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

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

[a] *IBM Research – Ireland*
[b] *KTH Royal Institute of Technology, 100 44 Stockholm, Sweden*
[c] *University of Innsbruck, 6020 Innsbruck,Austria*
[d] *University of Erlangen-Nurnberg, 91058 Erlangen, Germany*
[e] *Queens University of Belfast, Belfast BT7 1NN, United Kingdom*

Abstract

AllScale is a programming environment targeting highly scalable parallel applications, simplifying application development in the ExaScale era by siloing development responsibilities. The front-end AllScale API provides a simple C++ development environment and a suite of parallel constructs denoting tasks to be operated concurrently. Lower level tasks related to the machine and system level are managed by the AllScale toolchain at the core level. We present the development of two large-scale parallel applications within the AllScale API, namely, an advection-diffusion model with data assimilation and a Lagrangian space-weather simulation model based on a particle-in-cell

*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.gillan@qub.ac.uk (Charles Gillan), markidis@kth.se (Stefano Markidis), erwinl@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)

method. Mathematical formulations and implementations are presented and we 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 distinct applications. 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 computational fluid dynamics (CFD) are a fundamental driving force for modern HPC systems. Consequently, such systems consist of thousands of nodes, dozens of cores and complex memory hierarchies and optionally-equipped with accelerators. In order to fully exploit these systems applications typically require complex parallelisation schemes to address the characteristics exhibited by coarse-grained (inter-node) and by fine-grained (intra-node) parallelism.

Synchronization requirements, combined with multiple parallelisation schemes enforced by performance considerations require development of efficient code, placing very high skill demands on the application developer. 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.

Several research projects aim to address the challenge of developing effective code on modern large-scale systems. A particular advocate of this 'separation of concerns' in application development is the firedrake framework [1], 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) [2] employed by the FEniCS project [3] to provide an API that enables scientists to express PDEs in a high-productivity interpreted language. The PyOP2 framework [4] 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 AllScale (www.allscale.eu), a project funded by the European Commission in its Frontiers and Emerging Technologies in Horizon 2020 (FETHPC) Programme which aims to provide computational paradigms to tackle extreme-scale ExaScale computing (10^{18} Flops). 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 productivity and the development environment. These factors impose significant challenges for developers aiming to efficiently utilise all available resources. In particular, the development of such applications is a complex and labour intensive task requiring management of parallel control flows, data dependencies and underlying hardware resources, each of which embodies challenging problems of its own.

In this paper we present the parallelisation of two PDE-based applications using the novel AllScale toolchain that empowers development of highly-scalable parallel applications. 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. Utilizing 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.

The 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 particle-in-cell (PIC) method for space-weather simulations.

DA is a central technique in many ocean and 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 *a priori* knowledge encoded in equations of mathematical physics with *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 [5], oceanography [6], hydrology [7] and ecology processes [8]. A comprehensive review of recent developments in data assimilation for the study of ocean processes and events is provided in [9].

Space weather [10] is the study of processes originating in the sun and propagating through the solar system, with 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, 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 [11, 12]. In the PIC model, plasma particles from 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. In this work, we study the formation of the Earth's magnetic dipole by the PIC implementation within the AllScale environment.

The paper is structured as follows: the next section details the AllScale development environment and API; Section 3 describes the pilot applications and their implementation within the AllScale environment; Section 4 outlines the impact of the API on development productivity, assesses the quality of numerical solvers adopted and presents experimental scalability results; and

finally, the conclusions section presents outcomes from the study and future research steps.

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 (*prec*) [13], whereas the latter fulfil the concept of a *data item*. 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*. Domain scientists can use the AllScale User API 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 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 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 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 application developers are tasked with manually implementing such features per application. 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 utilization. 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 and data structures. 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 concerns 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. 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 correspondingly to implement resilience against node failures, known as hard faults. Our research identified that dealing with these faults, as opposed to software failures, was the key concern for all of the applications.

