AllScale toolchain pilot applications: PDE based solvers within a parallel development environment


Published in:
Computer Physics Communications

Document Version:
Peer reviewed version

Queen's University Belfast - Research Portal:
Link to publication record in Queen's University Belfast Research Portal

Publisher rights
Copyright 2019 Elsevier B.V.
This manuscript is distributed under a Creative Commons Attribution-NonCommercial-NoDerivs License (https://creativecommons.org/licenses/by-nc-nd/4.0/), which permits distribution and reproduction for non-commercial purposes, provided the author and source are cited.

General rights
Copyright for the publications made accessible via the Queen's University Belfast Research Portal is retained by the author(s) and / or other copyright owners and it is a condition of accessing these publications that users recognise and abide by the legal requirements associated with these rights.

Take down policy
The Research Portal is Queen's institutional repository that provides access to Queen's research output. Every effort has been made to ensure that content in the Research Portal does not infringe any person's rights, or applicable UK laws. If you discover content in the Research Portal that you believe breaches copyright or violates any law, please contact openaccess@qub.ac.uk.

Download date: 18. Mar. 2022
AllScale toolchain pilot applications: PDE based solvers using a parallel development environment

Fearghal O’Donncha\textsuperscript{a,*}, Roman Iakymchuk\textsuperscript{b}, Albert Akhriev\textsuperscript{a}, Philipp Gschwandtner\textsuperscript{c}, Peter Thoman\textsuperscript{c}, Thomas Heller\textsuperscript{d}, Xavier Aguilar\textsuperscript{b}, Kiril Dichev\textsuperscript{e}, Charles Gillan\textsuperscript{e}, Stefano Markidis\textsuperscript{b}, Erwin Laure\textsuperscript{b}, Emanuele Ragnoli\textsuperscript{a}, Vassilis Vassiliadis\textsuperscript{a}, Michael Johnston\textsuperscript{a}, Herbert Jordan\textsuperscript{c}, Thomas Fahringer\textsuperscript{c}

\textsuperscript{a}IBM Research – Ireland
\textsuperscript{b}KTH Royal Institute of Technology, 100 44 Stockholm, Sweden
\textsuperscript{c}University of Innsbruck, 6020 Innsbruck, Austria
\textsuperscript{d}University of Erlangen-Nuremberg, 91058 Erlangen, Germany
\textsuperscript{e}Queen’s University of Belfast, Belfast BT7 1NN, United Kingdom

Abstract

AllScale is a programming environment targeting simplified development of highly scalable parallel applications by dividing development responsibilities into silos. The front-end AllScale API provides a simple C++ development environment through a suite of parallel constructs expressions denoting tasks operating concurrently. This interfaces with the other components of the toolchain (core-level API, compiler and runtime) which manages tasks related to the machine and system level, hidden to the user. The paper describes the development of two large-scale parallel applications within the AllScale API, namely, an advection-diffusion model with data assimilation

*Corresponding author

Email addresses: feardonn@ie.ibm.com (Fearghal O’Donncha), riakymch@kth.se (Roman Iakymchuk), albert_akhriev@ie.ibm.com (Albert Akhriev), philipp.gschwandtner@uibk.ac.at (Philipp Gschwandtner), peter.thoman@uibk.ac.at (Peter Thoman), thomas.heller@fau.de (Thomas Heller), xaguilar@pdc.kth.se (Xavier Aguilar), k.dichev@qub.ac.uk (Kiril Dichev), c.gillian@qub.ac.uk (Charles Gillan), markidis@kth.se (Stefano Markidis), erwin1@kth.se (Erwin Laure), eragnoli@ie.ibm.com (Emanuele Ragnoli), vassilis.vassiliadis@ibm.com (Vassilis Vassiliadis), michaelj@ie.ibm.com (Michael Johnston), herbert.jordan@uibk.ac.at (Herbert Jordan), tf@dps.uibk.ac.at (Thomas Fahringer)

Preprint submitted to Computer Physics Communications November 20, 2019
and a Lagrangian space-weather simulation model based on a particle-in-cell method. We present mathematical formulations and implementations and evaluate parallel constructs developed using the AllScale API. The performance of the applications from the perspective of both parallel scalability, and more importantly productivity are assessed. We demonstrate how the AllScale API can greatly improve developer productivity while maintaining parallel performance in two applications with distinct numerical characteristics. Code complexity metrics demonstrate reduction in application specific implementations of up to 30% while performance tests on three different compute systems demonstrate comparable parallel scalability to an MPI version of the code.

Keywords: HPC, data assimilation, partial differential equation, numerical solvers, advection-diffusion, particle-in-cell.

1. Introduction

The computational and data requirements of modern simulation tools for applications such as weather, computational biology and fluid dynamics are a fundamental driving force for modern HPC systems. To meet these demands, current systems consist of thousands of nodes, each with dozens of cores, complex memory hierarchies, and often equipped with accelerators. In order to fully exploit these systems, applications typically require bespoke parallelisation schemes to achieve efficient performance across different compute systems and architectures.

Development of effective code requires consideration of multiple parallelisation schemes, load balancing, synchronisation of solution, together with effective discretisation of the numerical equations. These factors place very high skill demands on the application scientist. The developer needs knowledge of sophisticated domain related algorithmic formulation, together with advanced software engineering skills and understanding of system architecture. These skill demands are accentuated as the degree of parallelism increases and application codes are deployed on hundreds of thousands to millions of computational cores. Developing programming models that effectively exploit these systems typically requires a hybrid parallelisation scheme that separately addresses inter- and intra-node parallelism.

