A localised data assimilation framework within the ‘AllScale’ parallel development environment

Albert Akhriev*, Fearghal O’Donncha*, Philipp Gschwandtner†, Herbert Jordan†
*IBM Research - Ireland
Email: {albert_akhriev, feardonn}@ie.ibm.com
†University of Innsbruck, Austria
Email: {philipp,herbert}@uibk.ac.at

***Abstract*—This paper presents a localised data assimilation framework for forecasting the evolution of marine oil spills. It consists of an advection-diffusion model together with data assimilation and adaptive meshing to improve the accuracy and precision of forecasts, respectively. To provide high parallel scalability, all computation is localised to individual subdomains with the solution being synchronized between direct neighbours at the end of each timestep. No global communication is required during the simulation. The scheme is developed within a novel programming environment aimed at facilitating efficient code development by leveraging advanced ‘separation of responsibilities’ principles. The front-end API provides the developer with a simple C++ development environment and a suite of parallel constructs that denote tasks to be operated concurrently. Lower level tasks related to the machine and system level are managed by computer scientists at the core-level. We present parallel scalability compared to a benchmark MPI implementation.**

I. Introduction

Feasible and scalable systems for the accurate estimation of advection diffusion processes are required in several applications. Examples include forecasting oil spill evolution for remediation efforts [1], quantifying the transport of nutrients around aquaculture installations [2] and monitoring releases from industrial operations [3]. Typically, these are provided from the solution of a set of Partial Differential Equations (PDEs) on a discretised grid. To improve the accuracy of the prediction, methods exist to update the solution using measurements of the actual state via data assimilation (DA). DA improves the accuracy of forecasts provided by physical models and evaluates their reliability by optimally combining *a priori* knowledge encoded in equations of mathematical physics with *a posteriori* information in the form of sensor data. The situation being studied reduces to an inverse problem, where one wishes to use 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.

A key challenge facing the merging of mathematical models and data is computational expense. One desires an approach that provides the best estimate of the ”true” solution cognizant of model limitations and sensor uncertainties [4]. With the drive to model more realistic and detailed simulations, the computational demands of numerical solutions increase. At the same time, the last few years have seen an increased abundance and availability of sensors data, ranging from satellite data to in-situ marine data. Developing computationally efficient DA implementation is challenging with PDEs since the computational costs and demands escalate with the increase in the degrees of freedom of the corresponding discretization. For this reason, methods that enable practical state estimation approaches by reducing the dimensionality of the problem and distributing across compute resources are very active research areas.

Domain Decomposition (DD) is a standard tool in many scientific domains to reduce the complexity or computational cost of solution. Some of the factors which have motivated DD approaches include: 1) the solution of the subproblems is qualitatively or quantitatively easier than the original, 2) the original problem does not fit into the available memory space and 3) the subproblems can be solved with some concurrency (i.e. in parallel). This has facilitated many advances in simulation capabilities in the geosciences with most operational large-scale models adopting this paradigm [5]–[7]. In this approach, subdomains are distributed across computational cores and solved independently with a periodic synchronization step to ensure the fidelity of the solution. Synchronization typically occurs at the end of each computational timestep and involves a communication of boundary solution state to neighbour subdomains with MPI being the most popular protocol for exchanging data and synchronizing solutions. To improve computational performance, fine-grained parallelism is often implemented within subdomains, via for-example the OpenMP paradigm.

Synchronization requirements and multiple parallelisation schemas enforced by performance considerations means that the development of efficient code places very high skill demands on the application developer, encompassing knowledge of sophisticated domain related algorithmic formulation and solvers together with complex software engineering skills. This is accentuated as degree of parallelism becomes larger and codes are deployed on hundreds of thousands to millions of computational cores. Indeed, recent efforts have addressed this such as the LFric research project from the UK Met Office that aims to develop a replacement for the Met Office Unified Model in order to meet the challenges which will be presented by the next generation of ExaScale supercomputers [8]. Design of the model revolves around a principle of a ’separation of concerns’, whereby the natural science aspects of the code can be developed without worrying about the underlying architecture, while machine dependent optimisations can be carried out at a high level (by HPC experts). A particular advocate of this ”separation of concerns” is the firedrake framework [9] which aims for an automated system for the portable solution of partial differential equations 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 express PDEs in a high-productivity intepreted language. The PyOP2 framework [12] provides an abstraction between the domain scientist concerned with implementing the PDE numerics and parallel execution over multi-core platforms.

