Computer Physics Communications 220 (2017) 188–204

Contents lists available at ScienceDirect

Computer Physics Communications

journal homepage: www.elsevier.com/locate/cpc

Atlas: A library for numerical weather prediction and climate modelling

Willem Deconinck [*], Peter Bauer, Michail Diamantakis, Mats Hamrud, Christian Kühnlein, Pedro Maciel, Gianmarco Mengaldo, Tiago Quintino, Baudouin Raoult, Piotr K. Smolarkiewicz, Nils P. Wedi

European Centre for Medium-Range Weather Forecasts (ECMWF), Shinfield Park, Reading RG2 9AX, United Kingdom

A R T I C L E I N F O

Article history:
Received 10 January 2017
Received in revised form 31 May 2017
Accepted 5 July 2017
Available online 22 July 2017

Keywords:
Numerical weather prediction
Climate
Earth system
High performance computing
Meteorology
Flexible mesh data structure

A B S T R A C T

The algorithms underlying numerical weather prediction (NWP) and climate models that have been developed in the past few decades face an increasing challenge caused by the paradigm shift imposed by hardware vendors towards more energy-efficient devices. In order to provide a sustainable path to exascale High Performance Computing (HPC), applications become increasingly restricted by energy consumption. As a result, the emerging diverse and complex hardware solutions have a large impact on the programming models traditionally used in NWP software, triggering a rethink of design choices for future massively parallel software frameworks. In this paper, we present *Atlas*, a new software library that is currently being developed at the European Centre for Medium-Range Weather Forecasts (ECMWF), with the scope of handling data structures required for NWP applications in a flexible and massively parallel way. *Atlas* provides a versatile framework for the future development of efficient NWP and climate applications on emerging HPC architectures. The applications range from full Earth system models, to specific tools required for post-processing weather forecast products. The *Atlas* library thus constitutes a step towards affordable exascale high-performance simulations by providing the necessary abstractions that facilitate the application in heterogeneous HPC environments by promoting the co-design of NWP algorithms with the underlying hardware.

1. Introduction

In the early days of HPC, computational power was gained through taking advantage of vector instructions on large single-processor computers. With little modification to existing code bases, shared memory parallelisation was introduced when more and more processors were added to one computer, bringing forth multi-core units. However, when multi-node/multi-core architectures (many computers linked together) became widespread in the 1990s, the required effort to port existing codes to make use of distributed memory was significant, and often meant rewriting or redesigning large parts of the codes. Over a decade later, many NWP codes, such as the ECMWF Integrated Forecasting System (IFS), have grown in size to millions of lines, making use of hybrid parallelisation with distributed and shared memory on multi-node/multi-core architectures [1]. Here performance was gained through increasing the CPU flop-rates and the number of nodes of the HPC system.

In today's HPC landscape, it has become unfeasible to boost computational performance by increasing the CPU's clock speed as the Dennard scaling [2] has broken down since around 2006, so the increase in computational performance has to be obtained largely by introducing more parallelism. Enlarging the HPC systems with more nodes of current CPU technology will ultimately result in unaffordable energy costs for many NWP operational centres. For these reasons, and with the goal to develop solutions suitable for the next generation exascale systems, hardware vendors introduced many-core processors (also known as accelerators), such as Graphic Processing Units (GPUs) and Intel's Many Integrated Core (MIC) architectures. These computing technologies have a higher flop-rate and lower power consumption than traditional multi-core CPU processors, and they have less stringent cooling requirements. On the other hand, they suffer from lower single-thread performance than traditional multi-core CPU processors. To exploit their higher degree of parallelism, a substantial effort in redesigning existing codes is required, as with the advent of multi-node/multi-core architectures in the 1990s.

[*] Corresponding author.
E-mail address: willem.deconinck@ecmwf.int (W. Deconinck).

http://dx.doi.org/10.1016/j.cpc.2017.07.006

While there is no definitive answer regarding what the optimal solution in terms of performance and energy requirements is, many NWP centres are looking at hybrid strategies, having both traditional multi-core and many-core units within the same HPC system and inter-operate them exploiting at best the advantages of the two [3]. In the next few decades, there might be a complete transition to many-core architectures and some NWP centres have already adopted this solution [4,5]. Interoperating across different hardware is a key factor for a sustainable development of Earth system models and for an efficient software/hardware co-design [6]. In order to achieve these goals, it is essential to have full flexibility on the underlying data structure. This involves not only having flexibility in terms of algorithmic design and memory layout most suited for different hardware, but also enabling the development of different numerical modelling strategies for solving the partial differential equations (PDEs) describing the atmosphere and ocean dynamics.

A good example for the importance of the latter point is the current numerical modelling infrastructure adopted at ECMWF. The IFS at ECMWF solves the system of governing PDEs through a spectral-transform-based approach [7]. This approach requires discrete transformations between physical space (grid-points) and spectral space (spherical-harmonics or Fourier coefficients). In general, spectral-transform-based approaches require data-rich global communications that become the main limiting factor for extreme-scale computations [8]. In addition, the IFS has been developed targeting traditional multi-node/multi-core computing technologies while its portability and efficiency on emerging many-core hardware is still under investigation [9]. Research may show that alternative numerical strategies offer better scalability and efficiency for certain hardware configurations, such as compact-stencil grid-point methods that only require nearest-neighbour communication.

ECMWF is developing a library called *Atlas*, with the primary goals to exploit the emerging hardware architectures becoming available in the next few decades, and to support the development of alternative numerical algorithm strategies in operational NWP. These developments apply not only to the forecast model, but also to the post-processing of model output to generate *products*.[1]

Atlas is also expected to facilitate the coupling of an increasing number of Earth system components, such as the atmosphere, ocean, wave, surface, or sea-ice, and could effectively enhance existing couplers such as OASIS [10]. The challenge of coupling Earth system components has been addressed in the Earth System Modelling Framework (ESMF) [11] and the Earth System Prediction Suite [12]. ESMF and *Atlas* both provide similar fundamental building blocks for data structures and model development. However, *Atlas* has the distinct primary goal of accelerating novel numerical algorithm development for emerging hardware architectures, compared to ESMF's effort to enhance collaborative Earth system model development. More specifically, novel discretisation methods using hybrid unstructured meshes require specialised data structures currently not available in ESMF. Atlas does not aim to be used as an alternative to a coupler's "super structure", designed to couple different models or model components, but rather intends to be a flexible toolkit of components that can be combined to create custom parallel data structures.

The implicit assumption behind the design of *Atlas* is the ability to exploit the structure that may be present in a physical system. In a global NWP or climate model for example, there is a strong asymmetry of large horizontal and small vertical scales with dominant hydrostatic balance in the Earth's atmosphere. Moreover, there is a need to efficiently model the statistical effect of sub-grid-scale processes. The latter are typically arranged in independent vertical columns constituting what is called *physical parametrisation*, coupled via process-splitting to the atmospheric flow itself, called the *dynamical core*. The dynamical core in global NWP integrates the PDEs on the sphere whereas limited-area models (LAMs) solve the PDEs on a particular area of the sphere [13] to which lateral boundary conditions are supplied by a global model. Alternatively a LAM may be directly embedded into a global domain [14]. For more information on NWP and climate models, refer to [15].

Historically the development of an operational NWP model takes about 10 years. Therefore, it is imperative for *Atlas* to remain flexible and maintainable, aiming towards substantially reducing this development time. Given the aforementioned assumptions and restrictions, *Atlas*' aims are:

- Target massively parallel global and limited-area NWP and climate applications such as new dynamical core developments, and pre- and post-processing tools.
- Offer flexibility of choices in new numerical strategies and algorithmic paradigms under the same software platform to explore emerging hardware such as many-core architectures.
- Facilitate the implementation of different structured and unstructured point distributions on the sphere (global grids) and on limited areas of the sphere (non-global grids).
- Support different spatial discretisation strategies, such as the spectral transform approach currently used at ECMWF [7], compact-stencil finite volume methods [16–18], discontinuous spectral element methods [19–21] and possibly others, to solve the set of PDEs forming the dynamical core.
- Provide an array-type container to store variables (or fields) that is parallel-enabled variables (or fields) and provides support for domain-specific languages (DSL) such as GridTools [22]. The latter is achieved through an advanced data storage layer (for instance, Kokkos [23] or the GridTools native storage layer [22]). This core aim tries to ensure optimal use of emerging hardware such as many-core architectures.
- Support different programming languages, including Fortran and C++, providing object-oriented (OO) designs and data structure flexibility, so that *Atlas* can be used to update existing and support new code infrastructures.
- Provide object-oriented programming interfaces enhancing multi-disciplinary collaboration at multiple levels ranging from e.g. high-level mathematical operators, typically developed by domain scientists, to low-level data-storage abstractions, typically maintained by computer scientists.

The development of the *Atlas* library is part of the wider 'Scalability Programme' ongoing at ECMWF and *Atlas* represents one of the core strategic software infrastructure tools that ECMWF has initiated during the FP7 funded project on Collaborative Research into Exascale Systemware, Tools and Applications (CRESTA, http://www.cresta-project.eu). A first public version of the *Atlas* library is intended to be delivered as part of ESCAPE (http://www.hpc-escape.eu), a H2020 funded initiative that aims at finding energy-efficient numerical solution for Exascale computations [8], for which *Atlas* capabilities are used in the implementation of alternative numerical discretisations on emerging hardware.

[1] *Products* are fields that are disseminated upon request and post-processed to satisfy a customer's specific requirements.

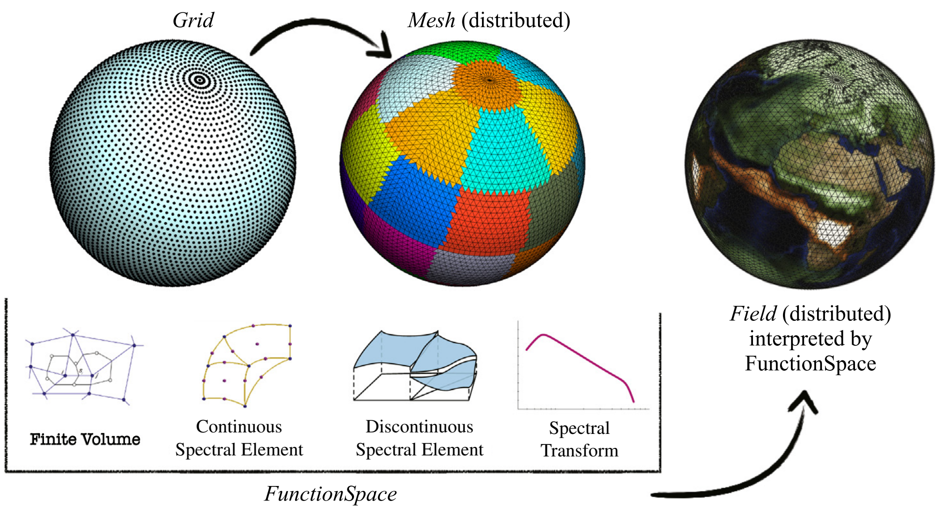

Grid
Mesh (distributed)
Field (distributed) interpreted by FunctionSpace
Finite Volume
Continuous Spectral Element
Discontinuous Spectral Element
Spectral Transform
FunctionSpace

Fig. 1. The conceptual design of *Atlas*.