A large number of programming models exist to support parallel developments, particularly in shared memory. The Cilk language allows
task-focused parallel programming and is an early example of efficient task scheduling via work stealing, while OpenMP [2], provides an efficient programming interface for loop and task based parallelism. In addition to languages and extensions, industry-standard and well-supported parallel libraries based on task parallelism have emerged, such as Intel Cilk Plus [3] or Intel TBB [4]. These task-based programming environments aim to exploit the concurrent execution of tasks within a program, and have proven widely applicable for consumer applications. However, they have little applicability for HPC solutions on large-scale systems.

Considering distributed systems, message-passing is the most widely used means of describing programs to run on multiple nodes [5]. Some task-based parallel libraries for C++ exist, such as HPX [6] and Charm++ [7]. These combine tasks with a global address space programming model to distribute tasks across multiple processes (or computer addresses). A comprehensive assessment of task-based parallel programming technologies for high-performance computing is provided in [8].

Several research projects aim to address the challenge of developing effective numerical solvers for partial differential equations (PDEs) on large-scale systems by separating domain science and computational science aspects. This ‘separation of responsibilities’ envisions the domain scientist working at an API layer (or similar), while the computer scientist or HPC expert manages performance and machine level optimisations at a deeper level. A particular advocate of this principle in application development is the Firedrake framework [9], which aims for an automated system for the portable solution of partial differential equations (PDE) using the finite element method (FEM). It builds on the Unified Framework Language (UFL) [10] employed by the FEniCS project [11] to provide an API that enables scientists to express PDEs in a high-productivity interpreted language. The PyOP2 framework [12] provides an abstraction between the domain scientist concerned with implementing the numerical methods for solution of PDEs numerics and the implementation of parallel execution over multi-core platforms.

In this paper we present the AllScale development toolchain [www.allscale.eu], which aims to provide computational paradigms to simplify parallel programming on large-scale compute systems. A key component of these future systems is parallelism of the order of $10^5-10^6$ cores. This degree of parallelism requires novel algorithmic structures to improve efficiency together with decoupling of the specification of parallelism from the associated management activities during program execution thereby improving produc-
tivity and the development environment. The goal of AllScale is that the domain scientist can develop the algorithmic components largely sheltered from considerations around computational performance, and management of parallelisation constructs. AllScale addresses this objective by providing a complete parallel toolchain (context-aware compiler and runtime system with dynamic optimisation) which is exposed to the developer through a standard, template-based C++ API. The design of this ExaScale development environment is based on three key principles:

1. Enabling the separation of responsibilities in the development of HPC applications;
2. Utilising industry standard programming languages and preserving compatibility with existing development and debugging tools;
3. Employing advanced programming language, compilation and runtime system technology to transparently integrate sophisticated services into parallel applications.

From the perspective of the application developer, the AllScale toolchain promises highly increased productivity by hiding parallel constructs and providing a development API reminiscent of serial applications. Similarly, the HPC expert can focus on developing and optimising generic parallel constructs and operators without any intrusion from algorithmic components.

This paper provides a comprehensive evaluation of the AllScale toolchain from the perspective of productivity (i.e. whether it simplifies the development process) and performance (i.e. parallel scalability on large systems). The two pilot applications consist of an advection-diffusion based model with data assimilation (DA) and a Lagrangian particle-in-cell (PIC) method for space-weather simulations.

The paper is structured as follows: the next section details the AllScale programming environment and motivations for its use. The AllScale programming environment aims to separate

2. AllScale Toolchain

This section outlines the AllScale programming environment and motivations for its use. The AllScale programming environment aims to separate
responsibilities between domain scientists, HPC experts and system level experts by offering a well-defined bridge between their worlds. The bridge provided by the AllScale API consists of two parts that represent the basic building blocks of every parallel algorithm:

- parallel control flow primitives;
- data structures.

The former are defined via a single, recursively parallel, higher-order operator \((\text{prec})\) \cite{13}, whereas the latter fulfil the concept of a data item. The \textit{prec} operator is used by the application developer to denote tasks to be computed in parallel and it enables the AllScale compiler \cite{1} to infer the context of a parallel task creation point. The compiler then creates multiple versions (sequential, shared and distributed memory parallel) of the code, from which the runtime system \cite{2} may select depending on the system state. This further means that the user needs to provide and maintain only a single implementation of the code.

Both building blocks are part of the AllScale Core API and follow the open/close principle of software engineering, by being open for extension but closed for modification. This technique allows any high-level operators and data structures needed by domain scientists (e.g. parallel loops, stencils, structured and unstructured meshes) to be implemented by HPC experts in the extensible AllScale User API. This layered API design is illustrated in Figure \cite{1}.

Domain scientists can use the AllScale User API \textit{without} requiring knowledge of the design of scalable operators, low-level data management and other aspects related to parallelism and synchronisation control code that would obstruct an otherwise clear implementation of a high-level algorithm. HPC experts likewise are relieved of the need for domain-specific knowledge or low-level optimisations but can focus on the development of efficient parallel operators offering domain-decomposition thereby reducing overall development overheads. In addition, system level experts are not required to support any high-level components but only their common base in the Core API, greatly reducing maintenance, optimisation and tuning effort. Finally, as the

\cite{1} \url{https://github.com/allscale/allscale_compiler}
\cite{2} \url{https://github.com/allscale/allscale_runtime}
AllScale API is implemented as an embedded DSL [14] in pure C++14, compatibility with existing compilers, debuggers, and many other toolchain tools required during the development process is preserved.