AllScale (www.allscale.eu/) is a FET H2020 (Frontiers and Emerging Technologies in Horizon 2020) funded project that aims to provide computational paradigms that can tackle extreme-scale ExaScale computing (10^{12} 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 to improve productivity and the development environment. Those impose significant challenges for developers aiming to efficiently utilise all available resources. In particular, the development of such applications is accompanied by the complex and labour intensive task of managing parallel control flows, data dependencies and underlying hardware resources each of these obligations constituting challenging problems on its own.

In this paper we present the paralellisation of a DA scheme for advection-diffusion flows using a novel toolchain that empowers effective development of highly scalable parallel applications. The design of the ExaScale development environment, named the AllScale tool chain, is based on 3 key principles

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

From the application developer perspective, it promises highly increased productivity by hiding parallel constructs and providing a development API reminiscent of serial applications. Results are compared with a benchmark MPI implementation from the perspective of performance (computational throughput) and developer productivity (ease of development).

In the remainder of this paper, Section II describes the model governing equations and numerical implementation together with the data assimilation scheme. The AllScale development environment and API is also briefly introduced. Section III outlines the experimental results and compares with an MPI implementation. Finally, we draw conclusions from the study and present the future research steps.

II. Methodology

This paper focuses on the development, performance and scalability of an advection-diffusion code with localised data assimilation schemes 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 [13]) is compared against both benchmark MPI simulations and simulations using the standard GNU C++ runtime system..

A. AMDADOS

AMDADOS (Adaptive Meshing and Data Assimilation for Dispersion of Oil Spills) is a model for simulating conservative tracer transport in subsurface flows. It resolves the time-dependent advection-diffusion model on a discretized finite difference grid. This paper considers 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 evolution of the constituent concentration over time from sensors distributed within the domain (typically with some associated sensor uncertainty level). Applying the problem to the marine environment, the *data assimilation* problem can be formulated as follow: *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 dispersion over a spatial domain is described by the following equation [14]:

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

where D is diffusion coefficient, $v_x = v_x(x, y, t)$, $v_y = v_y(x, y, t)$ 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 are 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 go for a simple Dirichlet condition [15].

The numerical solver used is the implicit (or backward) Euler method; it is used for its unconditional stability and ability to handle stiff problems, although the method is only first order accurate in time [16], [17].

The data assimilation (DA) scheme employed is that of 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 representation, 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 of the form [18]:

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

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 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. A detailed survey of those methods can be found in [19]. 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. The size of a subdomain must be fixed because it is defined by template parameters in corresponding C++ class and should be available at compile time. Within each subdomain, the filtering of model and observations is implemented and at the end of each iteration, neighbouring subdomain solutions are synchronized. At the run time, each subdomain is assigned to so called worker either an execution thread or a process in case of distributed application. The assignment and workload balancing is done automatically once the grid of subdomains have been exposed to parallel AllScale operators (as described in Section II-B).

To improve the precision of the data assimilation framework and allow for more accurate specification of sensor location, subdomains are processed at different resolution using the *multi-scaling* capability of Allscale API. Namely, the subdomains with observations are processed at fine resolution because this yields better Kalman filtering estimation. On the other hand, domains without observations (where we just integrate the governing equation) are processed at coarser resolution with less computational cost. In order to yield a seamless solution we operate over *extended subdomain* that comprises border points of the neighbour subdomains. Extended subdomain is depicted by dotted rectangle on Fig. 1.

Algorithm 1 summarizes the major steps in the data assimilation method. The solution propagates forward in time over the period $[0 \ldots T]$ starting from zero density $u(x,y,t)|_{t=0} = 0$. The data assimilation method then nudges towards the actual density as observed by sensors. We employ Kalman filter (line 8) that brings otherwise simulated density closer to observation (if there is a sensor in the subdomain), and this important driving force is responsible for convergence of estimated density field towards the true state. In the absence of observations in a subdomain, the conventional integration step is fulfilled (line 11).