In this paper, we present the conceptual design of *Atlas* by describing the key concepts in Section 2 and a typical operating workflow of the software. Implementation details of these concepts are presented in Section 3. We outline capabilities of the *Atlas* library through selected application examples in Section 4 for *global* NWP applications but these concepts apply to LAM applications as well. Section 5 summarises the paper and lays out future perspectives.

2. Conceptual design

Consider the following (simplified) PDE system

$$\frac{\mathrm{D}\boldsymbol{\phi}}{\mathrm{D}t} = \mathcal{R}_d(\boldsymbol{\phi}) + \mathcal{R}_p(\boldsymbol{\phi}). \tag{1}$$

The symbol $\boldsymbol{\phi}$ denotes the prognostic variables or fields of the model and $\mathrm{D}/\mathrm{D}t$ denotes the material derivative. The right hand side contains terms of resolved dynamics $\mathcal{R}_d(\boldsymbol{\phi})$ and physical parametrisations $\mathcal{R}_p(\boldsymbol{\phi})$. In particular, $\mathcal{R}_d(\boldsymbol{\phi})$ includes terms such as the pressure gradient and the Coriolis force. *Atlas* provides data structures for building a numerical strategy to solve Eq. (1). These data structures may contain a distribution of points (grid) and, possibly, a composition of elements (mesh), required to implement the numerical operations required. *Atlas* can also represent a given field $\boldsymbol{\phi}$ within a specific spatial projection. *Atlas* is capable of mapping fields between different grids as part of pre- and post-processing stages or as part of coupling processes whose respective fields are discretised on different grids or meshes. The latter is particularly relevant for the physics $\mathcal{R}_p(\boldsymbol{\phi})$, where some physical processes such as radiation may be represented on a coarser grid or mesh and may need to be projected onto a finer grid or mesh [24,25].

The key concepts in the design of the *Atlas* data structure are:

- *Grid*: ordered list of points (coordinates) without connectivity rules;
- *Mesh*: collection of elements linking the grid points by specific connectivity rules;
- *Field*: array of discrete values representing a given quantity;
- *FunctionSpace*: discretisation space in which a *field* is defined.

These concepts are depicted in Fig. 1, where we used the sphere to represent a global grid, mesh and field.

A *grid* is merely a predefined list of two-dimensional points, typically structured and using two indices `i` and `j` so that point coordinates and computational stencils (for e.g. derivatives) are easily retrieved without connectivity rules. In many cases a *grid* is enough to define *fields* with appropriate indexing mechanisms. For element-based numerical methods (generally unstructured) however, the *mesh* concept is introduced that describes connectivity lists linking elements, edges and nodes.

A *mesh* may be decomposed in partitions and distributed among MPI tasks. Every MPI task then allows computations on one such partition. Overlap regions (or halos) between partitions can be constructed to enable stencil operations in a parallel context.

In addition to a *grid* and *mesh*, it is necessary to introduce the concept *field*, intended as a container of discrete values of a given variable. A *field* can be discretised in various ways. The concept responsible for interpreting or providing the discretisation of a *field* in terms of spatial projection (e.g. grid-points, mesh-nodes, mesh-cell-centres) or spectral coefficients is the *function space*. The *function space* also implements parallel communication operations responsible for performing synchronisation of fields across overlap regions, which we refer to as halo-exchange hereafter.

A possible *Atlas* workflow consisting of the creation and discretisation of a *field*, is illustrated in Fig. 2, where we also emphasise some additional characteristics of each step.

The building blocks illustrated in Fig. 2 can then be used to implement additional operations required for specific applications.

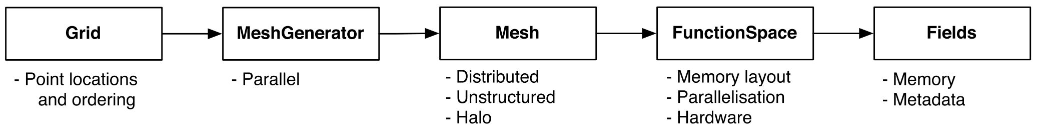


Fig. 2. Workflow of *Atlas* starting from *Grid* to the creation of a *Field*, discretised on a *Mesh* and managed by a *FunctionSpace*.

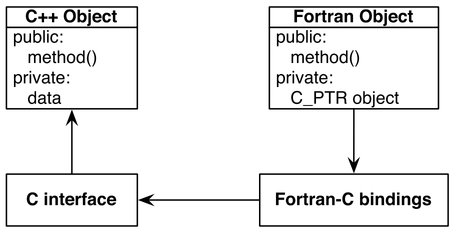


Fig. 3. Procedure how the Fortran interface to the C++ design is constructed. When a method in the Fortran object is called, it will actually be executed by the instance of its matching C++ class, through a C interface.

3. Implementation details

In this section, we present the implementation of concepts introduced in the preceding section, highlighting relevant snippets of code and outlining the OO design. We first introduce how the support for Fortran has been implemented, followed by each of the concepts introduced in the previous section: *Grid*, *Mesh*, *FunctionSpace* and *Field*. Lastly, we describe the parallelisation aspects and the software structure that performs operations of the vector calculus, using e.g. an edge-based finite volume spatial discretisation.

This section makes use of concepts from OO languages, in particular C++ [26], such as *class*, *derived type*, *member function*, *member variable*. Diagrams presented in this article make use of the Unified Modeling Language (UML) [27].

3.1. Programming languages

Atlas is primarily written in the C++ programming language. The C++ programming language facilitates OO design, and is high performance computing capable. The latter is due to the support C++ brings for hardware specific instructions. In addition, the high compatibility of C++ with C allows *Atlas* to make use of specific programming models such as CUDA to support GPU's, and facilitates the creation of C-Fortran bindings to create generic Fortran interfaces.

With much of the NWP operational software written in Fortran, significant effort in the *Atlas* design has been devoted to having a Fortran OO Application Programming Interface (API) wrapping the C++ concepts as closely as possible.

The Fortran API mirrors the C++ classes with a Fortran derived type, whose only data member is a raw pointer to an instance of the matching C++ class. The Fortran derived type also contains member functions or subroutines that delegate its implementation to matching member functions of the C++ class instance. Since Fortran does not directly interoperate with C++, C interfaces to the C++ class member functions are created first, and it is these interfaces that the Fortran derived type delegates to. The whole interaction procedure is schematically shown in Fig. 3.

The overhead created by delegating function calls from the Fortran API to a C++ implementation can be disregarded if performed outside of a computational loop. *Atlas* is primarily used to manage the data structure in a OO manner, and the actual field data should be accessed from the data structure before a computational loop starts.

3.2. Grid

In the NWP and climate modelling community (as opposed to, for instance, the engineering community) the grid is often a constant factor to be reused in many simulations, and one of *Atlas*' functions is to provide a catalogue of a variety of global grids defined by the World Meteorological Organisation in support of model inter-comparison initiatives.

The grids within *Atlas* are classified hierarchically from a completely unstructured to a fully structured interpretation. A non-exhaustive classification of grids used for global NWP is presented in Fig. 4. Note that the grid has no knowledge of any domain decomposition or parallelisation strategy.

In this classification the *Grid* class presents a generic unstructured view of all the points present in the grid (Fig. 5(a)). Even if the grid contains some form of structure, it can always be interpreted as being unstructured. A grid point is thus always accessible by a single index:

$$G = \{\boldsymbol{r}_1, \boldsymbol{r}_2, \dots, \boldsymbol{r}_{\mathtt{N}}\}, \qquad \boldsymbol{r}_{\mathtt{i}} \in \mathbb{R}^2 \tag{2}$$

where $\boldsymbol{r}_{\mathtt{i}} = [x_{\mathtt{i}}, y_{\mathtt{i}}]^{\mathrm{T}}$ is the $\mathtt{i}$th set of coordinates in a given frame of reference.

The next level in the inheritance tree is the *StructuredGrid* class, which makes the assumption that grid points are aligned equispaced in parallels along the *x* direction. Every parallel may have a different amount of gridpoints (Fig. 5(b) and Fig. 5(d)). No assumption is made on the spacing between the parallels in the *y* direction. The *StructuredGrid* class provides the grid point locations via a index $\mathtt{j}$ corresponding to the $\mathtt{j}$th parallel, and an index $\mathtt{i}$ corresponding to the $\mathtt{i}$th grid point on parallel $\mathtt{j}$:

$$G = f(\mathtt{i}, \mathtt{j}), \qquad f : \mathbb{N}^2 \rightarrow \mathbb{R}^2 \tag{3}$$

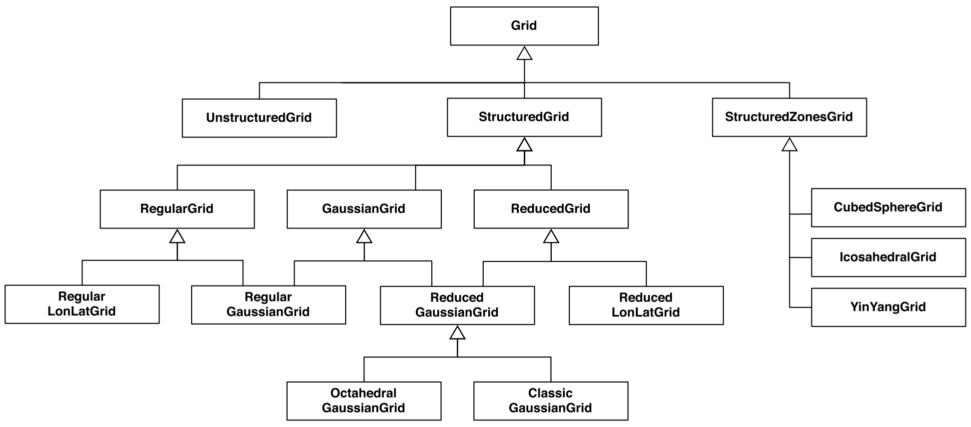


Fig. 4. A non-exhaustive classification of grid classes. Hollow arrows represent a "is a" relationship. With every step in the classification, more structure is present in the grid that can be exploited.

where f is generally a function of two indices, denoting the two coordinate directions, x and y. The grid is therefore a two-dimensional entity. A vertical direction z can be considered orthogonal to the horizontal direction and is then discretised by adding vertical layers.

The *ReducedGrid* class is a specialisation of the *StructuredGrid* that is explicit in having parallels with different amounts of gridpoints.

The *RegularGrid* class inherits from the *StructuredGrid* class and adds the restriction that every parallel has an equal number of grid points in the x-direction, making the grid regular and grid point locations are defined by two independent indices (`i`, `j`) associated to the x and y directions, respectively.

The *RegularLonLatGrid* is a global *RegularGrid* defined in geographical coordinates where parallels are equidistant in latitude. The grid includes parallels for the North Pole and the South Pole and every parallel contains a grid point on the Greenwich meridian.

