure (implemented within the AllScale core API at the level of the computer science expert) and loop over a two-dimensional vector \( \text{idx} \) using the \( \text{pfor} \) operator. The \( \text{idx} \) vector contains information on position of each subdomain from \([0,0]\) to \([M,N]\) where \( M \) and \( N \) are number of subdomains in \( x \) and \( y \) respectively. Since the application enables multiple levels of horizontal resolution within each subdomain (i.e. as part of the multi-scaling capabilities), initialisation proceeds for all levels. This demonstrates the fundamental concept of the parallel loop construct within the AllScale API. The pfor operator provides a parallel loop execution over a range applying a user defined function to each iterator.
Parallel DD based solvers are based on the paradigm of distributing the problem across compute cores, solving each state independently and synchronizing solution at intervals (generally each timestep). Listing 2 presents the implementation of a space-time decomposition of the problem that provides parallel constructs over the time dimension (T) and the two-dimensional spatial dimension, represented by each subdomain. Within this stencil template, an update operation (in this case a function to compute, or propagate forward, local subdomain solution) is applied to each element of the array (here each element is reference index to each subdomain). For each update, the user-defined update operation is combining the previous value of the solution within a locally confined area surrounding the targeted subdomain to obtain the updated value. Since each subdomain solution are independent for a single timestep and depend only on direct neighbours for solution synchronization, this provides a valuable resource for parallelism within a space-time decomposition.

Listing 1: Sample code to initialize concentrations on all subdomains to zero by means of a parallel loop construct using the pfor operator

```cpp
Listing 1: Sample code to initialize concentrations on all subdomains to zero by means of a parallel loop construct using the pfor operator.

// Initialize the model state variables for all subdomains.
pFor(Point(0,0), Point(M,N), [&](const Point_2D & idx) {
    // Iterate through all available grid resolutions.
    for (int layer = LayerFine; layer <= LayerLow; ++layer) {
        state[idx].setActiveLayer(layer);
        // Set all cells within subdomain at that resolution to zero
        state[idx].forAllActiveCells([](double & v) { v = 0.0; });
        // Implement knowledge related to external boundary conditions
        ApplyBoundaryCondition(state, idx);
    }
});
```

Listing 2: AllScale Stencil parallel computation

```cpp
Listing 2: AllScale Stencil parallel computation.

Exchange of boundary information to maintain solution fidelity is central to DD approaches. Within the AllScale API, synchronisation aspects are managed at the core API level facilitating trivial implementation (at least from the perspective of the application developer) of boundary exchange operations. Listing 3 outlines how we code boundary exchanges. Neighbouring domains (if they exist) are identified via Boolean data types. On each of the four boundaries, the overlapping local boundary are replaced by the computed values from the neighbouring, remote boundary. All additional synchronization considerations such as send/receive orderings, computational overlapping, etc. are managed at the level of the core API, hidden from the application developer. Further, despite
```
the stencil implementation providing a complete space-time decomposition, the API ensures that data from the appropriate time level are communicated.

Listing 3: AllScale boundary exchange implementation

```cpp
// for each subdomain update boundaries in each direction
for (Direction dir : { Up, Down, Left, Right }) {
    // obtain the local boundary
    auto local_boundary = state.getBoundary(dir);

    // obtain the neighboring boundary
    auto remote_boundary =
        (dir == Up) ? state[idx + utils::Coordinate<2>{-1, 0}].getBoundary(Down):
        (dir == Down) ? state[idx + utils::Coordinate<2>{ 1, 0}].getBoundary(Up):
        (dir == Left) ? state[idx + utils::Coordinate<2>{0,-1}].getBoundary(Right):
            state[idx + utils::Coordinate<2>{0, 1}].getBoundary(Left);

    // compute updated boundary
    assert(local_boundary.size() == remote_boundary.size());
    local_boundary = remote_boundary;
    state.setBoundary(dir, local_boundary);
}
```

As a comparison, Listing 4 presents a section of code demonstrating how boundary exchange in an MPI implementation of the code may look. Within this paradigm, the application loops over each boundary (where neighbour domain exists), packs boundary data and sends to its neighbour via an MPI_Send communication. The neighbouring domain must receive the data via a corresponding MPI_Recv call. Further care must be taken in the order of MPI_Send/MPI_Recv calls to avoid blocking by a process awaiting communication. The simplified coding structure of the AllScale API removes many of these manual synchronization requirements greatly reducing coding complexity.