To address hard faults, a generic implementation of checkpoint restart is available within the *AllScale Toolchain* and the efficiency of the technique

demonstrated using a particle in cell simulation code. 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 the two pilot applications within the AllScale toolchain. Aspects related to the development of the code within the user API are assessed while parallel performance within the AllScale runtime system (based on the HPX parallel standard library [15]) is compared against benchmark MPI simulations.

3.1. AMDADOS

The AMDADOS (Adaptive Meshing and Data Assimilation for Dispersion of Ocean Spills) model simulates conservative tracer transport in surface flows. It resolves the simulation of transport within a domain, Ω, with some initial concentration $\mathrm{u_{gt}(x, y, 0)}$ at location $\mathrm{p_c}$ 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 [17]:

$$\begin{aligned} \frac{\partial u}{\partial t} =& D\left(\frac{\partial^2 u}{\partial x^2} + \frac{\partial^2 u}{\partial y^2}\right) - v_x \frac{\partial u}{\partial x} - v_y \frac{\partial u}{\partial y}, \\ & \text{s.t.} \quad u|_{t=0} = \delta(x - x_c, y - y_c), \quad u|_{\partial\Omega} = 0. \end{aligned} \tag{1}$$

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 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 Ω. In our case, a high density value is mostly obtained far from the boundary and we can apply a Dirichlet condition [18]. The numerical solver used is the

implicit (or backward) Euler method; it is used for its unconditional stability and ability to handle stiff problems [19, 20].

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 [21]:

$$\mathbf{x^a} = \hat{\mathbf{x}} + \mathbf{K}(\mathbf{x}^\circ - \mathbf{H}\hat{\mathbf{x}}) \tag{2}$$

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

$$\mathbf{K} = \frac{\mathbf{P}\mathbf{H}^T}{\mathbf{H}\,\mathbf{P}\,\mathbf{H}^T + \mathbf{R}} \tag{3}$$

where $\mathbf{P}$ and $\mathbf{R}$ are the State Error Covariance Matrix and the Observation Error Covariance Matrix, respectively. We see from equation 3 that as the measurement error covariance $\mathbf{R}$ approaches zero, the gain weights the residual, $(\mathbf{x}^\circ - \mathbf{H}\hat{\mathbf{x}})$, more heavily guiding the model towards the measured state. On the other hand, as the *a priori* estimate of error covariance $\mathbf{P}$ approaches zero, the gain $\mathbf{K}$ 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, Ω, 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. At run time, each subdomain is assigned to a *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 [22] simulates the interaction between solar wind and the Earth magnetic field. The underlying PIC method [11] is one of the most common and powerful numerical techniques for the simulation of fusion, astrophysics, and space plasmas. For instance, PIC simulations [23] 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, 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. Since the high-energy plasma in space can damage spacecrafts 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 distributions function 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 [11, 12]:

$$\frac{\partial f}{\partial t} + v \cdot \frac{\partial f}{\partial r} + \frac{q}{m}\left(E + \frac{v \times B}{c}\right) \cdot \frac{\partial f}{\partial v} = 0, \tag{4}$$

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:

$$\frac{\partial B}{\partial t} = -c\nabla \times E, \tag{5}$$

$$\frac{\partial E}{\partial t} = c\nabla \times B - 4\pi J. \tag{6}$$

The coupling of the Vlasov equation and the Maxwell's equations is provided by the charge, ρ, and current, J, densities that are the moments of the

distribution function f

$$\begin{aligned} \rho &= \sum q \int f dv, \\ J &= \sum q \int v f dv. \end{aligned}$$

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:

$$\frac{dx}{dt} = v, \tag{7}$$

$$\frac{dv}{dt} = \frac{q}{m}(E + v \times B). \tag{8}$$

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 particles velocity 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, it is communicated to a different domain. Because of the non-uniform configuration of the electromagnetic field in space, computational particles concentrate in relatively small spatial regions, while few particles cover other spatial regions. 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 [24] to solve the equation of motion. In this work, we employ a second order scheme, called the Boris mover [25].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 are no direct interaction among particles, which makes computations on each particle non-overlapping. Note that AllScale automatically manages work distribution and load balancing thought its parallel constructs like *pfor*.