The *GaussianGrid* is a global *StructuredGrid* that has parallels that are distributed according to the roots of the Legendre polynomial of order $2N$ where N is the number of parallels between a Pole and the equator. This distribution of the parallels is referred to as "Gaussian" and facilitates numerical integration and spherical-harmonics transforms for fields discretised on a Gaussian grid [28,29]. Note that a *GaussianGrid* by construction has no points on any of the Poles or the equator.

The *RegularGaussianGrid* is both a *RegularGrid* and a *GaussianGrid*, in which each parallel has $4N$ equidistant grid points of which one lies on the Greenwich meridian.

The *ReducedGaussianGrid* is both a *ReducedGrid* and a *GaussianGrid*, in which each parallel can contain a different number of equidistand grid points, of which one lies on the Greenwich meridian.

The *ClassicGaussianGrid* is a *ReducedGaussianGrid* in which the number of grid points on the parallels *reduces* towards the poles, keeping the physical distance between grid points approximately equal [28,29].

The *OctahedralGaussianGrid* is similar to the *ClassicGaussianGrid*. However, the number of grid points on each parallel can be inferred from triangulating a regular octahedron projected onto the sphere. Certain modifications are required such as modifying the latitude of the parallels to the roots of the Legendre polynomial [30].

For convenience the above concrete grids can be retrieved by a short grid identifier, e.g. `O1280`, in which the number represents the number of parallels between the North Pole (90° latitude) and equator (0° latitude). The grid identifiers for the grid classes *RegularLonLatGrid*, *RegularGaussianGrid*, *ClassicGaussianGrid* and *OctahedralGaussianGrid* are respectively `L#`, `F#`, `N#` and `O#`. These grid identifiers provide a common language to uniquely reference a grid in terms of resolution and category.

The classification further envisions to incorporate grids like the *IcosahedralGrid*, the *CubedSphereGrid*, and the *YinYangGrid* [31], which can be seen as combinations of sub-grids. These grids inherit from a class *StructuredZonesGrid* as illustrated in Fig. 4.

Listing 1 details how one can create the grids shown in Fig. 5 using either a grid identifier as described previously, or a *Config* object containing enough information to construct the correct grid. Such *Config* object could be constructed using JavaScript Object Notation (JSON), either programmatically (from a string) or from a file. Internally a Factory design pattern [32] is responsible for instantiating the correct concrete grid implementation.

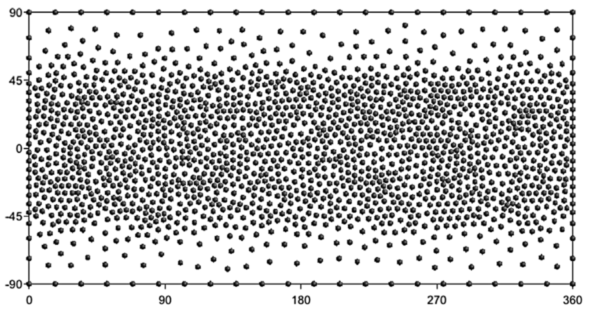
```
Grid unstructured( Config( file_path ) );
Grid N16( "N16" );
Grid L16( "L16" );
Grid O16( "O16" );
```

Listing 1: Construction of grids shown in Fig. 5

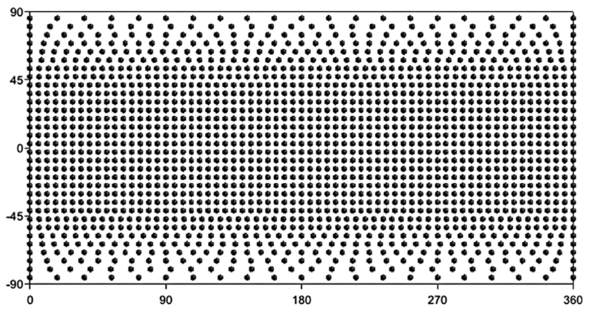
The object-oriented construction of the *Grid* object allows one to add any other grid of interest without disrupting the existing design.

3.3. Mesh

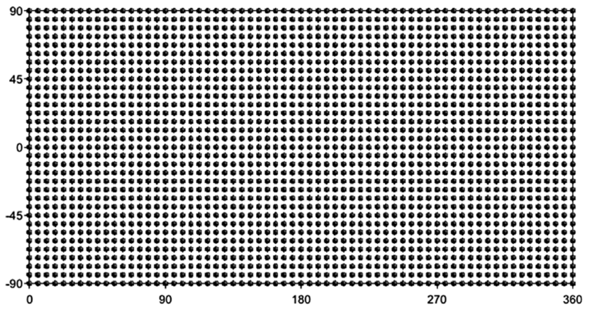
For a wide variety of numerical algorithms, a *Grid* (i.e. a mere ordering of points and their location) is not sufficient and a *Mesh* might be required. This is usually obtained by connecting grid points using polygonal elements (also referred to as cells), such as triangles or



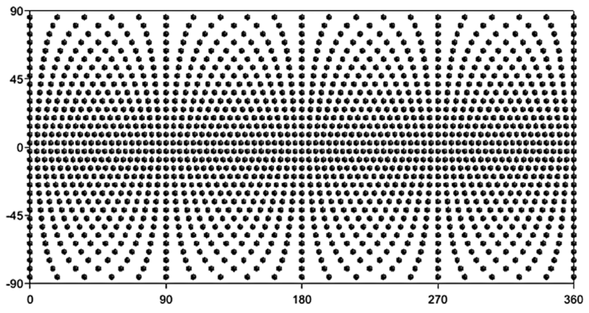
(a) UnstructuredGrid.



(b) ClassicGaussianGrid, `N16`.



(c) RegularLonLatGrid, `L16`.



(d) OctahedralGaussianGrid, `O16`.

Fig. 5. Four examples of global grids in geographical coordinates with similar resolution in the equatorial region. All grids can be represented by the base class *Grid*. Grids (b)–(d) can be represented by the *StructuredGrid* class. Grid (c) can furthermore be represented by the *RegularGrid* class.

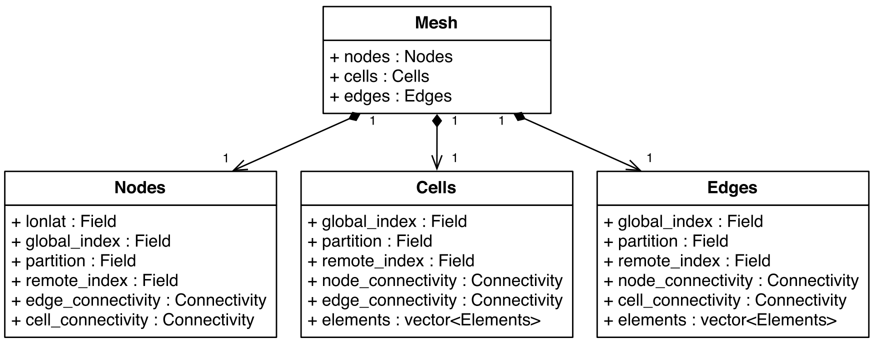


Fig. 6. Mesh composition.

quadrilaterals. A mesh, denoted by $\mathcal{M}$, can then be defined as a collection of such elements Ω_i:

$$\mathcal{M} := \bigcup_{\mathtt{i}=1}^{N} \Omega_{\mathtt{i}}. \tag{4}$$

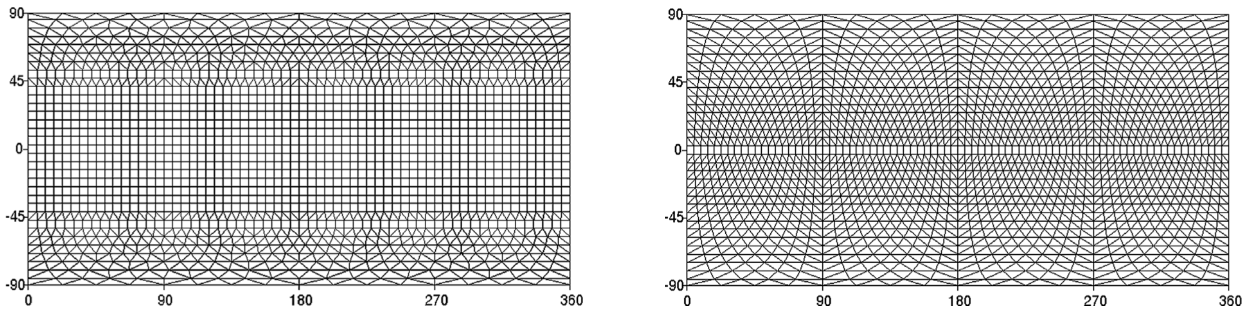
For *RegularGrids*, the mesh elements can be inferred, as a blocked arrangement of quadrilaterals. For the *UnstructuredGrid* class or even the *StructuredGrid* class (Section 3.2), these elements can no longer be inferred, and explicit connectivity rules are required. The *Mesh* class combines the knowledge of classes *Nodes*, *Cells*, *Edges*, and provides a means to access connectivities or adjacency relations between these classes (Fig. 6).

Nodes describes the nodes of the mesh, *Cells* describes the elements such as triangles and quadrilaterals, and *Edges* describes the lines connecting the nodes of the mesh. Fig. 6 sketches the composition of the *Mesh* class with common access methods for its components. Differently from the *Grid*, the *Mesh* may be distributed in memory. The physical domain S is decomposed in sub-domains S_p and a corresponding mesh partition $\mathcal{M}_{\mathrm{p}}$ is defined as:

$$\mathcal{M}_{\mathrm{p}} := \{\cup\, \Omega, \quad \forall\ \Omega\ \in\ \mathcal{S}_{\mathrm{p}}\}. \tag{5}$$

More details regarding this aspect are given in Section 3.4.

A *Mesh* may simply be read from file by a *MeshReader*, or generated from a *Grid* by a *MeshGenerator*. The latter option is illustrated in Fig. 2, where the grid points will become the nodes of the mesh elements. Listing 2 shows how this can be achieved in practice, and Fig. 7 visualises the resulting mesh for grids `N16` and `O16`.



(a) ClassicGaussianGrid, N16.   (b) OctahedralGaussianGrid, O16.

Fig. 7. *Mesh* generated for two types of *StructuredGrids* (Fig. 5).

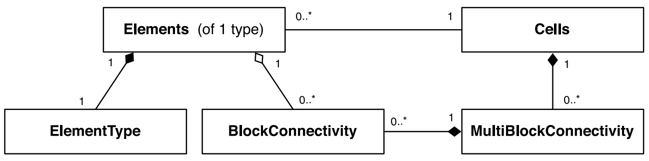


Fig. 8. Mesh *Cells* diagram.