The ease-of-use and direct expression of parallel semantics at a high level enabled by the AllScale User API is illustrated by two short code examples, shown in Listing 1 and 2. The former implements fine-grained dependencies between subsequent parallel loops, while the latter provides an example of how to use a built-in data structure. This Grid fulfils the requirements on a data item, and therefore the AllScale toolchain is aware of its semantics and capable of generating the required code to distribute it on a cluster. As a result of this design, domain scientists only need to consider the AllScale user-level parallelism primitives and data structures, and are shielded from the underlying complexity of programming a large-scale distributed memory system.
Listing 1: Two `pfor` operators initializing and incrementing data in a std::array with fine-grained synchronization. The second `pfor` will execute iteration `i` after the first has finished its iterations `i−1`, `i`, and `i+1`. Constructs specific to the AllScale API are shown in orange and underlined.

```cpp
#include <array>
#include <allscale/api/user/algorithm/pfor.h>
namespace alg = allscale::api::user::algorithm;
using ArrayType = std::array<int,N>;
const int N = 200;
void initAndIncrement(const ArrayType& data, ArrayType& output) {
    auto ref = alg::pfor(0,N,[&](int i) {
        output[i] = ...; // initialization
    });
    alg::pfor(1,N-1,[&](int i) {
        output[i] += data[i+1] + data[i] + data[i-1];
    }, alg::neighborhood_sync(ref));
}
```

Listing 2: Example usage of the `Grid` data structure.

```cpp
#include <allscale/api/user/data/grid.h>
// create a two-dimensional grid of integers of size 10x20
allscale::api::user::data::Grid<int, 2> grid({10,20});
// initialize all elements with 1.0
grid.pforEach([](int& element) { element = 1.0; });
// set element at position [7,9] to 5.0
grid[{7,9}] = 5.0;
```

The parallelism exposed via the parallel primitives of the AllScale API is controlled by the *AllScale Runtime System* – an extension of HPX, an established task-based runtime system [15]. AllScale’s application runtime model [16] is based on tasks, represented by calls to the `prec` operator. The conversion of `prec` calls to runtime-compatible entities is done by the *AllScale Compiler*. While it provides additional features, their discussion is omitted for brevity as they are not explicitly used in this work. Each runtime task can be sent to a so-called worker for processing or be split into two smaller tasks, which in turn can be processed in parallel or be split again. This recursive nesting of parallelism enables automatic control over the degree of parallelism at runtime without any additional manual effort. It is the foundation for sophisticated runtime system features such as automatic load balancing and provides a clear advantage over application models where developers are tasked with manually implementing such features. In addition, the *AllScale Runtime System* includes a monitoring component for real-time performance feedback. This information is used by the *AllScale Runtime System* in its
task scheduling process. Furthermore, the collected performance data can be pipelined to an external server (to visualise several metrics in real-time), thereby, giving application and system developers real-time inspection of software performance and resource utilisation. Some real-time metrics include timing for tasks, task throughput, power, memory usage, CPU load, idle rate and bytes sent and received through the network. The *AllScale Monitoring Component* also can provide post-mortem reports, i.e., logs, plots, and heat maps on simulations.

Developers implement code using the *AllScale API*, including its generic library of parallel algorithms (e.g. Listing 1) and data structures (e.g. Listing 2). The *AllScale Compiler* turns applications into binaries that can be effectively managed and tuned by the *AllScale Runtime* system to obtain efficient and resilient execution on a large variety of medium to large scale parallel computer systems. At its core, the AllScale programming model facilitates a separation of responsibilities in application development, an aspect that distinguishes it from other state of the art programming models. In these other models, application developers are required to bear in mind a wide range of considerations related to the execution environment (e.g. architecture, available cores, memory access). When that execution environment includes a very large number of compute cores, then developers need to monitor hardware resources in order to perform load balancing and optimise application performance. Within AllScale, the developer works at the API level, while aspects related to load balancing, performance or machine-specific optimisations are handled at the *AllScale Runtime* level – separating the domain scientist from the HPC expert. Further, the automatic management of tasks at the runtime level enables resilience against node failures, known as hard faults [17].

To address hard faults, a generic implementation of checkpoint restart is available within the *AllScale Toolchain*. In the presence of hard faults, the AllScale resilience component of the AllScale runtime restarts tasks and performs checkpoint/restarts with the support of the scheduler.

3. Methodology

This paper focuses on the development, performance and scalability of two pilot applications within the AllScale toolchain. Aspects related to the development of the code within the user API are assessed while parallel performance using the AllScale runtime system (based on the HPX paral-
lel standard library \cite{15}) is compared against benchmark MPI simulations. The case studies consider two applications representative of computationally intensive numerical solvers – data assimilation (DA), and space weather.

DA is a central technique in many geoscientific modelling and forecasting systems to optimally combine system physics and sensor measurements. DA improves the accuracy of forecasts provided by physical models and evaluates their reliability by optimally combining \textit{a priori} knowledge encoded in equations of mathematical physics with \textit{a posteriori} information in the form of sensor data. The situation being studied reduces to an inverse problem, where one uses sensor observations to infer the set of parameters or causal factors that produced them. Prediction, or the forward model, then proceeds from this updated state. DA has been applied across a large number of geoscientific domains including meteorology \cite{18}, oceanography \cite{19}, hydrology \cite{20} and ecology processes \cite{21}. A comprehensive review of recent developments in data assimilation for the study of ocean processes and events is provided in \cite{22}. In this study we deploy an advection-diffusion model with data assimilation to simulate dispersion in surface waters; the model used is the Adaptive Meshing and Data Assimilation for Dispersion of Ocean Spills (AMDADOS) code.