3.3. Experimental procedure

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

Domain decomposition based approaches are applicable 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). To provide a benchmark of correctness, we first run the simulation as a single global domain from which we extract 'observations' for the data assimilation scheme. This simulation also serves as the true solution against which the computed result from the distributed model can be readily compared.

Developer productivity considered the ease of development and maintainability 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 *pfor* or *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 bandwith. 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. Results

4.1. Correctness of solution

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

Figure 1 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 flatten field of density difference and L_2-norm of flatten 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

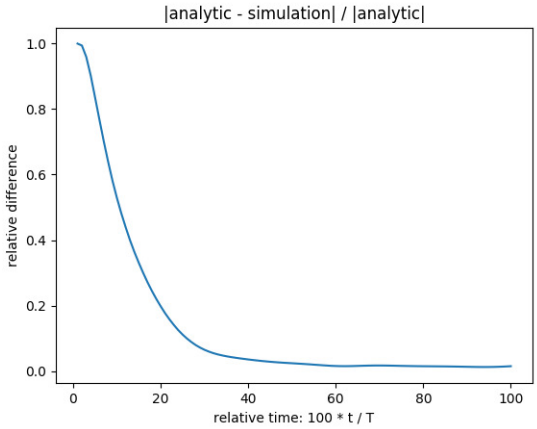

Figure 1: Relative difference ($\varepsilon = \|u_{gt} - u\|_2 / \|u_{gt}\|_2$) between the ground-truth density and data-assimilation solution, as a function of "relative time": $\tau = 100\, t/T$, where t is a physical time in seconds, and T is an integration period.

the true distribution when sufficient sensor information on the true state is ingested directing error towards zero over time.

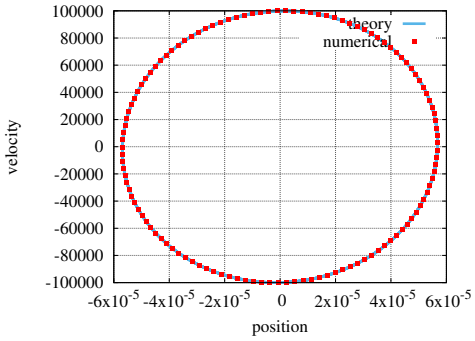

Figure 2: Results obtained for the $v \times B$ rotation: the analytical result is a closed circular orbit at the Larmor radius, which is shown by the solid blue line; the numerical results computed using the AllScale Toolchain, which is shown by the square points.

Figure 2 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.

4.2. Developer productivity

The AllScale environment is directed at improving developer productivity by separating algorithmic (domain science) aspects from 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 is parallel constructs required to synchronize 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 1 schematically outlines how boundary exchange were implemented for the AMDADOS application. Neighbouring domains (if they exist) are identified via Boolean data types. On each of the four boundaries, the overlapping local boundary is updated by the computed values from the neighbouring, remote boundary. All additional synchronization considerations, such as send/receive orderings, computational overlapping, load balancing etc., are 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 is allowed. Code maintainability is greatly improved while making transition between different architectures seamless.

Listing 1: AllScale boundary exchange implementation

```
// 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);
});
```

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 [26] 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) [27]; 3) McCabe's average cyclomatic complexity [28] across all modules (AVG CY); and 4) the sum total cyclomatic complexity across the entire code base (TOT CY). Figure 3 compares each metric normalized to its AllScale result.

Despite enabling a larger feature set – none of the MPI versions compared here support dynamic load balancing, monitoring interface or resilience support – the AllScale version 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 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

Figure 3: 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)

in the MPI version than that developed using the AllScale API.