Algorithm 1 Data Assimilation Framework

```
 1: ### * * * * * AMDADOS solver. * * * * *
 2: Input: file of sensor locations, file of observations, initial
    field $u(x,y,t)|_{t=0} = 0,\ \forall x,y \in \Omega$.
 3: Require: sub-divide the whole domain $\Omega$ into a number
    of subdomains $N_{subdomain}$.
 4: ### Integrate forward in time and estimate the density field
    $u(x,y,t)$.
 5: for (with time-step $\Delta t$ ) $t = 0$ to $T$ do
 6:     for (in parallel using Allscale API) $c = 1$ to
        $N_{subdomain}$ do
 7:         if $c$-th subdomain contains at least one sensor
            then
 8:             Solve governing equation (1) forward in
                time inside $c$-th subdomain using dis-
                cretization in the form of: $\mathbf{u}_{t+1} = \mathbf{B}_t^{-1}\mathbf{u}_t$.
                (where $\mathbf{B}_t^{-1}$ represents an implicit Euler
                discretization).
 9:             Update state inside subdomain using the
                Kalman filter and observation value.
10:         else
11:             Solve governing equation (1) forward in
                time inside $c$-th subdomain using dis-
                cretization in the form of: $\mathbf{u}_{t+1} = \mathbf{B}_t^{-1}\mathbf{u}_t$.
12:         end if
13:     end for
14: end for
```

B. AllScale toolchain

This section outlines the AllScale programming environment and motivates why we chose it for the implementation of our application. The AllScale programming environment aims at providing a separation of concerns between domain scientists, HPC experts, and system level experts by offering a well-defined bridge between their worlds. This bridge, provided by the *AllScale API* consists of two parts that represent the basic building blocks of every parallel algorithm:

- parallel control flow primitives, and
- data structures.

The former are defined via a single, recursively parallel, higher-order operator (*prec*) [20], whereas the latter fulfil the concept of a *data item*. Both 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 allows any high-level operators and data structures needed

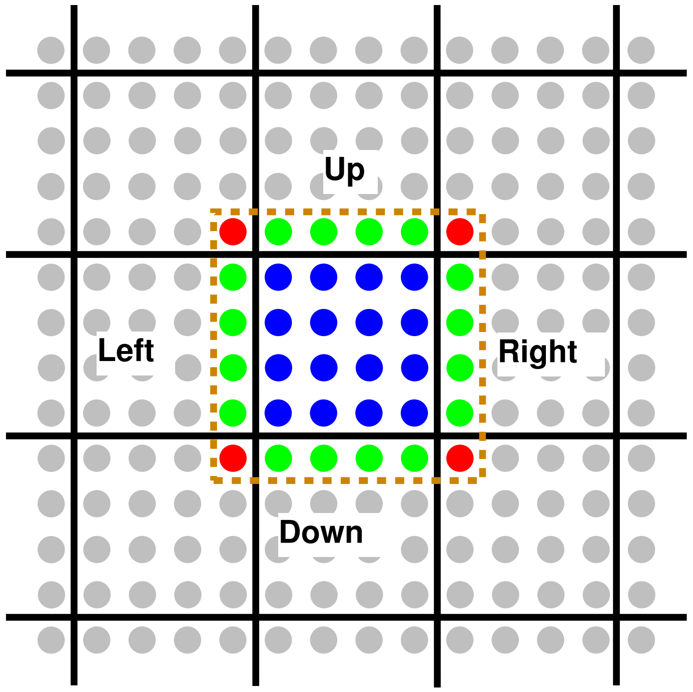

Fig. 1: Example of a grid subdomain, constituted by the blue nodal points, and corresponding extended one, outlined by dotted rectangle. Border (green) points of four immediate neighbour subdomains (*Left*, *Right*, *Up*, *Down*) are available for exchange.

by domain scientists (e.g. parallel loops, stencils, structured and unstructured meshes) to be implemented by HPC experts on top of these two components in the extensible *AllScale User API*. The high-level constructs of the User API can then be used by domain scientists without requiring knowledge on the design of scalable operators or the need for low-level data management or 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 that can be reused among multiple applications, reducing overall development overheads. Finally, 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. Furthermore, as the AllScale API is implemented as an embedded DSL [21] 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 [13]. Its application runtime model [22] 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*, and while it provides additional features, their discussion is omitted for brevity as they are not used in this work. Each runtime task can either 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, per-subdomain 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.