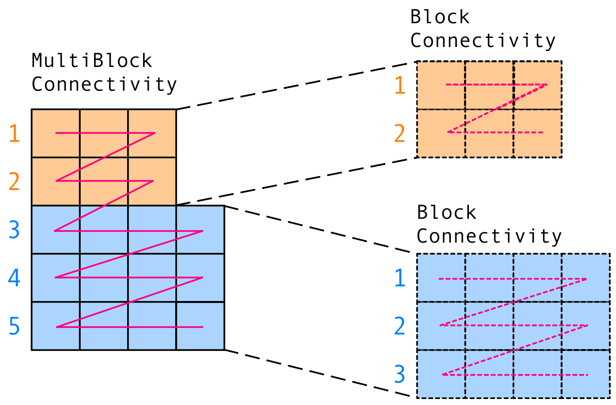


Fig. 9. *BlockConnectivity* points to blocks of *MultiBlockConnectivity*. Zig-zag lines denote how the data is laid out contiguously in memory.

```
Grid          grid( "O16" );
MeshGenerator generator( "structured" );
Mesh          mesh = generator.generate( grid );
```

Listing 2: C++ Mesh generation from a *StructuredGrid*

Several element types can coexist as cells, consequently the class Cells is composed from a more complex interplay of classes, such as *Elements*, *ElementType*, *BlockConnectivity*, and *MultiBlockConnectivity*. This composition is detailed in Fig. 8.

Often a numerical algorithm benefits from performing operations using elements of one element type at a time. This benefit typically comes from the constant number of nodes and edges that every element of one type connects to. To cater for this case, the *Elements* class provides node and edge connectivities as a *BlockConnectivity*, where every row of the connectivity table has the same constant size. The interpretation of the elements of this one type is delegated to the *ElementType* class. However, as some numerical algorithms do not require this distinction between element types, the class *Cells* provides a unified view of all elements regardless of their shape. The *MultiBlockConnectivity* provides a matching unified connectivity table.

To avoid duplication of memory, the actual data connectivity is stored in the *MultiBlockConnectivity* class, whereas *BlockConnectivity* instances point to blocks of the *MultiBlockConnectivity*, as can be seen in Fig. 9. Although currently the mesh is composed of two-dimensional elements such as quadrilaterals and triangles, three-dimensional mesh elements such as hexahedra, tetrahedra, etc. are envisioned in the design and can be naturally embedded within the presented data structure. However, at least for the foreseeable future in NWP and climate applications, the vertical discretisation may be considered orthogonal to the horizontal discretisation due to the large anisotropy of physical scales in horizontal and vertical directions. Given a number of vertical levels, polygonal elements in the horizontal are then extruded to prismatic elements oriented in the vertical direction (e.g. [33]).

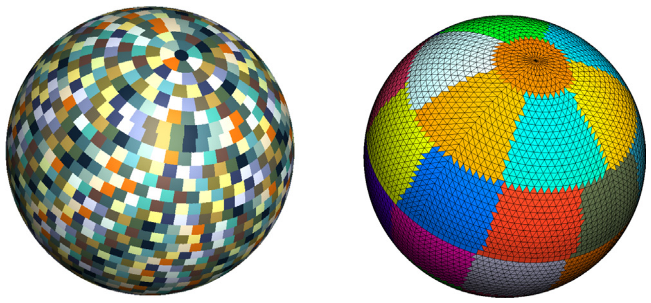


Fig. 10. *EqualRegions* domain decomposition. Left: O1280 mesh with ∼6.6 million nodes (∼9 km grid spacing) in 1600 partitions. Right: O32 mesh with 5248 nodes (∼280 km grid spacing) in 32 partitions.

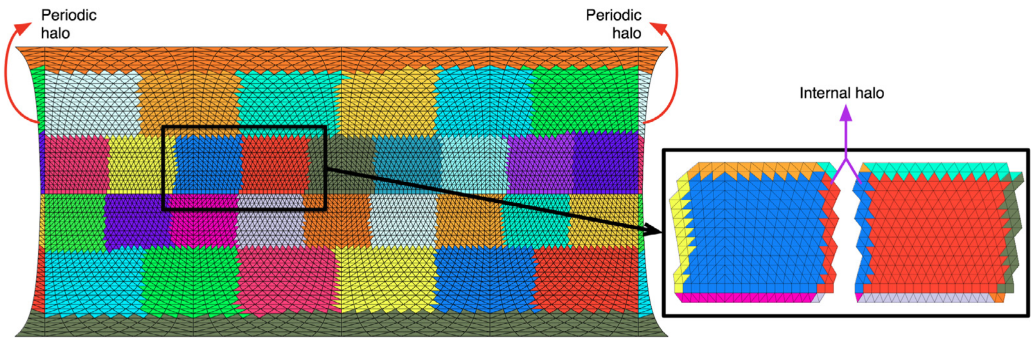


Fig. 11. Parallel overlap regions or halos shown for a O32 mesh with 32 partitions.

3.4. Parallelisation

Parallelisation in *Atlas* is achieved through distributing the *Mesh* into different partitions, each acting like a smaller mesh and each mesh partition $\mathcal{M}_\text{p}$ is managed by one MPI task. The idea is to load-balance numerical computations and memory among the MPI tasks, meaning that every mesh partition has approximately the same number of elements, or the same number of nodes.

There exist various strategies in how to partition a mesh, where each strategy may offer different advantages. While for *RegularGrids*, such as the one depicted in Fig. 5(c), the partitioning is logically following a checkerboard pattern, for general *ReducedGrids* as the ones shown in Fig. 5(b) and Fig. 5(d), an "equal regions" partitioning is more advantageous [8,34,35]. The "equal regions" partitioning algorithm divides a two-dimensional grid of the sphere (i.e. representing a planet) into bands from the North pole to the South pole. These bands are oriented in zonal directions and each band is then split further into regions containing equal number of nodes. The only exceptions are the bands containing the North or South Pole, that are not subdivided into regions but constitute North and South polar caps.

Examples of two meshes partitioned into different parallel regions using the *EqualRegions* partitioning algorithm are illustrated in Fig. 10.

Every mesh partition can be regarded as an independent mesh, but to allow for computational stencils that span from one mesh partition to the next, halos that overlap are created between relevant mesh partitions. *Atlas* provides functionality to incrementally grow the overlap between mesh partitions by node-sharing elements. Fig. 11 shows the overlap region generated for two such regions, as well as a so called "periodic overlap region" that can be used to treat the periodic East–West boundary as if it were an internal boundary between mesh partitions.

Discrete field values present in overlap regions require synchronisation with values of neighbouring partitions for performing stencil operations. For this synchronisation, the mesh partition must be aware of how it fits inside the whole mesh. As shown in Fig. 6, the *Nodes*, *Cells*, and *Edges* classes contain three fields, intended as discrete values, that provide exactly this awareness.

- The field named *global_index* contains a unique global index or ID for each node or element in the mesh partition as if the mesh was not distributed. The global index is independent of the number of partitions.
- The field named *partition* contains the partition index that has ownership of the node or element. Nodes or elements whose partition does not match the partition index of the mesh partition are also called ghost nodes or ghost elements respectively. These ghost entities merely exist to facilitate stencil operations (such as derivatives) or to complete, for instance, a mesh element.
- The field named *remote_index* contains the location of each node or element on the partition that owns it.

With the knowledge of partition and remote_index, it is possible to know, for each element or node, which partition owns it and at which index therein. Usually the *Atlas*' user will not be aware of these three fields as they are required only for constructing *Atlas*' internal parallel communication capabilities.

Currently, *Atlas* provides two parallel communication classes that, given the three fields such as *partition*, *remote_index* and *global_index*, can apply parallel communication operations repeatedly as needed:

- The *GatherScatter* class implements the communication operation that gathers data from all MPI tasks to one MPI task, and vice versa: the communication operation that scatters or distributes all data from one MPI task to all MPI tasks.

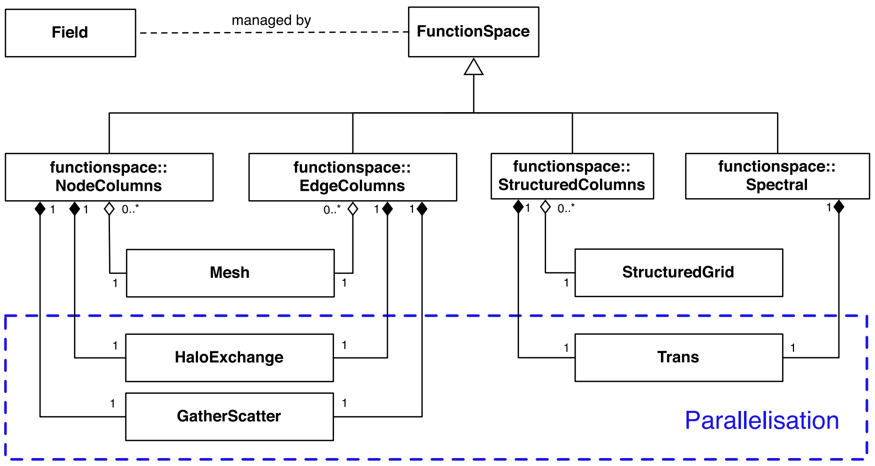


Fig. 12. *FunctionSpace* implementations including building blocks required to interpret *Fields* and abstract parallelisation.

- The *HaloExchange* class implements the communication operation that sends and receives data to and from MPI tasks containing nearest-neighbour partitions. This operation is typically required when synchronising small halos of ghost entities surrounding a domain partition.

These parallel communication classes form building blocks that provide parallel capabilities to the *FunctionSpace* class, which can manage the gathering, scattering or halo-exchanging of *Fields*.

Currently *Atlas* also optionally makes use of ECMWF's spherical harmonics transforms library, *Trans*, which implements communication operations for parallel distributed spectral fields discretised as spherical harmonics coefficients. It is envisioned that this functionality will instead be implemented in *Atlas* directly in future releases.

3.5. FunctionSpace

The *FunctionSpace* class ($\mathcal{P}$) is introduced because a *Field* ($\boldsymbol{\phi}$, Section 3.6) can be discretised on the computational domain in various ways: e.g. on a grid, on mesh-nodes, mesh-cell-centres or spectral coefficients. The representation of a given variable is intimately related to the spatial numerical discretisation strategy one wants to adopt (e.g. finite volume, spectral element, spectral transform, etc.). In addition to interpreting how a *Field* is discretised, the *FunctionSpace* also manages how the *Field* is parallelised and laid out in memory. Concrete *FunctionSpace* classes may implement parallel operations such as gather and scatter, reduce-all, or point-to-point communications, thus enabling the practical use of fields within parallel numerical algorithms. This step can be represented in a compact form as follows

$$\boldsymbol{\phi}_{\mathcal{P}} = \mathcal{P}(\boldsymbol{\phi}), \tag{6}$$

where $\boldsymbol{\phi}_{\mathcal{P}}$ indicates that the field is distributed and is fully enabled to perform numerical operations within the used representation.