Space weather \cite{23} is the study of processes originating in the sun and propagating through the solar system. It imparts numerous effects on people and technology in space and on earth ranging from auroras in the polar regions to electromagnetic disturbances disrupting currents in power and communication infrastructure. The shape of the Earth’s magnetosphere is determined by the microscopic interaction phenomena between the solar wind and the dipolar magnetic field of the planet. To describe these interactions correctly, we need to model phenomena occurring over a large range of time and spatial scales. In fact, the magnetosphere comprises regions with different particle densities and magnetic field intensities. One of the most widely used methods for space weather simulations is the PIC method \cite{24,25}. In the PIC model, computational particles represent the plasma particles from the solar wind. 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. This study adopts the widely used iPIC3D model.
3.1. **AMDADOS**

The AMDADOS model simulates conservative tracer transport in surface flows. It resolves the simulation of transport within a domain, $\Omega$, with some initial concentration $u_{gt}(x, y, 0)$ at location $p_c$ and time $t = 0$ that is propagated forward in time. Some sparse information, or ground-truth data is available on the constituent concentration evolution over time from sensors distributed within the domain (typically with some associated sensor uncertainty level). The data assimilation problem for this case can be formulated as follows: find a reasonably good approximation to the distribution of contaminant in the domain as a function of space and time given only a physical model and sparse observations.

The physical model of transport over a spatial domain is described by the following equation [26]:

$$\frac{\partial u}{\partial t} = D \left( \frac{\partial^2 u}{\partial x^2} + \frac{\partial^2 u}{\partial y^2} \right) - v_x \frac{\partial u}{\partial x} - v_y \frac{\partial u}{\partial y},$$

s.t. $u|_{t=0} = \delta(x-x_c, y-y_c)$, $u|_{\partial \Omega} = 0$.

where $D$ is the diffusion coefficient, $v_x = v_x(x, y, t)$ and $v_y = v_y(x, y, t)$ are the flow (current) velocity components and the initial condition is defined as a point source at some location $(x_c, y_c)$. Information external to the computational domain is specified by boundary conditions. Ideally, the absorbing boundary condition should be applied at the outer border $\partial \Omega$ of the domain $\Omega$. In our case, a high density value is mostly obtained far from the boundary and we can apply a Dirichlet condition [27]. The numerical solver used is the implicit (or backward) Euler method; it is used for its unconditional stability and ability to handle stiff problems [28, 29]. Akhriev et. al. [30] presents more details on the computational setup and initialisation for the numerical problem.

The DA scheme employed is the Kalman filter. The fundamental goal of data assimilation methods is to integrate available observation data with a dynamical model using an assimilation scheme. Since the data contains errors and models are imperfect representations, the assimilation scheme needs to consider confidence in both observations and model during the update phase. The 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.
In this scheme the analysis in the assimilation cycle is computed by the update equation \[31\]:

\[
x^a = \hat{x} + K(x^o - H\hat{x}) \tag{2}
\]

where \(x^a\) are the \textit{a posteriori} state estimate (or the updated solution), \(\hat{x}\) are the modelled data and \(x^o\) are the observed data. \(H\) is an operator that maps the forecasted data vectors into the observation space and \(K\) represents the Kalman gain, which can be written as:

\[
K = \frac{PH^T}{HPH^T + R} \tag{3}
\]

where \(P\) and \(R\) are the State Error Covariance Matrix and the Observation Error Covariance Matrix, respectively. We see from equation \[3\] that as the measurement error covariance \(R\) approaches zero, the gain weights the residual, \((x^o - H\hat{x})\), more heavily guiding the model towards the measured state. On the other hand, as the \textit{a priori} estimate of error covariance \(P\) approaches zero, the gain \(K\) weights the residual less heavily.

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. In this study, the global domain, \(\Omega\), is decomposed into a set of smaller sub-domains, which are distributed across computational cores. Each subdomain is implemented as a grid of nodal cells. Within each subdomain, the filtering of model and observations, is implemented and at the end of each iteration, neighbouring subdomain solutions are synchronized \[32\]. At run time, each subdomain is assigned to a \textit{worker}, either an execution thread or a process, in case of the distributed application. The assignment and workload balancing is done automatically once the grid of subdomains have been exposed to parallel AllScale operators.

3.2. iPIC3D

The iPIC3D (implicit Particle-In-Cell 3D) pilot application \[33\] simulates the interaction between solar wind and the Earth’s magnetic field. The underlying PIC method \[24\] is one of the most common and powerful numerical techniques for the simulation of fusion, astrophysics, and space plasmas. For instance, PIC simulations \[34\] are used to study the interaction of the Earth’s
electromagnetic field with hot plasma emanated by the sun, the so-called solar wind. The Earth’s magnetosphere is a large system with many complex physical processes, requiring realistic domain sizes and billions of computational particles. In the PIC model, computational particles represent the plasma particles from the solar wind. 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. Since the high-energy plasma in space can damage spacecraft and endanger the life of astronauts in space, it is important to enable efficient large-scale PIC simulations capable of predicting phenomena in space.