The specific operator used the work presented here is the *stencil* operator, which provides domain decomposition in both space and time in order to generate parallelism. Several implementations of this operator are available, including one with fine-grained task synchronisation that removes the otherwise global barrier often occurring in many implementations at the end of every stencil iteration. The data item in use is a regular, adaptive grid providing a user-defined, per-subdomain number of refinement levels and appropriate element access utilities.

By following the open/close principle, the AllScale API offers but is not limited to additional parallel primitives such as *pfor* (parallel loops), *preduce* (parallel reductions), or *vcycle* (V-cycle multi-grid solvers). Similarly, while the application presented in this work uses a regular adaptive grid, additional data structures currently offered by the API include non-adaptive grids, maps, trees, or unstructured meshes [23], [24].

III. Results

An assessment of an application developed within a novel programming environment such as AllScale requires consideration of three points: 1) one must ensure that the computed solution is correct and appropriate algorithmic sophistication is supported, 2) the parallel scalability of the application and 3) whether the API improves developer productivity and code maintainability. Experiments on a shared memory cluster were conducted to assess parallel scalability and the correctness of solution of the model. Experimental tests were conducted on compute server with 2 Intel Xeon 2.20GHz processors providing total of 44-core machine with Linux RedHat-7.4, 64-bit operating system. An MPI implementation served as benchmark of parallel scalability.

Many technical details differentiate the two implementation and how they invoke parallelism at different levels of inter- and intra-node computation. MPI is aimed primarily at distributed memory parallelism and requires explicit communication (e.g via MPI_Send/MPI_Recv). It is often implemented using an MPI + X model where MPI provides inter-node parallelism and a thread based paradigm such as OpenMP managing intra-node parallelism – this comes with the development overhead of having two separate protocols to manage (and code bases to maintain, potentially). The AllScale environment, which leverages the AllScale toolchain, supports both distributed and shared memory implementation. It computes over *localities* (inter-node) and *threads* (intra-node). In essence, it combines

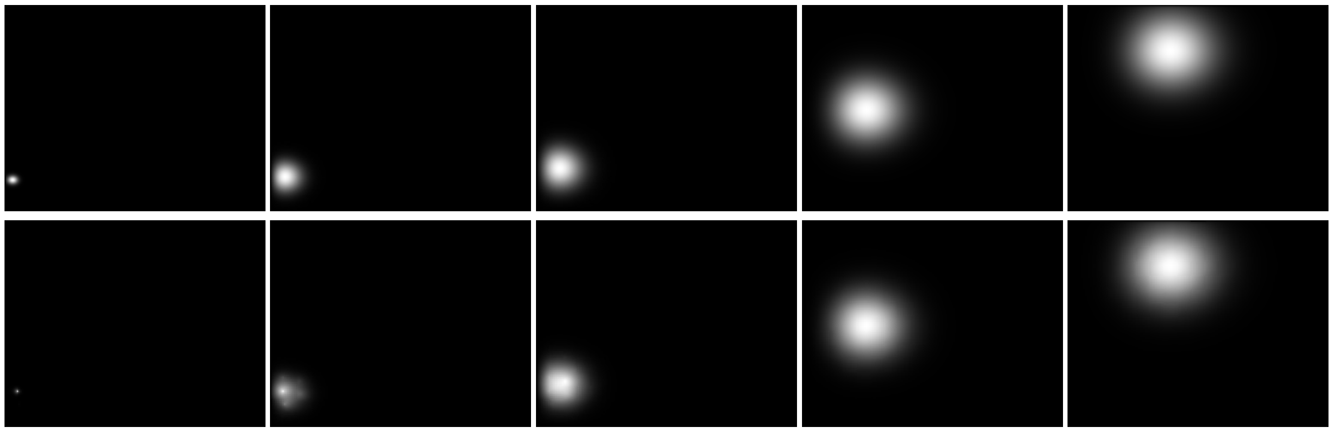

Fig. 2: Simulation of advection-diffusion process. *Top row*: Evolution in time of the “ground-truth” solution computed offline as a single global domain. *Bottom row*: data-assimilation solution that starts from zero density field and gradually catches up the ground-truth. There are 72960 nodal points representing 304×240 domain divided into 19×15 sub-domains for parallelization and 182 sensors pseudo-randomly scattered therein.