In *Atlas*, the *FunctionSpace* concept, depicted in Fig. 12, is implemented in a modular OO paradigm that allows adding as many different function spaces as required. This modularity allows third-party applications to extend the library with their own *FunctionSpaces* while still profiting from the parallelisation primitives provided by *Atlas* (highlighted in dashed blue). The currently implemented *FunctionSpace* classes include *NodeColumns*, *EdgeColumns*, *StructuredColumns* and *Spectral*:

- The *NodeColumns* function space class describes the discretisation of fields with values collocated at the nodes of the mesh, horizontally, and may have multiple layers defined in the vertical direction. Parallelisation is defined in the horizontal plane, so that complete vertical columns are available on each partition. The memory layout for fields defined using the *NodeColumns* function space is illustrated in Fig. 13. A *HaloExchange* object and *GatherScatter* object are responsible for the necessary parallel operations (Section 3.4). The *NodeColumns* function space also implements some simple additional features, such as calculating global minimum and maximum values of fields as well as some global reduction computations such as arithmetic mean values.
- The *EdgeColumns* function space class describes the discretisation of fields with values collocated at edge-centres of the mesh, and may have multiple layers defined in a vertical direction. The various operations described for the *NodeColumns* class are also available for this class.
- The *StructuredColumns* function space class describes the discretisation of distributed fields on a *StructuredGrid* object. Currently the *StructuredGrid* must be Gaussian (see Section 3.2) because currently the function space delegates its parallel primitives to a specific *Trans* object that only supports Gaussian grids. As the *Trans* object is an interface with an external library that implements spectral transformations, we do not report the details here, but it is a good example of how *Atlas* interfaces with pre-existing high performance codes. In a future release the parallelisation will be generalised to use a *GatherScatter* object instead, which does not rely on having a Gaussian grid. A field described using this function space, like the two above, can also have vertical levels.
- The *Spectral* function space class describes a field in terms of vertical layers of horizontal spherical-harmonics (global spectral representation). The parallelisation (gathering and scattering) is again delegated to the *Trans* object.

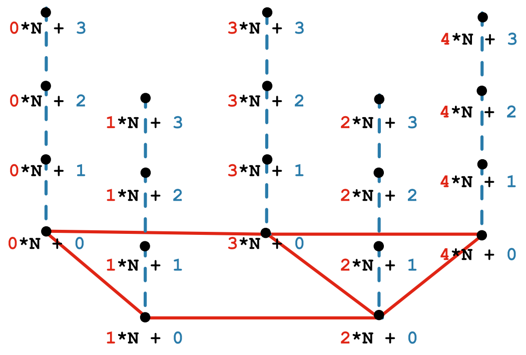


Fig. 13. Memory layout for fields discretised using the *NodeColumns* function space. A vertical column is contiguous in memory, and can be indexed using direct addressing. *N* stands for the number of vertical layers.

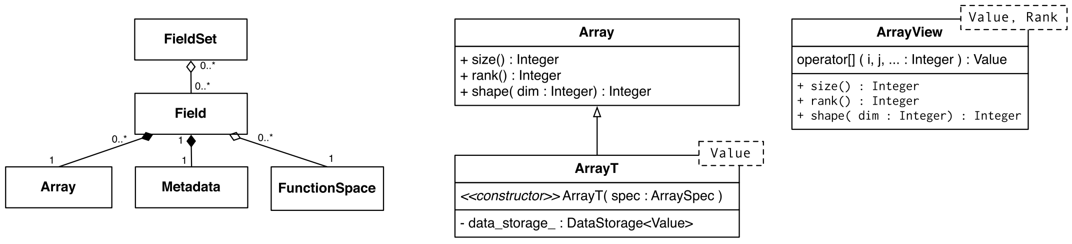

FieldSet
0..*
0..*
Field
0..*
1
1
1
0..*
1
Array
Metadata
FunctionSpace
Array
+ size() : Integer
+ rank() : Integer
+ shape(dim : Integer) : Integer
ArrayT
Value
<<constructor>> ArrayT(spec : ArraySpec)
- data_storage_ : DataStorage<Value>
ArrayView
Value, Rank
operator[] (i, j, ... : Integer) : Value
+ size() : Integer
+ rank() : Integer
+ shape(dim : Integer) : Integer

Fig. 14. Left: *Field* composition. Right: *Array* and *ArrayView* implementation.

With respect to parallelisation, each concrete *FunctionSpace* may implement methods like *haloExchange*, *gather*, *scatter*, or choose to delegate its implementation to one or more parallelisation primitives like *HaloExchange* and *GatherScatter*, which are then setup for the required memory layout. For global reduction operations, there are currently no primitives implemented, so that if required, each concrete *FunctionSpace* must implement the desired global reduction operations. This may be consolidated in future versions of the library.

Listings 3 and 4 are provided to help understand how a *FunctionSpace* can be used in practice to create a field, and perform a halo-exchange on this field. Listings 3 and 4 show both the C++ and the Fortran code, respectively.

```
NodeColumns functionspace( mesh, Levels(100), Halo(1) );
Field field = functionspace.createField<double>( "name" );
functionspace.haloExchange( field );
```

Listing 3: C++ *FunctionSpace* example use

```
type(atlas_NodeColumns) :: functionspace
type(atlas_Field)       :: field
functionspace = atlas_NodeColumns( mesh, levels=100, halo=1 )
field         = functionspace%create_field( "name", atlas_real(8) )
call functionspace%halo_exchange( field )
```

Listing 4: Fortran *FunctionSpace* example use

3.6. Field

The *Field* class contains the values of a full scalar, vector or tensor field. The *Field* values are stored contiguously in memory, and moreover they can be mapped to an arbitrary indexing mechanism to target a specific memory layout. The ability to adapt the memory layout to match, for instance, the most efficient data access patterns of a specific hardware is a key feature of *Atlas* . A *Field* also contains *Metadata* which stores simple information like a name, units, or other relevant information. The composition of the *Field* class is illustrated in Fig. 14.

A *Field* delegates the access and storage of the actual memory to an *Array* that accommodates memory storage on heterogeneous hardware.[2] If the *Field* is associated to a particular *FunctionSpace*, then the *Field* also contains a reference to it.

Fields can also be grouped together into one or more *FieldSet*s. They can then be accessed from the *FieldSet* by name or by index. In C++, access to the actual field data is via an *ArrayView* object that provides a multi-dimensional indexing accessor. In Fortran, the data is directly accessed through the multi-dimensional array intrinsics of the language. Practical use of the *Field*, both using C++ and Fortran, is given in Listings 5 and 6.

2 The *Array* is responsible to synchronise data across the device (e.g. a GPU) and the host (e.g. a CPU).

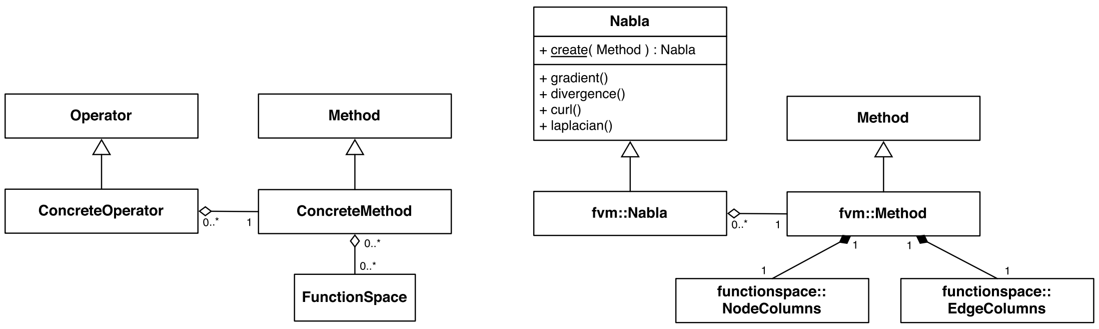


Fig. 15. Left: general design of numerical operators. Right: Derivative, divergence, curl, and Laplacian implemented in the *Nabla* vector operator specific for a finite volume *Method* (Section 4.3).

```
FieldSet fields;
fields.add( functionspace.createField<double>("temperature") );
fields.add( functionspace.createField<double>("pressure") );

Field field_T = fields["temperature"];
Field field_P = fields["pressure"];

ArrayView<double,2> T = make_view<double,2>(field_T);
ArrayView<double,2> P = make_view<double,2>(field_P);
for( size_t jnode=0; jnode<functionspace.nb_nodes(); ++jnode ) {
    for( size_t jlev=0; jlev<functionspace.nb_levels(); ++jlev) {
    // T(jnode,jlev) = ...
    // P(jnode,jlev) = ...
    }
}
```

Listing 5: C++ *Field* creation and data access

```
type(atlas_FieldSet) :: fields
type(atlas_Field)    :: field_T, field_P
real(8), pointer     :: T(:,:), P(:,:)

fields = atlas_FieldSet()
call fields%add( functionspace%create_field("temperature",atlas_real(8)) )
call fields%add( functionspace%create_field("pressure",   atlas_real(8)) )

field_T = fields%get("temperature")
field_P = fields%get("pressure")

call field_T%data(T)
call field_P%data(P)

do jnode=1,functionspace%nb_nodes()
   do jlev=1,functionspace%nb_levels()
   !  T(jlev,jnode) = ...
   !  P(jlev,jnode) = ...
   enddo
enddo
```

Listing 6: Fortran *Field* creation and data access

3.7. Mathematical operations

Many NWP and climate models contain algorithms to perform a variety of mathematical operations on fields such as computing derivatives or integrals. These operations are common to various applications, and relate closely to certain spatial discretisations or function spaces. *Atlas* provides implementations for some of these operations given a field that is compatible with the related *FunctionSpace* (Section 3.5). Fig. 15 sketches the philosophy adopted by *Atlas* regarding how to provide these operations.

The concrete implementation of the *Method* concept uses the *FunctionSpace* and *Field* classes, both required to generate a concrete numerical method. *Atlas* currently provides a *fvm::Method* class, which contains everything required to construct mathematical operators using an edge-based finite volume scheme (Section 4.3). A concrete *fvm::Nabla* operator then implements the actual numerical algorithm using the *fvm::Method*. Listing 7 details the practical construction of the *fvm::Method* and how the gradient of a scalar field defined in *NodeColumns* is computed using the *Nabla* operator.

```
fvm::Method method( mesh );
Nabla  nabla( method );
Field  scalar_field, gradient_field;
// Assume scalar_field, and gradient_field are defined in functionspace::NodeColumns
nabla.gradient(scalar_field,gradient_field);
```

Listing 7: C++ numerical operator Nabla that computes the gradient of a scalar field

4. Examples of applications

This section illustrates use cases of the *Atlas* library as part of different applications, highlighting the flexibility of the library in different contexts. Through *Atlas*, the development time of these applications was decreased significantly, as *Atlas* readily provides mesh generation functionalities, parallelisation solutions and certain mathematical operations. First we present two applications in Section 4.1 relying on *Atlas* that each implement a different advection scheme to solve a simple transport equation on the sphere. We then present how *Atlas* is used in the existing Fortran-based IFS model to contribute element-based gradient computations in Section 4.2. Section 4.3 then illustrates developments of a novel compact-stencil finite volume based dynamical core to solve the non-hydrostatic equations governing the atmosphere.

Finally, in Section 4.4, we present the developments of a new interpolation and remapping infrastructure which can be used for post-processing purposes or eventually within a model.