In kinetic simulations of plasmas, the evolution of the distribution functions $f$ for a given species (electrons, protons or heavy ion species) is calculated by solving numerically the transport equation without the collisional term, the so-called Vlasov equation [24, 25]:

$$\frac{\partial f}{\partial t} + v \cdot \frac{\partial f}{\partial r} + \frac{q}{m} \left( E + \frac{v \times B}{c} \right) \cdot \frac{\partial f}{\partial v} = 0,$$

where $q$ and $m$ are the charge and mass of the species, respectively; $v$ is the velocity dimension; $r$ is the space; and $B$ and $E$ are the magnetic and electric fields, accordingly. The Vlasov equation is solved in combination with Maxwell’s equations:

$$\frac{\partial B}{\partial t} = -c \nabla \times E,$$
$$\frac{\partial E}{\partial t} = c \nabla \times B - 4\pi J.$$

The coupling of the Vlasov equation and the Maxwell’s equations is provided by the charge, $\rho$, and current, $J$, densities that are the moments of the distribution function $f$

$$\rho = \sum q \int f dv,$$
$$J = \sum q \int v f dv.$$

One of the most successful approaches to solve the Vlasov-Maxwell system is the PIC method. In the PIC method, the original distribution function, $f$, is described by means of computational particles: particle positions and
velocities are randomly sampled according to the initial given distribution function. At every computational cycle, particle positions and velocities can be updated, solving numerically the equation of motion for each particle:

\[
\begin{align*}
\frac{dx}{dt} &= v, \\
\frac{dv}{dt} &= \frac{q}{m}(E + v \times B).
\end{align*}
\]

At each computational step, it is possible to reconstruct the distribution function using the computational particle positions and velocities.

In general, the workflow of iPIC3D can be summarized in two steps: 1) electric $E$ and magnetic $B$ fields as well as the velocity of the particles $v$ and position $x$ are initialized on the grid using the set-up defined in the input file; 2) the Maxwell equation and the equation of motion are calculated simultaneously on the grid for several cycles. The number of cycles to run the simulation is also specified in the input file.

Typically, parallel iPIC3D simulations divide the simulation box into several domains that are equal in size. Each domain is assigned to a process that carries out the computation for the particles in the domain. When a particle exits the domain, the code transfer the data describing the particle to another domain. Because of the non-uniform configuration of the electromagnetic field in space, computational particles concentrate in relatively small spatial regions, while the density of particles in other regions is substantially lower. The distribution results in having more particles in certain simulation domains than others and results in the work-imbalance problem: processes with fewer particles wait for other processes with more particles to finish their computations at every time step.

iPIC3D represents a challenging HPC application as 1) the computation of particle motion is expensive and 2) large variation in particle distribution among cells leads to large load imbalances. Many PIC implementations try to use explicit solvers such as leapfrog approximations or the explicit Tajima’s scheme \[35\] to solve the equation of motion. In this work, we employ a second order scheme, called the Boris mover \[36\]. The Boris scheme suits much better task-based parallelism for many-core computation due to its local nature. Additionally, in case of the PIC simulations, there is no direct interaction among the particles, which makes computations on each particle non-overlapping. Note that AllScale automatically manages work distribution and load balancing through its parallel constructs like $pfor$. 

13
4. Results

4.1. Experimental procedure

The experimental set-up considered three factors: 1) the computed solution is correct and appropriate algorithmic sophistication is supported, 2) the impact of the AllScale API on developer productivity, code complexity and maintainability, and 3) parallel scalability.

Domain decomposition based approaches are ubiquitous in simulation due to the promise of reduced computational demand (by distributing across compute resources, reducing the size of 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 computed if modelled as a single global domain). For both applications we analysed the generated solution against analytical or benchmark values.

Developer productivity considered the ease of development and code complexity of the application. Porting to the AllScale API exploited the domain or data decomposition paradigm of the applications to leverage recursive parallelism. For AMDADOS, parallelism was implemented by distributing individual subdomains across cores, while iPIC3D essentially used a set of parallel loops iterating over all cells in the 3D-space grid. Such parallel instructions are directly mapped to a recursive formulation of the \texttt{pfor} or \texttt{stencil} operator provided by the AllScale API. For both pilot applications, synchronization and latency remain hidden to the user. Contrary to an MPI parallel application, where synchronization must be handled by the user via repeated MPI calls, the AllScale implementation has a much closer feel to a serial application.

Parallel scalability experiments were conducted on a number of different infrastructures, namely:

- Cray XC40 system at the PDC Center for High Performance Computing in KTH (“Beskow” system);
- Megware manufactured system based on Intel Xeon E5-2630v4 processors at the Friedrich-Alexander University (FAU) (“Meggie” system);
- Intel Xeon based system at TU Wien (“VSC-3” system).

The Beskow system is a Cray XC40 system, based on Intel Xeon E5-2698v3 Haswell and Intel Xeon E5-2695v4 Broadwell processors, and Cray
Aries interconnect technology. It has 2 Intel processors per node, for a total of 32 cores in the Haswell nodes, and 38 cores in the Broadwell nodes. Haswell nodes have 64 GB of memory per node, and Broadwell nodes 128 GB. The Meggie cluster is a high-performance compute resource with high speed interconnect. Each node contains two Intel Xeon E5-2630v4 "Broadwell" chips running at 2.2 GHz connected by an Intel OmniPath interconnect with up to 100 GBit/s bandwidth. The VSC-3 system provides 2020 nodes, each equipped with 2 processors (Intel Xeon E5-2650v2, 2.6 GHz, 8 cores from the Ivy Bridge-EP family), 64 GB of main memory and internally connected with an Intel QDR-80 dual-link high-speed InfiniBand fabric.

4.2. Correctness of solution

Experimental tests evaluated the capabilities of the pilot applications to accurately resolve representative test cases. Namely, the ability of the AMDADOS pilot application to accurately reconstruct a contaminant spread given initial conditions, user defined flow field and sparse observation data across the domain, and a comparison of the iPIC3D numerical solution outputs against an analytical solution.