the advantages of MPI inter-node and thread-based parallelism intra-node within a single parallel programming paradigm.

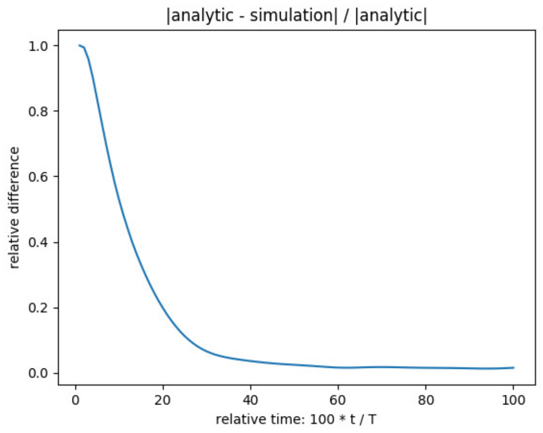

Fig. 3: Relative difference ($\varepsilon = \|u_{gt} - u\|_2 / \|u_{gt}\|_2$) between the ground-truth density and data-assimilation solution, as a function of “relative time”: $\tau = 100\,t/T$, where t is a physical time in seconds, and T is an integration period.

Domain decomposition based approaches have huge applicability in simulation due to the promise of reduced computational demand (by distributing across compute resources, reducing the size of linear algebra matrices, etc.). An important consideration however, is to ensure fidelity of the solution; i.e. the computed solution should be qualitatively (if not quantitatively) equivalent to that 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 also serves as the true solution against which the computed result from the distributed model can be readily compared

Fig. 2 presents snapshots of results from a number of stages during the simulation cycle. Results are compared to the “ground-truth” solution computed as part of the observation generator routine (i.e. routine that generates data for assimilation into the model).

Fig.3 shows how relative error fades away as simulation progresses. 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 the true distribution when sufficient sensor information on the true state is ingested. Of importance, no contamination of results develops from boundary exchange protocols; i.e. no aliasing is evident at sub-domain boundaries in Fig. 2 while Fig. 3 demonstrates that data assimilation directs error towards zero over time.

Performance results are a key metric of this analysis. We focus on weak scaling analysis to provide insight into the ability of the model to increase computational throughput as the number of compute threads increases. Throughput for this study is defined as the number of subdomains computed per second simulation time. In this experiment, the number of experiments were increased proportionally with compute threads (with 8 subdomains assigned to each thread to ensure compute saturation). As some of the subdomains contain observation data (and hence invoke data assimilation), the computational expense of subdomains varies significantly (with a data assimilation step being approximately four times more expensive than a Fig 4 presents the weak scaling results comparing AllScale implementation to the MPI benchmark. We see that at low number of compute threads, the computational throughput is quite similar. As number of cores increases, AllScale significantly outperforms the MPI implementation. At 44 threads, AllScale processes 36.8 subdomains compared to 25.5 for MPI – an increase of 44%.

This increase in computational performance is comple-

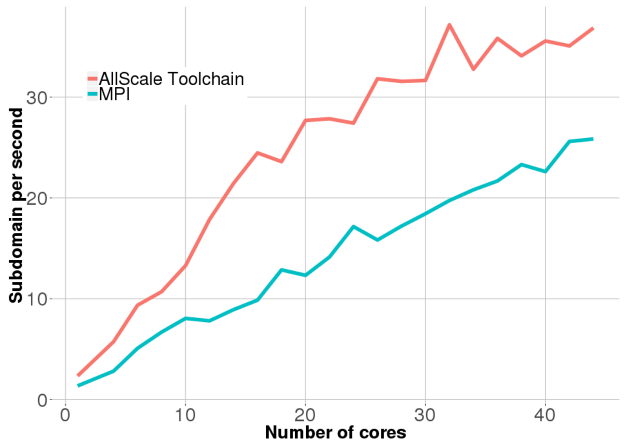

Fig. 4: Weak scaling test results computed using the AllScale API. Results present throughput – defined as the number of subdomains computed per second – plotted against number of threads, with the size of the problem domain (i.e. number of subdomains) increased proportionally with compute threads. Results compare the AllScale implementation (red) to MPI (cyan).