4.3. Parallel scalability

A fundamental objective of the AllScale environment is parallel scalability. Figure 4 presents scaling performance running the AMDADOS pilot application using the AllScale API on three different systems, namely Beskow, Meggie and VSC-3 cluster described in section 3.3. 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.

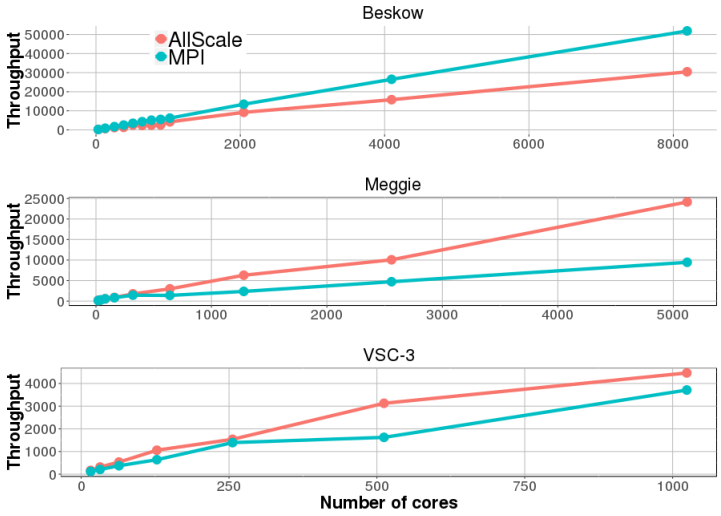

Figure 4: AllScale scalability results compared to an MPI implementation for the AMDADOS pilot application on three different compute systems, namely Beskow, Meggie and VSC-3 (see Section 3.3. Throughput on the y-axis denotes the number of subdomains processed per second

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

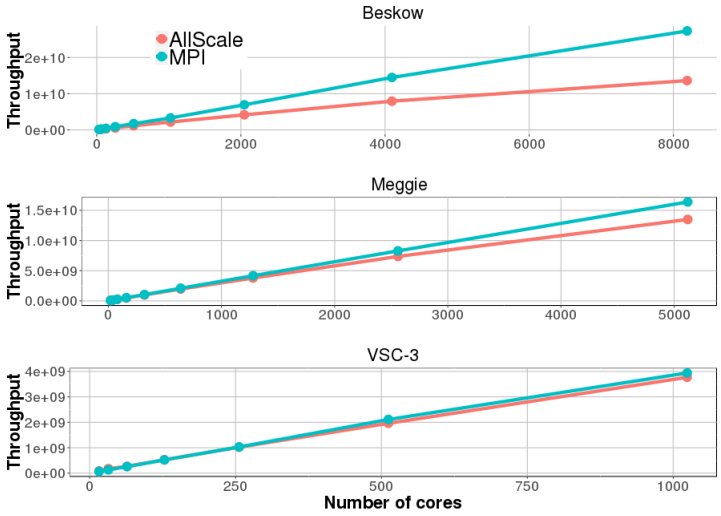

Figure 5: AllScale scalability results compared to an MPI implementation for the *iPIC3D* pilot application on three different compute systems, namely Beskow, Meggie and VSC-3 (see Section 3.3. Throughput on the y-axis denotes the number of particles computed per second

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

- For the Beskow Cray system, our implementation does not fully utilize the networking capabilities, resulting in parallel degradation;
- The highly optimised Cray MPI implementation significantly reduces the overheads of communication in MPI, thereby degrading some of the performance advantage observed on the other systems.

This reflects 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.