Most presented applications contain the following code to define distributed fields that are discretised collocated with a given grid's points.

```
// Input
string grid_name  = "O1280"
size_t nb_levels  = 137;
size_t halo_width = 1;

// Generate a parallel distributed mesh from a given StructuredGrid's name
Grid grid ( grid_name );
Mesh mesh = MeshGenerator( "structured" ).generate( grid );

// Define discretisation of fields using a function space
NodeColumns functionspace( mesh, Levels(nb_levels), Halo(halo_width) );

// Coordinate field in longitude and latitude (degrees)
Field lonlat = mesh.nodes().lonlat();

// Create fields
FieldSet fields;
fields.add( functionspace.createField<double>( field_name ) );

/* ... Initialise fields (see Listing 5) ... */

// Parallelisation
functionspace.haloExchange( fields );
```

Listing 8: Code used by most application examples to define distributed fields discretised collocated with a given grid's points.

The code presented in Listing 8 could also be written using Atlas' Fortran API. All applications can then use access methods to the fields' data as described in Listing 5 or 6. The correctness of the Atlas implementations is verified by a suite of unit-tests, and confidence is obtained through the correctness of following application examples.

4.1. Trajectory-based semi-Lagrangian and control-volume-based Eulerian advection approaches

Non-linear advective transport is fundamental in NWP and climate applications, where it is commonly applied to run the model forward in time for days (NWP) or up to hundreds of years (climate). Two approaches to advective transport are the *trajectory based* semi-Lagrangian (SL) method [36], and the formally conservative and sign-preserving *control-volume-based* MPDATA (multidimensional positive definite advection transport algorithm) method [18,37,38].

The two approaches are applied to the following passive advective transport problem

$$\frac{D\psi}{Dt} = 0, \qquad \frac{D}{Dt} = \frac{\partial}{\partial t} + \mathbf{v}\cdot\nabla, \qquad \mathbf{v} = (u, v, w), \tag{7}$$

on a three-dimensional geospherical domain (longitude$\times$ latitude$\times$ height). In Eq. (7) the scalar tracer field ψ

$$\psi(\lambda,\phi,z) = \begin{cases} 1 & \lambda\in[70^\circ,110^\circ]\ ,\ \phi\in[-20^\circ,20^\circ] \\ 0 & \text{otherwise} \end{cases}$$

is advected by a prescribed wind field $\mathbf{v} = (u, v, w)$

$$\begin{aligned} u(\lambda,\phi,z) &= \alpha\omega\cos(\phi) + \alpha K\cos(4\lambda)\cos(\phi)^3, \\ v(\lambda,\phi,z) &= -4\alpha K\sin(\phi)\sin(4\lambda)\cos(\phi)^3, \\ w(\lambda,\phi,z) &= 0, \end{aligned}$$

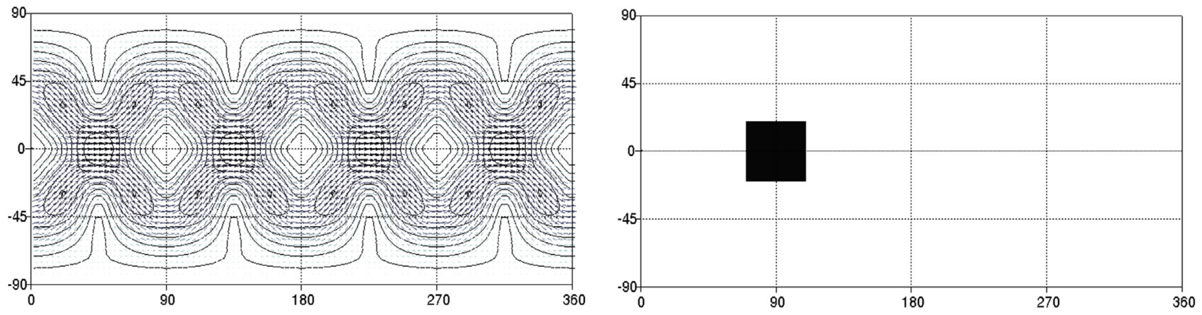


Fig. 16. Left: constant in time wind field where solid lines mark the wind isolines. Right: tracer field at initial time.

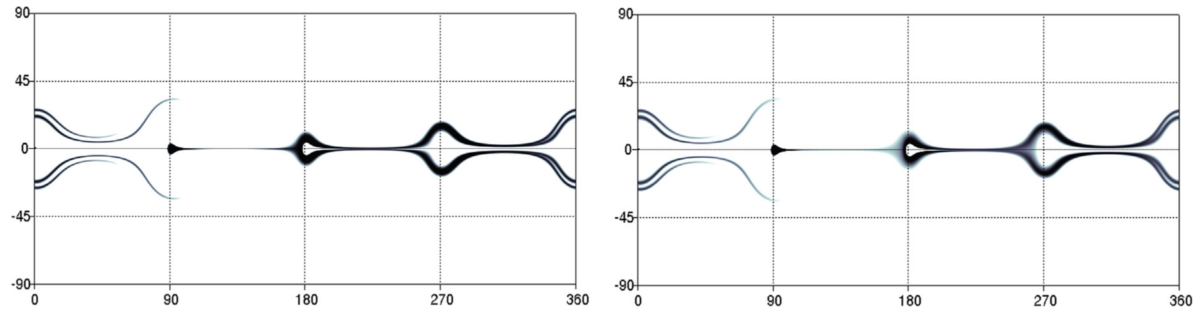


Fig. 17. Tracer field as initialised in Fig. 16 at t+7 days simulated using SL method (left) and MPDATA (right) using the same O512 mesh.

where $\alpha = 6371 \times 10^3$ [m] is the Earth radius, $\omega = K = 7.848 \times 10^{-6}$ [s^{-1}] is the Earth angular velocity and λ, ϕ are the longitude and latitude, respectively. The prescribed wind field is derived from the Rossby–Haurwitz test case [39] initial condition, and kept constant in time. Fig. 16 shows both the wind field (on the left) as well as the initially square-shaped tracer (on the right). The tracer is evolved forward in time for 7 days, and Fig. 17 shows the resulting tracer field at the end of the simulation. The subfigure on the left represents the trajectory based SL solution, while the subfigure on the right depicts the control-volume-based MPDATA solution.

The time-step used for the SL method is 900 [s] and the horizontal resolution is approximately 20 [km] corresponding to a maximum CFL number (timestep $\times$ wind speed/grid spacing) close to 5. MPDATA used a time step such that the maximum CFL number was 0.5. Even though the time step for the MPDATA scheme is smaller, the scheme contains no conservation errors.

In both applications *Atlas* aided in mesh generation and the parallelisation. Whereas the SL application currently is restricted to structured grids, the MPDATA algorithm is applicable to unstructured meshes, and *Atlas* helped in managing the complexities of such data structures.

4.2. Compute grid-point element-based derivatives in IFS using Atlas

The operational IFS model at ECMWF is a spectral-transform model, relying on spherical-harmonics transforms to compute horizontal derivatives. In the IFS, the *Atlas* library introduced the possibility of computing derivatives in grid-point space using the *Atlas* library with a finite volume based second-order scheme. The required derivatives have been constructed as a mathematical operator on *Field* classes (cf. Section 3.7). An example of derivatives computed with the spectral-transform method compared to derivatives computed with *Atlas* can be seen in Fig. 18.

It can be seen that there are some small differences in proximity to steep gradients (where the density of the isolines increases). In order to compare the difference in using spectral derivatives (the current operational approach) and grid-point second-order finite volume derivatives during a forecast simulation, we used the virtual temperature T_v field, defined as

$$R\,T \equiv R_{\mathrm{dry}}\,T_v = R_{\mathrm{dry}}\,T\left[1 + \left(\frac{R_{\mathrm{vap}}}{R_{\mathrm{dry}}} - 1\right) q\right]. \tag{8}$$

In this equation, T is the temperature, R_{vap} and R_{dry} are the specific gas constants of water vapour and dry air, respectively, and q is the specific humidity. The derivatives of $\mathrm{T_v}$ are then used inside the IFS model to calculate the horizontal gradients of the geopotential as needed in the rhs of the momentum equation.

To calculate the horizontal derivatives of geopotential we used three different approaches. We first adopted the operational methodology of calculating the derivatives of $R\,T$ using the spectral transform approach (Spectral). As an alternative to this methodology we used spectral derivatives for only T and computed additionally the local derivatives of q (Grid-point 1), or in a second alternative the local gridpoint derivatives of the specific gas constant R (Grid-point 2). The latter may be advantageous as moist variables in IFS are never transformed to spectral space to avoid spurious negative values. The mean scores of 12 forecasts using the three different methods can be seen in Fig. 19. Scores are a measure of forecast skill and are computed in this case as the anomaly correlation of 500 hPa geopotential height in the Northern hemisphere verified against ECMWF operational analysis.

Up to five days into the forecast, the three approaches show identical behaviour. From day six, the two grid-point approaches (overlapped) deviate from the reference. Given the small sample size it can only be concluded that the differences are larger than typically

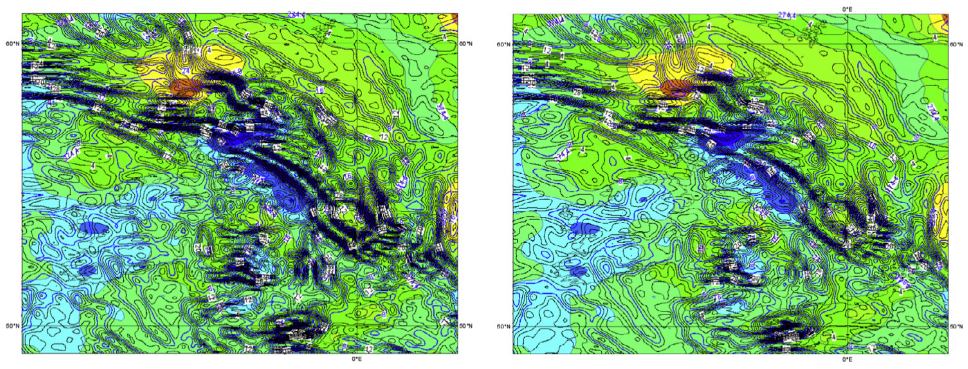


Fig. 18. Solid lines show the magnitude of the specific humidity q horizontal gradient obtained using the spectral-transform method (left) and using the grid-point finite volume based second-order method (right). The coloured shaded areas show the temperature at the 850 hPa model level. (For interpretation of the references to colour in this figure legend, the reader is referred to the web version of this article.)

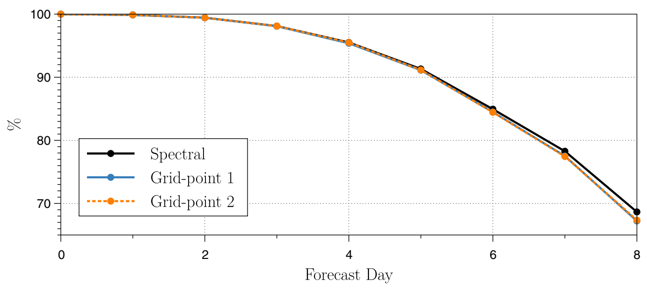


Fig. 19. Scores comparing forecasts using T_v derivatives computed with the spectral transform approach and compared to two alternative approaches where part of the derivatives of moist quantities are calculated locally in grid point space.

incurred by a random perturbation and they may depend on the implementation details of the gradient calculation. Nevertheless, this application example highlights the ability of the *Atlas* library to interplay with the existing Fortran-based IFS code and to contribute with new algorithmic developments (i.e. compact-stencil grid-point derivatives). The possibility of using locally computed grid-point gradients within the IFS model is attractive for scalability reasons since local gradients are cheaper to calculate, and they may also be used to calculate auxiliary local flow diagnostics not readily available in the spectral context.