Apparent is the greatly simplified coding implementations enabled by the AllScale API. Applying operations to all elements is similar to a serial application with a parallel for (the `pfor`) operator substituting the classical for operator. Other common parallel operations such as data exchange between separate grids are also greatly simplified. By following the templates provided by the AllScale SDK it is evident that users can develop a huge range of domain decomposition based applications with little knowledge of HPC or parallel computing concepts (i.e. simply by learning AllScale specific development skills).
D6.7 – AMDADOS application – preparation and evaluation (b)

Listing 4: Sample MPI implementation of boundary exchange and synchronization within a DD problem

```c
// For each subdomain, loop over boundaries and check
// whether boundaries exist in each direction and synchronise
for (size_t ind = 0; ind < 4; ind++)
{
    Connection * conn = decProb->connections[ind];
    // for each subdomain boundary check if
    // neighbour exist to East or West
    if (conn->lhs == subProblem || conn->rhs == subProblem)
    {
        // get connection nodes index and subproblem
        if (conn->rhs == subProblem) // neighbour to the West
        {
            SchursPackSendMPI(decProb, subProblem, conn, timestep, bcType, -1, proc_id);
        }
        if (conn->lhs == subProblem) // neighbour to the East
        {
            // Receive boundary data from East
            SchursRecvUnpackMPI(decProb, subProblem, conn, timestep, bcType, -1, proc_id);
            // Send boundary data to East
            SchursPackSendMPI(decProb, subProblem, conn, timestep, bcType, -1, proc_id);
        }
        if (conn->rhs == subProblem) // neighbour to the West
        {
            // Receive boundary data from West
            SchursRecvUnpackMPI(decProb, subProblem, conn, timestep, bcType, -1, proc_id);
        }
        if (conn->up == subProblem || conn->down == subProblem)
        {
            ... // For neighbour subdomains to the South and North
            ... // repeat equivalent exchange of boundary data
            ... // and synchronisation as implemented for East and West above
        }
    }
}
```

4 Validation and performance

Domain decomposition-based approaches have huge applicability in simulation due to the promise of reduced computational demand (by distributing across compute resources, reducing the size of linear algebra matrices, etc.). An important consideration however, is to ensure fidelity of the solution; i.e. the computed solution should be qualitatively (if not quantitatively) equivalent to that of a single, serial global domain. This is particularly important for implementations such as the AllScale API where many parallel constructs are managed at the kernel level, outside the remit of the application developer. Implementations using a novel technology such as AllScale, at a relatively low level of maturity, requires careful analysis of results to ensure fidelity. The current implementation provides a valuable benchmark of correctness as the observation generator used to provide data for the assimilation scheme also
serves as the true solution. Hence, computed results can be readily compared against these data. Figure 2 presents snapshots of results from a number of stages during the simulation cycle. Results are compared to the "correct" solution computed as part of the observation generator routine (i.e. routine that generates data for assimilation into the model).

![Figure 2: Simulation of advection-diffusion process. Top row: Evolution in time of the "correct" solution computed offline as a single global domain. Bottom row: data-assimilation solution that starts from zero density field and gradually catches up the ground-truth density.](image)

Figure 3 shows how relative error fades away as simulation progressing. The relative error is computed as a ratio between $L_2$-norm of the field of density difference and $L_2$-norm of the ground-truth density (provided by the Python observation generator script):

$$\text{Error} = \| u_{gt} - u \| / \| u_{gt} \|.$$

The data-assimilation solver behaves as expected in nudging the solution towards the correct solution. It catches up the true distribution as soon as the first sensor has been touched by a high-density spot. Of importance, no contamination of results develops from boundary exchange protocols; i.e. no aliasing is evident at subdomain boundaries in Figure 2 while Figure 3 demonstrates that data assimilation directs error towards zero over time.
4.1 Developer experience and productivity

A key result of this study that is more difficult to quantify is developer productivity. Even for a relatively small HPC study, ease of programming is greatly improved. At the simplest level, aspects related to synchronisation and message passing are removed from the developer’s responsibility. Moving to a higher level of abstraction, aspects related to the hardware architecture are managed by the AllScale environment, meaning that the domain expert is not exposed to complex topics related to the tuning of the application for various architectures: these are handled by the computer scientist at the data structure level, again removed from any domain specific algorithmic implementations. Finally, since data structures are self-contained representations of each subdomain solution at the core API level, aspects related to load balancing are removed from the developer’s view, eliminating complex and code-intensive aspects such as overlapping computation and communication and other cumbersome load balancing approaches. Instead, load balancing can be managed at higher level of abstraction using advanced online monitoring and management complex.