![Relative difference](image)

Figure 2: Relative difference ($\varepsilon = \|u_{gt} - u\|_2/\|u_{gt}\|_2$) between the ground-truth density and data-assimilation solution, as a function of “relative time”: $\tau = 100t/T$, where $t$ is a physical time in seconds, and $T$ is an integration period.

Domain decomposition based approaches invoke parallelism by distributing subdomains across compute resources and synchronising the solution periodically. To provide a benchmark of correctness for the AMDADOS ap-
plication, we first run the simulation as a single, global domain, from which we extract observations for the data assimilation scheme. This also serves as the true solution against which the computed result from the distributed model can be readily compared. The objective then reduces to evaluating the ability of the parallel data assimilation scheme to accurately reconstruct the benchmark solution.

Figure 2 shows how relative error fades away for the AMDADOS application as simulation progresses. Starting from time \( t = 0 \), the evolution of the simulated field closely tracks the correct solution, which is exposed to the model in terms of assimilated sensor data. The relative error is computed as a ratio between \( L_2 \)-norm of difference between computed and benchmark density, and \( L_2 \)-norm of ground-truth density: \( \varepsilon = \|u_{gt} - u\|_2/\|u_{gt}\|_2 \). The data-assimilation solver ‘nudges’ the solution towards the correct solution, catching up with the true distribution when sufficient sensor information on the true state is ingested directing error towards zero over time.

![Figure 2](image)

Figure 3: Results obtained for the \( v \times B \) rotation: the analytical result is a closed circular orbit at the Larmor radius, which is shown by the solid blue line; the numerical results computed using the AllScale Toolchain, which is shown by the square points.

Figure 3 compares the computed numerical results of iPIC3D’s particle mover against an analytical solution and demonstrates a correct representation of system dynamics for relatively small problems such as the \( v \times B \) (or \( v \) cross \( B \)) rotation. The \( v \times B \) rotation is a known example used for correctness as it tests particles rotation about the field line in the absence of the electric field \( E \) in (5)-(6). For medium size simulations, we compare particles density (number of particles) per cell, recorded with certain intervals (e.g. every 10 time steps) for both the parallel runs against the sequential execution. For real world simulations, we record the final aggregated results of
the simulation – such as the total electric and magnetic energy as well as the
total kinetic energy – and analyze them according to the energy preserving
law, permitting a small deviation due to round-off errors.

4.3. Developer productivity

The AllScale environment is directed at improving developer productivity
by separating algorithmic (domain science) and HPC aspects. By providing
a clean separation, it aims to ease the development of numerical solvers while
decreasing maintenance effort, and allowing the independent optimisation of
system-level components without changing application code.

A cumbersome development task in many-core applications are implementa-
tion of parallel constructs required to synchronise solution across cores
and nodes. Within the AllScale API, synchronisation aspects are managed at
the core API level facilitating trivial implementation of boundary exchange
operations. Listing 3 schematically outlines how boundary exchange were im-
plemented for the AMDADOS application. Neighbouring domains (if they
exist) were identified via Boolean data types. On each of the four boundaries,
the overlapping local boundary were updated by the computed values from
the neighbouring, remote boundary. All additional synchronization consider-
ations, such as send/receive orderings, computational overlapping, and load
balancing, were managed at the level of the core API and runtime and hidden
from the application developer. Further, by separating parallel aspects from
the application development, shared or distributed memory parallel simulations
via simple command line instructions specifying locality and threads
agnostic of architecture are allowed. Code complexity is greatly reduced
while making transition between different architectures seamless.
Listing 3: AllScale boundary exchange implementation

```cpp
// Declare a grid structure: number of sub-domains in all dimensions,
// size of sub-domains (at different resolutions, if desired) and the
// type of sub-domain structure (e.g. a matrix of floating-point values).
// The infrastructure will be maintained by AllScale API implicitly.
...

// for each subdomain update boundaries in each direction
pfor(Point(0,0), Point(M,N), [&](const Point_2D & idx) {
    // init A with current state
    A = state[idx]
    // update boundaries
    for(Direction dir :{Up,Down,Left,Right}){
        // obtain the local boundary
        auto local_boundary = A[idx].getBoundary(dir);
        // obtain the neighboring boundary
        auto remote_boundary =
            (dir == Up ) ? A[idx+{-1,0}].getBoundary(Down ) :
            (dir == Down) ? A[idx+{1,0}].getBoundary(Up ) :
            (dir == Left) ? A[idx+{0,-1}].getBoundary(Right) :
            (dir ==Right) ? A[idx+{0,1}].getBoundary(Left ) ;
        // compute updated boundary
        assert(local_boundary.size() == remote_boundary.size());
        local_boundary = remote_boundary;
        state.setBoundary(dir,local_boundary);
    });
```

A comparable MPI implementation involved significantly more complex
code, comprising explicit formulations for sub-domain initialisation, and data
exchange. Grid initialisation involves an explicit call to MPI initialisation
functions together with a mapping of process identity to enable appropriate
communication between pertinent subdomains (i.e. neighbour domains in
each direction). Listing 4 gives a schematic of an MPI implementation of
data exchange and synchronisation using message passing. Adopting a sim-
ilar numerical grid discretisation as for AllScale, explicit calls to send and
receive data are invoked. These involve packing of data to be communicated,
and calls to non-blocking MPI send functions; these are complemented with
calls to MPI receive and unpacking of variables at local boundary. For illus-
tration purposes, we include an outline of the function to send data, while
the corresponding receive function is very similar in terms of code complexity
(not presented). In total, the send and receive implementation involved ap-
proximately 70 lines of code, consisting of tedious, error-prone bookkeeping
of neighbours, direction of data send/receive, and data to be communicated.
The interested reader may refer to our GitHub repository containing the
AllScale toolchain and both pilot applications presented here (both AllScale
and MPI implementations): [https://github.com/allscale/](https://github.com/allscale/)
Listing 4: Sample MPI implementation of boundary synchronisation

// Initialise grid and create sub-domains attached to this process.