mented by a more delineated development interface that makes performance tuning and optimisation simpler. In a shared memory implementation such as this, OpenMP would likely give superior performance than MPI by avoiding unnecessary data exchange; this however would involve developing and maintaining an additional parallel programming paradigm. The Allscale implementation leverages thread based parallelism intra-node with communication between nodes thereby avoiding the overheads of MPI in shared memory.

From the application developer perspective, a key difference between the AllScale and MPI implementation is the level of control that the developer must enforce over communication. Within the AllScale API, synchronisation aspects are managed at the core API level facilitating trivial implementation of boundary exchange operations. The user simply invokes what data is passed to particular neighbours and the runtime system manages aspects such as ordering of send/receive, computational overlapping, etc. This can be an arduous and error-prone task in MPI to provide an efficient implementation that ensures MPI send/receive pairs are correctly coupled and there is no excessive wait-time between when processes send and receive data. This greatly simplifies coding implementation. Following the templates provided by the AllScale SDK users can develop a huge range of domain decomposition based applications with little knowledge of HPC or parallel computing concepts (i.e. simply by learning usage of AllScale API).

IV. Conclusions

This study demonstrates the capabilities of the AllScale toolchain and its feasibility as part of the next generation of HPC programming environments. Developing within the AllScale user API provides many advantages to the scientist. User productivity is 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 architecture.

Work is ongoing on performance tuning of a distributed memory implementation using the AllScale toolchain. Large scale experiments will be conducted to evaluate 1) scalability of the code at the many 1000 core level and assess the performance of the load balancing module (data assimilation provides a valuable test-case due to the very different computational expense of subdomains depending on whether observation exists or not). To meet the needs of the scientific community, the objective is a code that provides a delineated, maintainable code (i.e. separation of concern) together with high scalability at large number of cores.

Acknowledgements

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

References

[1] W. J. Guo, Y. X. Wang, M. X. Xie, and Y. J. Cui, "Modeling oil spill trajectory in coastal waters based on fractional Brownian motion," *Marine Pollution Bulletin*, vol. 58, no. 9, pp. 1339–1346, 2009.

[2] F. O'Donncha, M. Hartnett, and S. Nash, "Physical and numerical investigation of the hydrodynamic implications of aquaculture farms," *Aquacult. Eng.*, vol. 52, pp. 14–26, 2013.

[3] L. Koziy, V. Maderich, N. Margvelashvili, and M. Zheleznyak, "Three-dimensional model of radionuclide dispersion in estuaries and shelf seas," *Environmental Modelling & Software*, vol. 13, no. 5-6, pp. 413–420, 1998.

[4] F. O'Donncha, M. Hartnett, S. Nash, L. Ren, and E. Ragnoli, "Characterizing observed circulation patterns within a bay using {HF} radar and numerical model simulations," *Journal of Marine Systems*, vol. 142, pp. 96–110, 2015.

[5] J. Michalakes, S. Chen, J. Dudhia, L. Hart, J. Klemp, J. Middlecoff, and W. Skamarock, "Development of a next generation regional weather research and forecast model," in *Developments in Teracomputing: Proceedings of the Ninth ECMWF Workshop on the use of high performance computing in meteorology*, vol. 1. World Scientific, 2001, pp. 269–276.

[6] F. O'Donncha, E. Ragnoli, and F. Suits, "Parallelisation study of a three-dimensional environmental flow model," *Computers & Geosciences*, vol. 64, pp. 96–103, 2014.