A key consideration to enable full exploitation of AllScale capabilities is the development of proficient programming skills within the AllScale API. The AllScale user API is exposed to the user in the form of template-based C++ libraries. These provide powerful capabilities to write generic programs and exploit parallelism at higher levels of abstraction. For many domain experts, this may involve an educational step to get familiar with expressing their applications in this form. Additionally, the user needs a firm understanding of the different parallel structures provided by AllScale (pfor, stencil, etc.) and how parallelism is exploited. To promote these skills, educational material,
documentation and tutorials are a key requirement to effectively disseminate and develop the AllScale toolchain. The challenge that this presents is evident when one compares against highly established and mature market incumbents. As an example, MPI is almost 30 years old, the user manual extends to 868 pages and countless tutorials exist from a rich variety of sources. To enable AllScale challenge the status quo, it is imperative that appropriate educational material is provided to enable users develop effective applications.

At a more practical level, nested recursive parallelism facilitated by AllScale discourages any global synchronizations. This constraint mean that significant algorithmic changes may be required to develop efficient code within the AllScale environment. Further, external libraries are not easily integrated within the toolchain (Iakymchuk et. Al. 2016). As an example, the MPI version of the AMDADOS pilot application used the armadillo library to concisely express matrix operations. Moving to the AllScale API required that necessary linear algebra functions (matrix inversion, multiplication, etc.) be manually developed as part of the AMDADOS application.

5 Benchmarks
Performance results are a key metric of this analysis. Experiments focus on shared memory parallelism to provide insights into throughput, contention and parallel performance at different levels of load allocation. Tests were conducted on compute server with 2 Intel Xeon 2.20GHz processors providing total of 44-core/88-thread machine empowered by Linux RedHat-7.4, 64-bit operating system. The first test investigates strong scaling performance to gain insight into both algorithmic and AllScale scalability. A key consideration of all tests was how the application scales within the grid-based implementation, i.e. to understand what the relative proportions of computation, communication and management of the compute workloads and distribution are. Figure 4 presents the strong scaling results. The problem size is fixed and the number of working threads increased from 1 to 44 in a series of simulations. In general results exhibit acceptable performance. Increasing the number of CPU cores significantly reduces the simulation time before plateauing as parallel overheads saturate.
Figure 4: Scalability test results. Number of seconds to run a 3500 second simulation when the number of working threads is varied.

The second test investigates the total computational throughput of the application. We enabled all the CPU resources (88-thread) to the application increasing the problem size (number of subdomains) in a series of simulations. Figure 5 presents results of this experimental study. For this experiment ideal scaling would be a linear increase in simulation time as the problem size is increased. For moderate problem size, this trend is respected with simulation time increasing approximately as linear function of problem size. However, at large problem size we observe a huge degradation in parallel performance. Increasing the problem size from 784 to 1024 subdomains increases the simulation time by 156% despite the problem size increasing by only 30%.

Figure 5: Scalability test results. Number of seconds to run a 3500 timestep simulation when the size of the problem (number of subdomains) increases, each subdomain consists of 16x16 finite difference cells.
Analysis and profiling suggests this to be a result of parallel overheads. Monitoring CPU usage during the simulation demonstrates huge variation in usage over the course of the simulation. In particular, there are periods where usage drops quite drastically suggesting that the volume of work being distributed is not always sufficient to avoid idle threads. Analysis of simulation statistics using the Oprofile Linux profiling tool demonstrates that the ratio of time spent executing Runtime operations increases significantly at large problem size. Experimental configurations of 400 subdomains result in 6% of time being spent in parallel overheads; this number increases to 35% when the problem size is increased to 1024. Work is ongoing to better understand the issue underpinning this and to achieve high scalability and throughput of the AMDADOS application.

6 Conclusions
This document demonstrates and quantifies the capabilities of the AllScale API implementation of the AMDADOS pilot application. It serves as an evaluation of the toolchain and its feasibility as part of the next generation of HPC programming environments. Developing within the AllScale user API provides many advantages to the scientist. User productivity is enhanced as parallel structures are hidden at the core level of the API. This advantage is particularly true for users writing new parallel code from scratch as one simply follows the provided template and write the code in a manner very similar to serial code. All programming is done in pure C++ eliminating the need to learn any specific parallel tools.

Future work is focused on extending this study to distributed memory experiments. In particular, large scale experiments will be conducted to evaluate 1) scalability of the code at the many 1000 core level, 2) performance of load balancing (data assimilation provides a valuable test-case due to the very different computational expense of subdomains depending on whether observation exists or not) and 3) detailed evaluation of the performance of the recursive parallel space-time decomposition.

Bibliography


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).

D2.1 – Requirement Specifications and Reports on External Technological Developments (a). Innsbruck: AllScale Project, 2016, 64.

https://doi.org/10.1016/j.aquaeng.2012.07.006