// Main time integration loop
for (long timestamp = 0; timestamp < Nt; ++timestamp) {
    ...  
    // Call Function to exchange boundary
    // values between domains using MPI

    grid forallLocal([&](SubDomain * sd) {
        sd->SendBoundariesToNeighbours(grid, timestamp);
    });
    // Corresponding function to receive data.
    // Here we diligently collect boundary values from the peer
    // sub-domains (4 neighbours in 2D case) using MPI_Recv()
    // function. About 40 lines of code.
    grid forallLocal([&](SubDomain * sd) {
        sd->ReceiveBoundariesFromNeighbours(timestamp);
    });
    // ....... Continue subdomain computation
    // ....... within the time loop

    // Define function to send boundary data to pertinent neighbour
    virtual void SendBoundariesToNeighbours(const MpiGrid & grid, long timestamp) {
        // Get boundaries from state field outer elements
        // and place boundary values in buffer for communicating
        for (int x = 0; x < Sx; ++x) {
            m_send_boundary[Down][x] = state(x, 1); // Data to be
            m_send_boundary[Up][x] = state(x, Sy); // communicated N/S
        }
        for (int y = 0; y < Sy; ++y) {
            m_send_boundary[Left][y] = state(1, y); // Data to be
            m_send_boundary[Right][y] = state(Sx, y); // communicated E/W
        }
        // Send boundaries to all neighbour subdomains in non-blocking
        // fashion. Note that, in the final implementation we optimised by
        // not sending data on the same node but copying directly
        m_send_count = 0;
        for (int b = 0; b < NSides; ++b) { // iterate on all four sides
            const Neighbour & nei = m_neighbour[b]; // neighbour domain
            MPI_CHECK(MPI_Isend(static_cast<void*>(m_send_boundary[b].data()),
                           static_cast<int>(m_send_boundary[b].size()),
                           MPI_DOUBLE, nei.rank, nei.tag,
                           MPI_COMM_WORLD,
                           &(m_send_request[m_send_count]));
            ++m_send_count;
        }
    }

    These schematic descriptions of data exchange in AllScale and MPI pro-
vide a qualitative illustration of the simplified exchange process enabled by the API. To better quantify the potential to streamline application code, we evaluated the code complexity of both applications against the standard MPI implementation. We used the open source tool CMetrics to measure several widely used code complexity metrics on the code bases for the AllScale and MPI versions of the pilots.

The four metrics considered were: 1) source lines of code (SLOC – without spaces/comments); 2) Halstead’s Mental Discriminations (H MEN D); 3) McCabe’s average cyclomatic complexity across all modules (AVG CY); and 4) the sum total cyclomatic complexity across the entire code base (TOT CY). Halstead’s Mental Discriminations are a complexity metric based on the number of uniquely identified operators and operations in the

Figure 4: Code metric comparisons for AllScale and MPI implementation of pilot applications. The y-axis represents each metric normalised to the AllScale implementation (i.e. AllScale metric = 1.0 for each metric)
given program, while McCabe’s metric captures complexity by considering the number of linearly independent paths through the program’s control flow graph. Figure 4 compares each metric normalized to its AllScale result.

Despite enabling a larger feature set – the MPI version did not support dynamic load balancing, monitoring interface or resilience support – the AllScale implementation contains lower application code complexity in all metrics. The largest relative difference is observed in the average cyclomatic complexity comparison for iPIC3D producing a reduction of 30% compared to the benchmark. This is primarily due to required MPI context management having a significant impact on this metric.

A further and more subjective, estimate considered the complexity of code necessitated by parallel constructs, namely, statements in user code which can be attributed directly to introducing parallelism or to managing parallel data. For AllScale, this would include calls to parallel operators, and for MPI it would include all ”MPI.*” calls. Analysis demonstrates that for both pilot applications, there were between 2.5 and 3.5 times more parallel constructs in the MPI version than that developed using the AllScale API.

4.4. Parallel scalability

A fundamental objective of the AllScale environment is parallel scalability. Figure 5 presents scaling results running the AMDADOS pilot application using the AllScale API on three different systems, namely Beskow, Meggie and VSC-3 cluster described in section 4.1. Computational performance is evaluated in terms of the total throughput (here defined as the number of subdomains computed) as compute resources increase (in a weak scaling implementation), and compared to the benchmark MPI implementation.

For the AMDADOS pilot application, AllScale gives excellent performance on two of the three systems evaluated. The AllScale implementation gives up to 2x speedup compared to the manually tuned, MPI version. For both VSC-3 and Meggie cluster, throughput is similar, up to approximately 16 nodes. Beyond this, the AllScale version significantly outperforms MPI, leading to a performance speedup of 2.67x and 1.92x on the Meggie and VSC-3, cluster respectively. Profiling of the application code demonstrates this to result from the significant communication overhead in the pilot applications. The recursive parallelism of the AllScale system that overlaps communication and computations significantly improves on the flat profile of the MPI implementation.
Figure 5: AllScale scalability results compared to an MPI implementation for the AMDA-
DOS pilot application on three different compute systems, namely Beskow, Meggie and
VSC-3 (see Section 4.1). Throughput on the y-axis denotes the number of subdomains
processed per second while the x-axis denotes number of cores. Experiments were con-
ducted on up to 256 nodes for both Beskow and Meggie system, and up to 64 nodes for
the VSC-3 system.