[7] G. E. Hammond, P. C. Lichtner, and R. T. Mills, "Evaluating the performance of parallel subsurface simulators: An illustrative example with PFLOTRAN," *Water resources research*, vol. 50, no. 1, pp. 208–228, 2014.

[8] T. Melvin, S. Mullerworth, R. Ford, C. Maynard, and M. Hobson, "LFRic: Building a new Unified Model," in *EGU General Assembly Conference Abstracts*, vol. 19, 2017, p. 13021.

[9] F. Rathgeber, D. A. Ham, L. Mitchell, M. Lange, F. Luporini, A. T. T. Mcrae, G.-T. Bercea, G. R. Markall, and P. H. J. Kelly, "Firedrake," *ACM Transactions on Mathematical Software*, vol. 43, no. 3, pp. 1–27, dec 2016. [Online]. Available: http://dl.acm.org/citation.cfm?doid=2988516.2998441

[10] M. S. Alnæs, A. Logg, K. B. Ølgaard, M. E. Rognes, and G. N. Wells, "Unified form language: A domain-specific language for weak formulations of partial differential equations," *ACM Transactions on Mathematical Software (TOMS)*, vol. 40, no. 2, p. 9, 2014.

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

[12] F. Rathgeber, G. R. Markall, L. Mitchell, N. Loriant, D. A. Ham, C. Bertolli, and 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, nov 2012, pp. 1116–1123. [Online]. Available: http://ieeexplore.ieee.org/document/6495916/

[13] H. Kaiser, T. Heller, B. Adelstein-Lelbach, A. Serio, and 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*, ser. PGAS '14. New York, NY, USA: ACM, 2014, pp. 6:1–6:11. [Online]. Available: http://doi.acm.org/10.1145/2676870.2676883

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

[15] T. Y. Miyaoka, J. F. d. C. A. Meyer, and J. M. R. SOUZA, "A General Boundary Condition with Linear Flux for Advection-Diffusion Models," *TEMA (São Carlos)*, vol. 18, no. 2, pp. 253–272, 2017.

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

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

[18] G. Welch and G. Bishop, "An Introduction to the Kalman filter. University of North Carolina at Chapel Hill, Department of Computer Science," TR 95-041, Tech. Rep., 2004.

[19] M. S. Mahmoud and H. M. Khalid, "Distributed Kalman filtering: a bibliographic review," *IET Control Theory & Applications*, vol. 7, no. 4, pp. 483–501, 2013.

[20] H. Jordan, P. Thoman, P. Zangerl, T. Heller, and T. Fahringer, "A context-aware primitive for nested recursive parallelism," in *Euro-Par 2016: Parallel Processing Workshops*, F. Desprez, P.-F. Dutot, C. Kaklamanis, L. Marchal, K. Molitorisz, L. Ricci, V. Scarano, M. A. Vega-Rodríguez, A. L. Varbanescu, S. Hunold, S. L. Scott, S. Lankes, and J. Weidendorfer, Eds. Cham: Springer International Publishing, 2017, pp. 149–161.

[21] P. Zangerl, H. Jordan, P. Thoman, P. Gschwandtner, and T. Fahringer, "Exploring the Semantic Gap in Compiling Embedded DSLs," in *2018 International Conference on Embedded Computer Systems: Architectures, Modeling, and Simulation (SAMOS)*. Zenodo, Jul. 2018. [Online]. Available: https://doi.org/10.5281/zenodo.1309475

[22] H. Jordan, T. Heller, P. Gschwandtner, P. Zangerl, P. Thoman, D. Fey, and T. Fahringer, "The AllScale Runtime Application Model (incl. Appendix)," in *2018 IEEE International Conference on Cluster Computing (CLUSTER)*. Zenodo, Jul. 2018. [Online]. Available: https://doi.org/10.5281/zenodo.1322421

[23] "AllScale API Wiki," 2018. [Online]. Available: https://github.com/allscale/allscale_api/wiki

[24] H. Jordan and P. Gschwandtner, "D2.6 AllScale API Specification (b)," AllScale: An Exascale Programming, Multi-objective Optimisation and Resilience Management Environment Based on Nested Recursive Parallelism. Project Number 671603, Tech. Rep., 2017.