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 architectures. 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 [29], 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

[1] F. Rathgeber, D. A. Ham, L. Mitchell, M. Lange, F. Luporini, A. T. T. Mcrae, G.-T. Bercea, G. R. Markall, P. H. J. Kelly, Firedrake, ACM Transactions on Mathematical Software 43 (3) (2016) 1–27. doi:10.1145/2998441.
URL http://dl.acm.org/citation.cfm?doid=2988516.2998441

[2] M. Alnæs, A. Logg, K. Ølgaard, M. Rognes, G. Wells, Unified form language: A domain-specific language for weak formulations of partial differential equations, ACM Transactions on Mathematical Software 40 (2).
URL https://dl.acm.org/citation.cfm?id=2566630

[3] A. Logg, K.-A. Mardal, G. Wells (Eds.), Automated Solution of Differential Equations by the Finite Element Method, Vol. 84 of Lecture Notes in Computational Science and Engineering, Springer Berlin Heidelberg, Berlin, Heidelberg, 2012. doi:10.1007/978-3-642-23099-8.
URL http://link.springer.com/10.1007/978-3-642-23099-8

[4] F. Rathgeber, G. R. Markall, L. Mitchell, N. Loriant, D. A. Ham, C. Bertolli, P. H. Kelly, PyOP2: A High-Level Framework for Performance-Portable Simulations on Unstructured Meshes, in: 2012 SC Companion: High Performance Computing, Networking Storage and Analysis, IEEE, 2012, pp. 1116–1123. doi:10.1109/SC.Companion.2012.134.
URL http://ieeexplore.ieee.org/document/6495916/

[5] P. L. Houtekamer, H. L. Mitchell, P. L. Houtekamer, H. L. Mitchell, A Sequential Ensemble Kalman Filter for Atmospheric Data Assimilation, Monthly Weather Review 129 (1) (2001) 123–137. doi:10.1175/1520-0493(2001)129<0123:ASEKFF>2.0.CO;2.
URL http://journals.ametsoc.org/doi/abs/10.1175/1520-0493{%}282001{%}29129{%}3C0123{%}3AASEKFF{%}3E2.0.CO{%}3B2

[6] J. A. Carton, B. S. Giese, J. A. Carton, B. S. Giese, A Reanalysis of Ocean Climate Using Simple Ocean Data Assimilation (SODA), Monthly Weather Review 136 (8) (2008) 2999–3017. doi:10.1175/2007MWR1978.1.
URL http://journals.ametsoc.org/doi/abs/10.1175/2007MWR1978.1

[7] Y. Liu, H. V. Gupta, Uncertainty in hydrologic modeling: Toward an integrated data assimilation framework, Water Resources Research 43 (7). doi:10.1029/2006WR005756.
URL http://doi.wiley.com/10.1029/2006WR005756

[8] M. Williams, P. A. Schwarz, B. E. Law, J. Irvine, M. R. Kurpius, An improved analysis of forest carbon dynamics using data assimilation, Global Change Biology 11 (1) (2005) 89–105. doi:10.1111/j.1365-2486.2004.00891.x.
URL http://doi.wiley.com/10.1111/j.1365-2486.2004.00891.x

[9] C. A. Edwards, A. M. Moore, I. Hoteit, B. D. Cornuelle, Regional Ocean Data Assimilation, Annual Review of Marine Science 7 (1) (2015) 21–42. doi:10.1146/annurev-marine-010814-015821.
URL http://www.annualreviews.org/doi/10.1146/annurev-marine-010814-015821

[10] G. Lapenta, Particle simulations of space weather, J. Comput. Phys. 231 (2012) 795821.

[11] C. Birdsall, A. Langdon, A. Langdon, Plasma Physics via Computer Simulation, Plasma Physics Via Computer Simulations, 2004. doi:10.1201/b16827.
URL https://www.taylorfrancis.com/books/9781482263060

[12] Y. Grigoryev, V. Vshivkov, M. Fedoruk, Numerical Particle-In-Cell Methods: Theory and Applications, VSP BV, AH Zeist, 2005.