Figure 6 presents weak scaling performance obtained for the iPIC3D pi-
lot application using the AllScale environment on the same infrastructure.
Computational throughput is defined in terms of the number of particles com-
puted per second as a function of the number of nodes, following the weak
scaling paradigm. The AllScale implementation of iPIC3D shows comparable
performance against its MPI counterpart on Meggie and VSC-3 computing
infrastructure. For small node count, the AllScale version slightly outper-
forms MPI, however, on larger node counts, MPI provides higher throughput,
particularly on the Beskow system.

On the Beskow system, MPI significantly (by 1.7x and 1.99x for AMDA-
DOS and iPIC3D, respectively) outperforms AllScale. This results from two
cumulative factors:

• For the Beskow Cray system, our implementation does not fully utilise
the networking capabilities, resulting in parallel degradation;

• The highly optimised Cray MPI implementation reduces the overheads
of communication in MPI, thereby degrading some of the performance
Figure 6: AllScale scalability results compared to an MPI implementation for the *iPIC3D* pilot application on three different compute systems, namely Beskow, Meggie and VSC-3 (see Section 4.1). Throughput on the y-axis denotes the number of particles computed per second while the x-axis denotes number of cores. Experiments were conducted on up to 256 nodes for both Beskow and Meggie system, and up to 64 nodes for the VSC-3 system.

advantage observed on the other systems.

In practical applications, one is often concerned with the turnaround time to solve a problem of fixed size. Figure 7 presents strong scaling results for both pilot applications when running up to 128 nodes and compares to the MPI implementation. For both applications, performance tends to plateau after approximately 500 cores as communication overheads manifest, mirrored for both MPI and AllScale implementations. For AMDADOS, the AllScale version slightly outperforms MPI until about 256 cores; profiling indicated that beyond this, the nested parallelism and task splitting implemented by AllScale did not provide efficient performance at the problem-size-per-node being computed. The *iPIC3D* application exhibits near-identical performance for both AllScale and MPI on low number of cores, beyond which the MPI version performs noticeably better. Of note is the relatively small problem size that can be computed (restricted by the memory of a single node) for a memory-intensive applications such as *iPIC3D*. As for the AMDADOS pilot application, the problem size limits the value of task splitting and distribution to idle threads, favouring the more direct compute and distribution model of MPI.
Figure 7: Strong scaling results compared to an MPI implementation for the AMDADOS and iPIC3D pilot application on the VSC-3 system. The x-axis represents the number of cores (simulations were conducted up to 128 nodes with each node containing 16 cores), and the y-axis represents the simulation time to compute 14,400 subdomains and $5\times10^{-10}$ particles for AMDADOS and iPIC3D respectively.

These results reflect the challenges of introducing new programming environments to compete with well-established, mature products. Results demonstrate that by using the AllScale environment, applications can be easily ported to different architectures and executed with no manual interventions. However, to achieve full performance, further tuning is necessary on the Beskow system to achieve parity with MPI, while the task-splitting and distribution overhead may prove onerous in communication-bound scenarios as observed for iPIC3D in Figure 7.

5. Conclusions

This study demonstrates the capabilities of the AllScale toolchain and its feasibility as part of the next generation of HPC programming environments. Application development using the AllScale API provides many advantages to the scientist. User productivity is greatly enhanced as parallel structures are hidden at the core level of the API. All programming is done in pure C++, eliminating the need to learn any specific parallel tools and avoiding the MPI+X burden of having different parallel languages for different archi-
tectures. Code maintainability is improved by separating computer science and domain science aspects, while the architecture agnostic design eliminates any need for multiple code bases. A number of additional features such as dynamic load-balancing, monitoring interface and hard fault resilience are automatically provided to the user.

We present the development, porting and execution of two application codes, namely a data assimilation framework leveraging localised filtering and domain decomposition (AMDADOS), and a particle in cell code for simulation of space-weather (iPIC3D). Using real-world applications, we evaluate code complexity and parallel simulation constructs qualitatively and quantitatively to evaluate the potential benefits provided by the AllScale environment for developer productivity. Parallel synchronization aspects are greatly simplified resulting in reduction in code complexity of up to 30% compared to the MPI code.

Parallel scalability demonstrates the potential impact for HPC applications. The AllScale implementation achieves comparable performance to manually-tuned versions using industry-standard MPI parallel libraries. The AllScale version of the AMDADOS application significantly outperforms the MPI version, possibly due to the greater control provided by the fine-grained stencil implementation for recursive parallelism in space-time. On the Beskow system, performance degradation largely results from inefficient usage of networking implementation imposing a significant penalty on parallel scalability. To promote further experimentation and scientific replication, the AllScale environment and pilot applications are publicly available (https://github.com/allscale). Full details on installation of the environment and integration of pilot applications is provided in [40], while a Python script is provided in the AllScale GitHub repository to run and collate all scalability experiments presented in this paper.

This paper presents parallel scalability on systems up to 8,192 cores using X86 type architectures. To promote uptake amongst modern HPC computing applications, work is ongoing to understand and tune application performance on larger systems up to 50,000 cores. Furthermore, modern HPC systems generally encompass heterogeneous architectures combining CPU and GPU cores to maximise performance. Research is ongoing to to provide support for CPU and GPU integration using the AllScale API
Acknowledgements

This project has received funding from the European Union’s Horizon 2020 research and innovation programme as part of the FETHPC AllScale project under grant agreement No 671603.

References


URL http://dl.acm.org/citation.cfm?doid=2676870.2676883


URL http://www.allscale.eu/docs/D5.7ResilienceManager.pdf