4.3. Non-hydrostatic finite volume dynamical core

The FVM of the IFS is developed as an alternative dynamical core module at ECMWF to supplement the spectral formulation currently employed in operational forecasting [16] and is built using *Atlas*. FVM integrates the compressible Euler equations in a geospherical framework [16,40]. The horizontal spatial discretisation is fully unstructured using the median-dual finite volume approach. This is combined with a structured flux-form finite difference approach in the vertical direction. A centred two-time-level integration scheme is employed with 3D implicit treatment of acoustic, buoyant, and rotational modes [17]. The integration procedure uses the finite volume implementation of the multidimensional non-oscillatory MPDATA advection scheme [38,18]. A generalised, optionally adaptive, terrain-following vertical coordinate is implemented to accommodate the underlying orography [41,42]. For interoperability with the currently operational spectral IFS, the finite volume mesh of FVM is built about the points of the octahedral reduced Gaussian grid depicted in Fig. 5(d).

Fig. 20 illustrates FVM capabilities for the simulation of a tropical cyclone at intermediate complexity [43]. The model configuration involves the moist compressible formulation of FVM [44] with coupling to parametrisations for warm-rain cloud microphysics, surface fluxes from the ocean, and vertical turbulent mixing in the planetary boundary layer. The results after 10 days of simulation in Fig. 20 show a coherent vortex with typical features of a tropical cyclone. Weak winds are apparent in the centre of the storm (the "eye") and a ring of strong winds of speed greater than 60 [m/s] around. Largest precipitation amounts occur in the "eye-wall" region about the storm centre, where horizontal convergence in the boundary layer leads to significant vertical velocities (not shown).

In the implementation of the FVM module, the mesh and connectivity requirements for FVM have been implemented using *Atlas*, along with the nearest-neighbour communication operations (halo-exchanges), parallel management, and memory layout. In particular, the *Atlas* function space *NodeColumns* is responsible for defining the memory layout (cf. Fig. 13) and parallel communication operations.

This example illustrates the value of the *Atlas* library in accelerating new developments of alternative dynamical cores.

4.4. Interpolation, remapping and filtering capabilities

In a variety of numerical applications, it is convenient to represent data in different spaces, or in the context of NWP, different grids or the spectral domain. Mapping data between representations is commonly achieved through transformation or interpolation operations and may be done for many different reasons, such as normalising variables (or fields) to the same grid, decreasing or increasing resolution, rotating the grid, etc. When mapping a variable from one representation to another, there is an intrinsic loss of information and, generally,

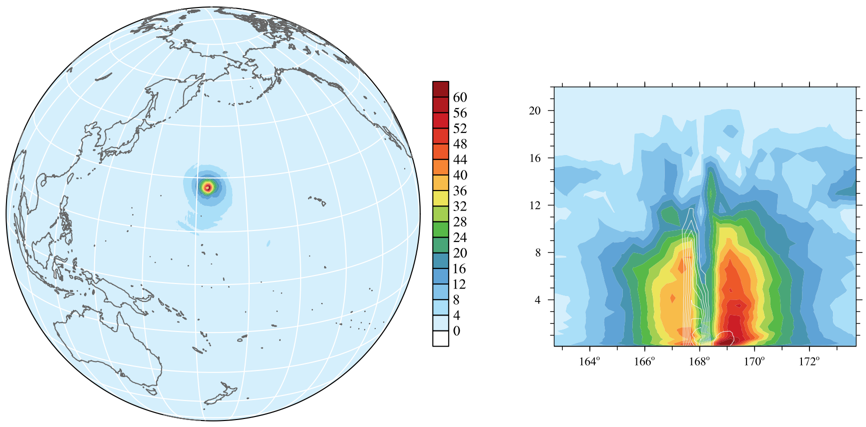


Fig. 20. FVM global simulation of a tropical cyclone at day 10. Left: surface wind ([m s^{-1}], shaded), right: zonal-height cross section at the latitude 27° N of the horizontal wind ([m s^{-1}], shaded) and precipitation mixing ratio ([g kg^{-1}], $\Delta = 0.1$ starting from zero, white contour lines). The simulation employed the O360 octahedral reduced Gaussian grid and 30 stretched vertical levels over a 30 km deep domain (only up to 22 km shown).

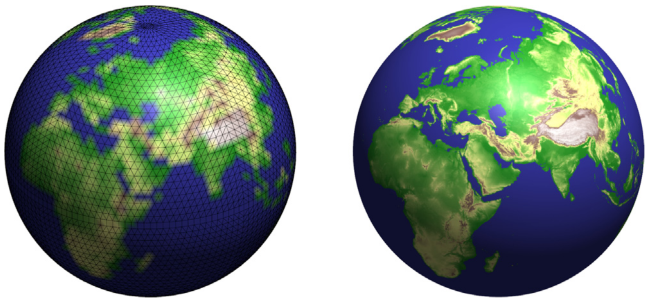


Fig. 21. Geopotential height (z [J kg^{-1}] or [m^2 s^{-2}]) at the surface mapped from octahedral reduced Gaussian grid (O1280, right) to O32 (left), with *P1* FE elements; Quadrilaterals are placed at the equator, with the North/South poles patched for visualisation.

there are no exact methods. It is therefore important to choose an interpolation methodology with the desired numerical properties. In NWP and climate modelling, properties such as *conservation* and *monotonicity* may be crucial.

Due to its design, *Atlas* provides a management of fields abstracted from the lower-level details: data locality, global reduce operations (both order-independent and not) and other parallelism concerns. In this way, it provides a solid and flexible foundation to build interpolation methods compatible with a variety of field representations. In *Atlas*, one particular interpolation method interprets fields in the classical finite element (FE) sense. The field is discretised on mesh-nodes (Section 3.3), and linear basis functions (*P1*), defined in the mesh-elements, describe a continuous linear evolution of the field between the nodes. It is these elements – typically a simple polygon (e.g. triangle or quadrilateral) –that form the units of the discretised space by associating a geometry to field values, and are therefore the fundamental linear interpolating entities by definition. Currently the *Atlas* space discretisation (mesh generation) and interpolation methods apply linear (*P1*) basis functions on 3D triangles and quadrilaterals, from which follows a natural extension to quadratic (*P2*) and cubic (*P3*) elements. Interpolation methods operate on two fields, associated to their (different) grids: a source and a target. Using a FE approach, the interpolation consists of constructing a linear operator $\mathbf{W}_{\mathrm{ij}}$, of `i` rows and `j` columns, relating the source field values $\mathbf{x}_{\mathrm{j}}$ to the target $\mathbf{x}_{\mathrm{i}}$:

$$\mathbf{x}_{\mathrm{i}} = \mathbf{W}_{\mathrm{ij}}\mathbf{x}_{\mathrm{j}}. \tag{9}$$

To maintain global *conservation* in the interpolation process, it is possible to rebalance the interpolant matrix using diagonally lumped mass matrices from the source and target meshes ($\mathbf{M}_{\mathrm{jj}}$, $\mathbf{M}_{\mathrm{ii}}$). These mass matrices can be built by distributing and averaging the mesh element sizes (area) to the source and target grid points ($\mathbf{m}_{\mathrm{j}}$, $\mathbf{m}_{\mathrm{i}}$):

$$\mathbf{x}_{\mathrm{i}} = \mathbf{M}_{\mathrm{ii}}\mathbf{W}_{\mathrm{ij}}\mathbf{M}_{\mathrm{jj}}^{\mathrm{T}}\,\mathbf{x}_{\mathrm{j}}, \qquad \begin{array}{l} \mathbf{M}_{\mathrm{ii}} := \operatorname{diag}\mathbf{m}_{\mathrm{i}} \\ \mathbf{M}_{\mathrm{jj}} := \operatorname{diag}\mathbf{m}_{\mathrm{j}}. \end{array} \tag{10}$$

A typical interpolation procedure consists in first projecting each target grid point `i` to the source mesh. This projection is implemented using a ray tracing algorithm [45], with the encompassing source mesh element found using a *k*-d tree search algorithm [46]. Subsequently, *P1* basis functions are used within that element to calculate the weights to interpolate the field value in target point `i` from the source point `j` field values [47]. Two mapping examples are provided for geopotential height and 2 m temperature in Fig. 21 and Fig. 22 respectively.

This application is another example of how *Atlas* accelerates the development of interpolation tools, by offering data structures to describe *Grids* and by the construction of a *Mesh* used in element-based interpolation algorithms.

5. Summary and future perspectives

Atlas is an innovative software library developed at ECMWF for NWP and climate services. It brings together interoperability on existing and emerging hardware with flexibility in terms of dynamical core design, spatial discretisation and pre- and post-processing options. The

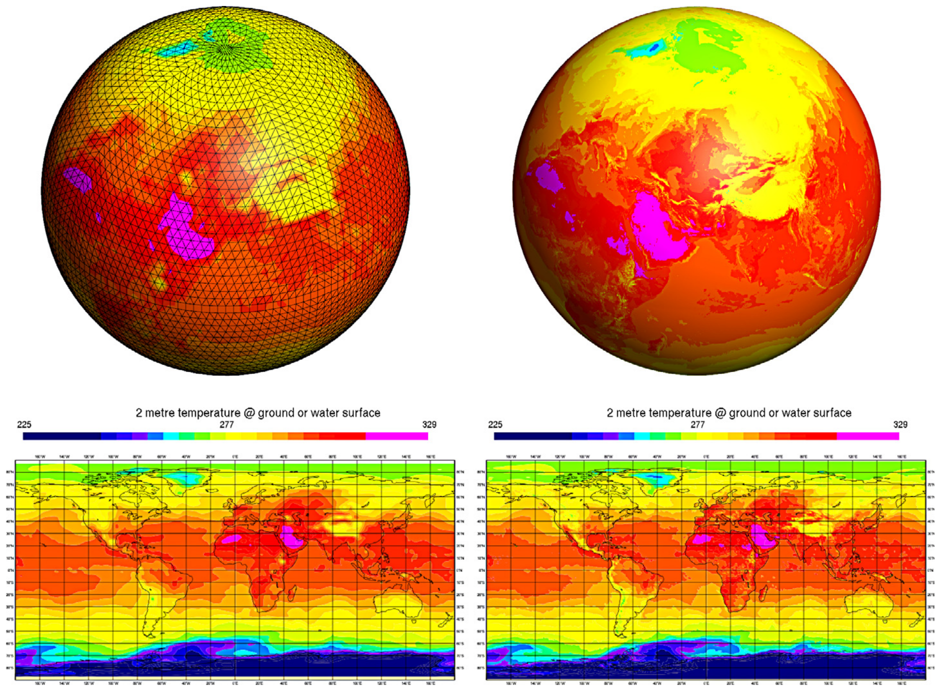

2 metre temperature @ ground or water surface
225
277
329
2 metre temperature @ ground or water surface
225
277
329

Fig. 22. 2 m temperature at ground or water surface (2*t* [K], 24th August 2016 12 UTC) mapped from octahedral reduced Gaussian grid O1280 (right) to O32 (left). Interpolation uses *P1* FE elements. Colour maps are limited to the range [225.15, 329.15]. (For interpretation of the references to colour in this figure legend, the reader is referred to the web version of this article.)