[13] H. Jordan, P. Thoman, P. Zangerl, T. Heller, T. Fahringer, A Context-Aware Primitive for Nested Recursive Parallelism, in: Euro-Par 2016: Parallel Processing Workshops, Springer, Cham, 2017, pp. 149–161. doi:10.1007/978-3-319-58943-5_12.
URL http://link.springer.com/10.1007/978-3-319-58943-5{_}12

[14] P. Zangerl, H. Jordan, P. T. . I. . . . , U. 2018, Exploring the Semantic Gap in Compiling Embedded DSLs, in: 2018 International Conference on Embedded Computer Systems: Architectures, Modeling, and Simulation (SAMOS)., 2018.
URL https://doi.org/10.5281/zenodo.1309475

[15] H. Kaiser, T. Heller, B. Adelstein-Lelbach, A. Serio, D. Fey, HPX: A Task Based Programming Model in a Global Address Space, in: Proceedings of the 8th International Conference on Partitioned Global Address Space Programming Models - PGAS '14, ACM Press, New York, New York, USA, 2014, pp. 1–11. doi:10.1145/2676870.2676883.
URL http://dl.acm.org/citation.cfm?doid=2676870.2676883

[16] H. Jordan, T. Heller, P. G. . I. . . . , U. 2018, The AllScale Runtime Application Model, in: 2018 IEEE International Conference on Cluster Computing (CLUSTER), 2018, pp. 445–455.
URL https://ieeexplore.ieee.org/abstract/document/8514904/

[17] W. Hundsdorfer, J. G. Verwer, Numerical solution of time-dependent advection-diffusion-reaction equations, Vol. 33, Springer Science & Business Media, 2013.

[18] T. Y. Miyaoka, J. F. d. C. A. Meyer, J. M. R. SOUZA, A General Boundary Condition with Linear Flux for Advection-Diffusion Models, TEMA (São Carlos) 18 (2) (2017) 253–272.

[19] T. Sauer, Numerical Analysis (2nd), Addison-Wesley, New Jersey, 2012.

[20] J. C. Butcher, Numerical methods for ordinary differential equations, John Wiley & Sons, 2016.

[21] G. Welch, G. Bishop, An Introduction to the Kalman filter. University of North Carolina at Chapel Hill, Department of Computer Science, Tech. rep., TR 95-041 (2004).

[22] S. Markidis, G. L. Simulation, R. Udin, Multi-scale simulations of plasma with iPIC3D, Elsevier 80 (7) (2010) 1509 –1519.
URL https://www.sciencedirect.com/science/article/pii/S0378475409002444

[23] G. Lapenta, S. Markidis, S. Poedts, D. Vucinic, Space weather prediction and exascale computing, Computing in Science and Engineering 15 (5) (2013) 68—-76.
URL https://ieeexplore.ieee.org/abstract/document/6244803/

[24] T. Tajima, Computational Plasma Physics, Westview Press, 2004. doi:10.1201/9780429501470.
URL https://www.taylorfrancis.com/books/9780429501470

[25] P. Boris, Relativistic plasma simulation-optimization of a hybrid code, in: Proc. 4th Conf. Num. Sim. Plasmas, 1970, pp. 3–67.
URL https://ci.nii.ac.jp/naid/10009996893/

[26] I. Herraiz, cmetrics (2007).
URL https://github.com/MetricsGrimoire/CMetrics

[27] M. Halstead, Elements of Software Science (Operating and programming systems series), Elsevier Science Inc., New York, NY, 1977.

[28] T. McCabe, A Complexity Measure, IEEE Transactions on Software Engineering SE-2 (4) (1976) 308–320. doi:10.1109/TSE.1976.233837.
URL http://ieeexplore.ieee.org/document/1702388/

[29] F. O'Donncha, D6.9 Installation, integration and deployment of the AllScale environment and pilot applications, Tech. rep., IBM Research – Ireland, Dublin (2018).