Atlas library can be used in Fortran and C++ codes, and provides parallel decomposition, threading, and OO paradigms for more flexible code design. These characteristics are essential when co-designing algorithms on emerging hardware technologies, on the route to exaflops (and beyond) computations.

Atlas supports and implements in one framework different numerical discretisations for integrating the PDEs of NWP and climate models. *Atlas* can be employed in the context of a forecast model, or in post-processing of the forecast data. *Atlas* supports the efficient mapping of fields from one mesh to another. This aids in the generation of products that need to be disseminated to external institutions, and in visualisation of NWP results.

Atlas is written in the C++ programming language and supports Fortran developments widely used in the HPC scientific community. The *Atlas* library exploits extensively the OO capabilities of C++, and various design patterns [32], making *Atlas* a modular framework that simplifies developing new features and improves maintainability.

Atlas source code is kept in a git version control repository [48] and its operating system portability is maximised by a build system based on CMake [49].

Atlas supports various software developments at ECMWF. One example is the development of FVM, an alternative dynamical core module for the IFS based on the finite volume method [16,44]. Another example in which *Atlas* is fundamental is the development of so called NWP and climate dwarfs — isolated algorithmic building blocks that constitute a NWP or climate model. These dwarfs are then scrutinised and optimised for many-core architectures both in terms of time to solution as well as energy to solution as part of the ESCAPE project [8].

Future perspectives.

In the coming years, *Atlas* will continue its integration within the ECMWF modelling infrastructure, extending its current functionalities. In particular, *Atlas* is already used by ECMWF's Meteorological Interpolation & Regridding (MIR) software package. MIR will handle the mappings between grids for the retrieval of data from ECMWF's Meteorological Archive and Retrieval System (MARS) [50], one of the largest meteorological data archives in the world.

In the research context, *Atlas* allows flexible choices for hybrid use of various discretisation techniques. *Atlas* is also expected to facilitate the integration of new Earth System Model components, each operating with different numerical discretisation techniques. *Atlas* also opens research avenues in the area of model uncertainty quantification, e.g. by supporting the generation of random fields based on the solutions of stochastic PDEs or facilitating the calculation of auxiliary flow dependent perturbations.

6. Availability

Currently, *Atlas* is an ECMWF proprietary software. A first public version is expected to be released under an open-source Apache 2 license in 2018.

Acknowledgements

This work was supported by funding received from the EU's Seventh Framework Programme (FP7/2007–2013) under the grant agreements No. 287703 (CRESTA project), by funding received from the European Union's Horizon 2020 research and innovation programme under grant agreement No. 671627 (ESCAPE project) and, in part, by funding received from the European Research Council under the European Union's Seventh Framework Programme (FP7/2012/ERC Grant agreement No. 320375). We also want to thank Anna Agusti-Panareda for providing an input file for Fig. 1.

References

[1] D. Dent, G.-R. Hoffmann, P.A.E.M. Janssen, A.J. Simmons, Fujitsu Sci. Tech. J. 33 (1) (1997) 88–101.
[2] R.H. Dennard, F.H. Gaensslen, H. nien Yu, V.L. Rideout, E. Bassous, Andre, R. Leblanc, IEEE J. Solid-State Circuits (1974) 256.
[3] O. Fuhrer, C. Osuna, X. Lapillonne, T. Gysi, B. Cumming, M. Bianco, A. Arteaga, T. Schulthess, Supercomputing Frontiers and Innovations 1 (1) (2014) 45–62.
[4] D. Leutwyler, O. Fuhrer, B. Cumming, X. Lapillonne, T. Gysi, D. Lüthi, C. Osuna, C. Schär, EGU General Assembly Conference Abstracts, Vol. 16, 2014, p. 11914.
[5] X. Lapillonne, O. Fuhrer, P. Spörri, C. Osuna, A. Walser, A. Arteaga, T. Gysi, S. Rüdisühli, K. Osterried, T. Schulthess, EGU General Assembly Conference Abstracts, Vol. 18, 2016, p. 13554.
[6] T. Schulthess, Nat. Phys. 11 (5) (2015) 369–373.
[7] N. Wedi, M. Hamrud, G. Mozdzynski, Mon. Weather Rev. 141 (10) (2013) 3450–3461.
[8] W. Deconinck, M. Hamrud, C. Kühnlein, G. Mozdzynski, P. Smolarkiewicz, J. Szmelter, N. Wedi, Parallel Processing and Applied Mathematics, Springer, 2016, pp. 583–593.
[9] A. Müller, M.A. Kopera, S. Marras, L.C. Wilcox, T. Isaac, F.X. Giraldo, Int. J. High Perform. Comput. Appl. (2017) in press.
[10] S. Valcke, Geosci. Model Dev. 6 (2013) 373–388.
[11] C. Hill, C. DeLuca, V. Balaji, M. Suarez, A. d. Silva, Comput. Sci. Engg. 6 (1) (2004) 18–28.
[12] G. Theurich, C. DeLuca, T. Campbell, F. Liu, K. Saint, M. Vertenstein, J. Chen, R. Oehmke, J. Doyle, T. Whitcomb, A. Wallcraft, M. Iredell, T. Black, A.M. d. Silva, T. Clune, R. Ferraro, P. Li, M. Kelley, I. Aleinov, V. Balaji, N. Zadeh, R. Jacob, B. Kirtman, F. Giraldo, D. McCarren, S. Sandgathe, S. Peckham, R. Dunlap, Bull. Amer. Meteor. Soc.
[13] Y. Seity, P. Brousseau, S. Malardel, G. Hello, P. Bénard, F. Bouttier, C. Lac, V. Masson, Mon. Weather Rev. 139 (3) (2011) 976–991.
[14] G. Zängl, D. Reinert, P. Rípodas, M. Baldauf, Q. J. R. Meteorol. Soc. 141 (687) (2015) 563–579.
[15] P. Bauer, A. Thorpe, G. Brunet, Nature 525 (7567) (2015) 47–55.
[16] P.K. Smolarkiewicz, W. Deconinck, M. Hamrud, C. Kühnlein, G. Mozdzynski, J. Szmelter, N.P. Wedi, J. Comput. Phys. 314 (2016) 287–304.
[17] P.K. Smolarkiewicz, C. Kühnlein, N.P. Wedi, J. Comput. Phys. 263 (2014) 185–205.
[18] C. Kühnlein, P.K. Smolarkiewicz, J. Comput. Phys. 334 (2017) 16–30.
[19] G. Mengaldo, Ph.D. thesis, Imperial College London, 2015.
[20] G. Mengaldo, D. Grazia, P. Vincent, S. Sherwin, J. Sci. Comput. 67 (3) (2016) 1272–1292.
[21] S. Marras, J. Kelly, M. Moragues, A. Müller, M. Kopera, M. Vázquez, F. Giraldo, G. Houzeaux, O. Jorba, Arch. Comput. Methods Eng. (2015) 1–50.
[22] T. Gysi, C. Osuna, O. Fuhrer, M. Bianco, T. Schulthess, Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis, SC '15, ACM, New York, NY, USA, 2015, pp. 41:1–41:12.
[23] H. Edwards, C. Trott, D. Sunderland, J. Parallel Distrib. Comput. 74 (12) (2014) 3202–3216.
[24] J.-J. Morcrette, G. Mozdzynski, M. Leutbecher, Mon. Weather Rev. 136 (12) (2008) 4760–4772.
[25] G. Mozdzynski, J.-J. Morcrette, ECMWF Technical Memorandum (721).
[26] B. Stroustrup, The C++ Programming Language, Pearson Education, 2013.
[27] http://www.uml.org.
[28] M. Hortal, A. Simmons, Mon. Weather Rev. 119 (1991) 1057–1074.
[29] N.P. Wedi, Philosophical Transactions of the Royal Society A 372.
[30] S. Malardel, N. Wedi, W. Deconinck, M. Diamantakis, C. Kühnlein, G. Mozdzynski, M. Hamrud, P. Smolarkiewicz, ECMWF Newsletter 146 (2016) 23–28.
[31] A. Staniforth, J. Thuburn, Quarterly Journal of the Royal Meteorological Society 138.
[32] E. Gamma, R. Helm, R. Johnson, J. Vlissides, Design Patterns: Elements of Reusable Object-Oriented Software, Pearson Education, 1994.
[33] A. MacDonald, J. Middlecoff, T. Henderson, L.J.-L., High Perform. Comput. Appl. 25 (1) (2011) 392–403.
[34] P. Leopardi, Electron. Trans. Numer. Anal. 25 (12) (2006) 309–327.
[35] G. Mozdzynski, Proceedings of the Twelfth ECMWF Workshop: Use of High Performance Computing in Meteorology, Vol. 273, World Scientific, 2007, pp. 148–166.
[36] H. Ritchie, Mon. Weather Rev. 115 (1987) 608–619.
[37] P. Smolarkiewicz, L. Margolin, J. Comput. Phys. 140 (2) (1998) 459–480.
[38] P. Smolarkiewicz, J. Szmelter, J. Comput. Phys. 206 (2) (2005) 624–649.
[39] D. Williamson, J. Drake, J. Hack, R. Jakob, P. Swarztrauber, J. Comput. Phys. 102 (1) (1992) 211–224.
[40] J. Szmelter, P.K. Smolarkiewicz, J. Comput. Phys. 229 (1) (2010) 4980–4995.
[41] N.P. Wedi, P.K. Smolarkiewicz, J. Comput. Phys. 193 (1) (2004) 1–20.
[42] C. Kühnlein, P.K. Smolarkiewicz, A. Dörnbrack, J. Comput. Phys. 231 (7) (2012) 2741–2763.
[43] K. Reed, C. Jablonowski, J. Adv. Modelling Earth Syst. 4 (2012) 4001.
[44] P.K. Smolarkiewicz, C. Kühnlein, W. Grabowski, J. Comput. Phys. 341 (2017) 208–229.
[45] T. Möller, B. Trumbore, ACM SIGGRAPH 2005 Courses, ACM, 2005, p. 7.
[46] J.L. Bentley, Commun. ACM 18 (9) (1975) 509–517.
[47] M.J. Fagan, Finite Element Analysis: Theory and Practice, Longman Scientific & Technical, 1992.
[48] L. Torvalds, J. Hamano, Git: Fast Version Control System, 2010. http://git-scm.com.
[49] https://cmake.org.
[50] B. Raoult, ECMWF Newsletter 72 (1996) 15–19.